APPLYING PETRI NET REDUCTION TO SUPPORT ADA-TASKING DEADLOCK DETECTION

S. Tu, S. M. Shatz and T. Murata

Software Systems Laboratory
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60680

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 Ada-tasking deadlock detection. Net reduction can ease reachability analysis by reducing the size of the net while preserving relevant properties. By combining Petri net theory and knowledge of Ada tasking semantics, we derive some specific efficient reduction rules for Petri net models of Ada-tasking. In addition, we suggest a method by which a useful description of a detected deadlock state can be easily obtained from the reduced net's information.

1. INTRODUCTION

Developers of Ada-tasking programs are faced with a number of difficult verification problems and unfortunately the complexity of many problems is known to be intractable. As an example, Taylor [1] proved that the classical (static) deadlock detection problem is NP-complete, even for a simple Ada program without select statements, branches and loops within each task's control flow.

A number of techniques have been proposed for static analysis of Ada tasking [1-6]. All are subject to the accuracy limitations imposed by static analysis [11]. We previously defined a Petri net framework for this type of analysis [2] and developed a toolkit that supports this approach [6]. The architecture of the toolkit is shown in Fig 1. The FETS (Front-End Translator Subsystem) translates Ada source or Ada Tasking Language (ATL) design descriptions [2] into Petri net format, i.e., a Petri net extended by inscriptions. We refer to the resulting Petri net as an "Ada-net". The BIDS (Back-end Information Display Subsystem) receives users' queries and presents tasking analysis results based on analysis of the Ada-net.

In this paper, we consider analysis for the classical case of deadlock detection. We use the term deadlock (even in Petri nets) to mean a system state which has no successor states. In the context of the Petri net approach, the straightforward way to detect deadlocks would be to search the reachability graph for states that have no successor states. This is very easy to automate with our toolkit system [6], but is inefficient or even intractable since the complexity for generating a reachability graph is of exponential complexity. Previously, we studied the use of Petri net structural invariants to aid deadlock detection in Ada tasking [3]. 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 detection by use of another Petri net technique, net reduction.

Recently, Long and Clarke [4] introduced a "task interaction graph" representation for creating conccurrence graphs. The goal is similar to ours, although we feel that our method has an inherent advantage of being based on a model that is both theoretically mature (and continues to be widely and actively studied) and is supported by a number of available tools. Also, our two-phase approach is distinguished by the philosophy of first deriving a semantically rich model independent of any specific analysis issue, and then reducing this model with algorithms that are designed for the specific issue of concern. This separation allows us to use known reduction theory in model optimization.

2. NET REDUCTION

Petri net researchers have developed various reduction techniques for general Petri net analysis. Petri net reduction aims to transform a given Petri net, N, into another Petri net, N', such that N' is easier to algorithmically analyze. To illustrate this, we use an example program called the gas station program [8]. 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;

Given the above 3-task program, our translator produces the Ada-net shown in standard Petri net graphical form in Fig 2. As a comparison, we also show a corresponding reduced net in Fig 3. At this time, the reader should not be concerned with the details of either net (e.g., labels, markings, etc.). The nets are shown here only to give the reader an intuitive comparison of their sizes.

We now consider two relevant issues for net reduction, preserved properties and algorithm complexity. First, all properties that are of concern must be preserved during the reduction process. This determines the correctness of any analysis using reduction techniques. For this paper, our sole interest is in deadlock detection. Second, the reduction algorithms are not helpful if they require information about the original net that is only obtainable from the net's reachability graph -- remember the goal is to avoid generating the original net's reachability graph!

The conditions that must be checked to determine if it is appropriate to apply a particular reduction rule are called the rule's application conditions. These application conditions are classified into two kinds: structural conditions and behavioral conditions. Structure conditions depend on only the local structures (connection features) of a few nodes (transitions and/or places) regardless of the marking evolution. On the other hand, behavioral conditions depend on reachable markings. Generally, when a rule is defined, the application conditions are as much as possible based on the structure of the net.

3. SOME GENERAL REDUCTION RULES

As a starting point, we first selected from [7] those reduction rules whose application conditions are simple and structural. These reduction rules were defined to be applicable to general (domain-independent) Petri nets; thus they apply to our Ada-nets. In the next section we will add some Ada-net-specific reduction rules.

Note that whenever a 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. The first reduction rule fuses two transitions h and f when f has exactly one input place p and p is an output of h. This idea has been generalized to two subsets of transitions in the following rule.

Rule 1 (Post-Fusion of Transitions)
A non-empty subset G of T is post-fusable with another subset of transitions H iff there exists a place p such that the following three conditions are satisfied:

(a) ∀feG, "fe[p] -- the only input of f is p. 
    pe"f" -- p is not an output of f.
(b) ∀heH, pe"h" and pe"h" -- p is not an input of h.
(c) ∀te T -(H∪G), pe"t" and pe"t" -- p is disconnected
    from other transitions except for those belonging
    to H or G.

Conditions (a) and (b) imply that H∩G=∅.

The operation of Rule 1 is to fuse each he H with each fe G by producing a transition <hf> such that "<hf>"="h and <hf>" = "f<"h""-[p]" (Note the use of bag sum). p and all the transitions te (H∪G) are deleted. Totally we produce |H|×|G| transitions. Fig 4 shows the Post-Fusion of transitions h1 and h2 with f1 and f2. In the reduced net, N2, 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 N2 is the same as that of the original net.

Note that the implementation complexity of Rule 1 is linear. Let n be the number of nodes in an Ada-net. It is reasonable in the context of Ada-nets to assume that the number of arcs connected with one node is bounded by a constant or to assume that the number of nodes that connect with Ω(n) arcs is bounded by a constant. Under these assumptions, the necessary number of scans on the net to find a group of transitions that satisfies the conditions of Rule 1 is bounded by a constant.

The second reduction rule fuses two transitions f1 and f2 connected by a place p that is the sole output of f1 and an input of f2 (not necessarily the sole input to f2). Moreover, f2 can be replaced by a subset of transitions in this rule.

Rule 2 (Pre-Fusion of Transitions)
A subset G of transitions may be pre-fused with a transition h iff there exists a place p such that the following conditions are satisfied:

(a) h"=p and pe"h" -- p is the only output of h and p is
    not an input to h.
(b) ∀feG, pe"f" and pe"f" -- no transition of G has p as
    an output.
(c) ∀te G∪{h}; pe"t" and pe"t" -- p is disconnected
    from all other transitions.
(d) ∀qε "h, iq"=1 -- h does not share its input places
    with other transitions.
(e) Mp(p)=∅.

The operation of Rule 2 is simply to fuse h with each fe G by substituting "f with "f + "h"-[p] and then delete h and p (Note the use of bag summation). Fig 5 shows transition h fusing with f1 and f2. The properties preserved through pre-fusion and the implementation complexity are the same as for post-fusion.

The third reduction rule is a special case of Rule 1 (Post-fusable transitions). We explicitly specify it because it is easier to apply and commonly occurring.
Rule 3 (Serial-Fusion of Transitions)
If a place \( p \) is the unique output place of \( t_1 \) and the unique input place of \( t_2 \), i.e., \( t_1^\bullet_p = t_2 \), then \( t_1 \) and \( t_2 \) can be fused into \( <t_1, t_2> \) such that \( <t_1, t_2>^\bullet_p = t_1 \) and \( <t_1, t_2>^\bullet_{t_2} = t_2 \).

Rule 4 (Parallel Redundant Places)
In a Petri net, if two places share both the same input transition(s) and output transition(s) (possibly empty), then one of the places is redundant.

By applying the above four rules, we can reduce Ada-nets, but only to a limited extent. Some domain-specific reduction rules are discussed next.

4. SOME ADA-NET SPECIFIC REDUCTION RULES

Here we adapt two reduction rules from [7] that are valid for general Petri nets but complex to evaluate. In order to detour the expensive computation for checking these rules, we use knowledge of the semantics of Ada tasking and produce some Ada-net domain-specific reduction rules. The application conditions for these rules are simplified by explicitly using information from the Ada tasking programs.

The Ada-nets are created by a translation algorithm that uses a set of Petri net templates corresponding to Ada statements [2]. In an Ada net, each place is automatically labeled by the translation program. The template of an entry call statement is shown in Fig 6(a) and that of an accept statement with and without an accept body is shown in Fig 6(b) and (c), respectively. Generally, a rendezvous is represented with the net structure shown in Fig 6(d). Details on other templates and the translation can be found in [2].

Definition 4.1
In an Ada-net, we call a place having a label prefixed with the string "begin-", "wait-", "accept-", "ack-entry-", "ack-accept-", "entry-ex-" and "select-" as begin-place, wait-place, accept-place, ack-entry-place, ack-accept-place, ex-place and select-place, respectively.

One should note that the transition fusion rules 1, 2 and 3 introduced in the previous section can be applied to Ada-nets without changing any of the place labels. These labels are good information carriers that assist us in defining the following specific Ada-net reduction rules.

Intuitively, if a place never independently inhibits the firing of its output transitions, we can simply remove this place without changing the net's behavior. This condition can be formally defined [7] and the definition leads to the following conclusion: Certifying that a place satisfies this condition in an arbitrary Petri net is equivalent to the problem of solving a set of integer homogeneous linear equations. In order to avoid solving the equations, we take advantage of knowledge of the semantics of Ada tasking and the information carried by the Ada-net's labels to allow easy recognition of some commonly expected redundant places.

Rule 5 (The Removal of Wait-places)
In an Ada-net, all wait-places are redundant places and removing them will not change the original net's deadlock characteristics.

The proof of Rule 5's correctness can be found in [9]. Rule 5 is a sufficient condition for a place to be redundant in an Ada-net. It is so simple that we can remove all the places satisfying the condition of this rule in one scan through the net, which often makes further reductions easier. Another commonly expected case of a redundant place is related to an accept-place when some task calls another task's entry twice sequentially and the first relevant accept statement has no accept body. Rule 6, introduced below, precisely describes this case.

Rule 6 (Redundant Accept-place)
In an Ada-net, let \( p \) be an accept-place corresponding to an accept statement, "accept \( X \)", which belongs to a task \( T \). If there is no other accept statement in \( T \) with the entry named \( X \), and \( p \) is a single input/output accept-place, then \( p \) is redundant if the following additional conditions are satisfied.

Let \( p^\prime = \{1\} \) and \( p^\prime = \{2\} \).

1. \( \exists q \in p^\prime, q \) is another accept-place.
2. \( \exists q_1 \in p^\prime \) and \( q_2 \in p^\prime \) such that \( q_1 \) and \( q_2 \) correspond to the statements belonging to the same task.

The removal of \( p \) will preserve the same properties of the original net as those preserved by Rule 5. This is proved in [9]. Since the suffix of every place label is a number referring to certain program statement as shown in Fig 6, we can easily check the additional condition (b) of Rule 6, using these suffix number of the places' labels.

Rule 7 (Redundant ex-place)
If an Ada tasking program has an accept body with at most one caller, then in the corresponding Ada-net structure representing this accept body, the ex-place is a redundant place.

Note that Rules 4 through 7 are all based on the same idea of a redundant place. Next, we introduce a different kind of reduction rule. If a place \( p \) is connected to two subnets, \( S_1 \) and \( S_2 \), and the tokens produced by each subnet will never "mix up" in \( p \) (i.e., if \( p \) is marked by the transitions of \( S_1 \), then no transitions of \( S_2 \) will put tokens into \( p \) until \( p \) is unmarked), then place \( p \) can be split into two places. The condition of this rule is obviously a behavioral condition, which is generally difficult to check. So this reduction rule is difficult to apply to a general net. However, in Ada-nets we can observe a typical local structure that naturally satisfies the above condition. This structure corresponds to the case of multiple entry calls to a common entry. Using the semantic information carried by the labels of places, we can give a specific reduction rule that makes a splittable-place reduction feasible in Ada-nets.

Rule 8 (Splittable place in an Ada-net)
In an Ada-net, NET, a place \( p \) is splittable if the following conditions hold,

1. \( p \in \text{begin}-, \text{ack-ex}-, \text{ack-accept}-, \text{entry-ex}-, \text{exit-ex}- \), such that
k \sum_{j=1}^{k} \text{entry-ex-} j = \text{p} \quad \text{and} \quad \bigcup_{j=1}^{k} \text{entry-ex-} j' = \text{p}'

If the conditions hold, place p can be split into k places \( p_j = \text{entry-ex-} j \) and \( p'_j = \text{entry-ex-} j' \), for \( j=1, \ldots, k \).

As an example, in Fig. 7(a) place p is split into k places, p1, p2, \ldots, pk as shown in Fig. 7(b). A proof of Rule 8's Correctness can be found in [9].

Rule 8 only considers the situation in which the accept execution body is represented by a single place. This is because in many cases the net structure of an accept body can be reduced into a single place. Every newly generated place becomes a parallel place of an ex-place and it then can be removed by Rule 4. In contrast, if the accept execution body cannot be reduced to one place, then even if one of these places splits, no further reduction rules can remove the newly generated places. Thus the splitting cannot benefit the later reduction or analysis.

We now give two additional reduction rules that are simple and useful. In original (i.e., unreduced) Ada-net models, selective-wait statements are modeled with two levels of nondeterminism [2]. Unfortunately, this modeling produces self-loops that inhibit reduction rules from being applied to the transitions involved. By reducing the selective-wait model to a single-level of nondeterministic choice, we can remove the self-loop and thus not impede further reduction operations. Fig. 8 shows the single-level model for a selective-wait with two accept alternatives, the first having two potential callers and the second only one.

Rule 9 (Reduction of Selective-wait Structures)

Let p be a select-place (in a selective-wait model) and let A be the set of related accept-places, i.e., \( A = \{ q \mid q \text{ is an accept-place and } q \in p' \} \). The following reduction can be performed:

1. Remove all output transitions of p;
2. Set \( p' = \{ t' \mid t' \in q' \} \) where \( q \in A \);
3. Remove all accept-places, i.e., all \( a \in A \).

The general reduction rules, Rules 1 to 4, do not apply to source places (places that have no input transitions). For example, the three begin-places in Fig 2 cannot be removed by Rules 1 to 4, but the existence of these begin-places leads to many uninteresting states. To avoid this, we have Rule 10.

Rule 10 (The removal of Begin-places)

In an Ada-net, let b be a begin-place (see Definition 4.1) and let \( b' = \{ t \} \). If (1) \( \forall t \in b \), and (2)\( \forall \text{pe } t \), \( p + \phi \), then b and t can be removed. Furthermore, every output place of t should be initially marked in the reduced net (since begin-places are initially marked in Ada-nets).

In an Ada-net, every begin-place only has one output transition. Intuitively, Condition (1) in Rule 10 avoids removing synchronization structures and Condition (2) avoids "over-reducing" a net and creating an isolated marked place. Putting a token into every output place of t means that we assume t has been fired. If different reduction rules are to be applied to a net, Rule 10 should be applied last since all the other rules (Rules 1 through 9) assume the places are unmarked, but Rule 10 can induce a marking.

5. DEADLOCK DETECTION AND INFORMATION PRESERVATION

Now we can explain how to do the reduction shown in Section 2, i.e., how to reduce the net of Fig 2 to that of Fig 3. First, we substitute the net structure representing the selective-wait statement with the compact structure according to Rule 9. Next by Rule 5, we remove all the wait-places, wait-ack-Operator-4, wait-ack-Pump-5, wait-ack-Pump-6, wait-ack-Pump-25, wait-ack-Operator-16, and wait-ack-Change-29. Then by Rule 7, we remove the ex-places, entry-ex-24-4, entry-ex-28-16, and entry-ex-15-6. And by Rule 6, we remove the accept place, accept-15. Now we can apply Rule 3 to many groups of transitions: 12,13, 14, 15, 122, 16, 17, 123, 124, 114, 115, 116, 117, 118, 122, 126, 127, 128, 129, 130, 131. We also repeatedly apply Rules 1, 2 and 4 and at last we apply Rule 10. The final reduced net is shown in Fig 3. Remember all of these applied reduction rules are of linear complexity.

If we tried to use the domain-independent rule for redundant places as in [7] instead of using our Ada-net specific reduction rules (Rules 5, 6 and 7), we would have to solve a system of integer linear equations having a 47x33 coefficient matrix for every place (totally 47 place). Clearly this process would result in a high computation cost.

With the reduced net, we can detect deadlocks by the reachability graph approach. The reachability graph of the reduced net of Fig 3 is shown in Fig 9; it is composed of 4 states in a chain. In 3 steps we can reach the deadlock state, "ack-entry-Customer-29." For comparison, we note that the reachability graph of the original Ada-net has 82 states. For this 82-node graph, a depth-first search analysis would require 26 steps to reach the deadlock state. The token allocation in Fig 2 shows the deadlock marking in the original-net, which can be represented by the following string


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. This interpretation is based on the deadlock marking of the original net. But, is it possible to get the same interpretation with the reduced net? The answer is "yes" because we can regenerate a full description of the deadlock marking from the reachability graph of the reduced net. To do so, we use the reduction rules as follows.
First, we apply Rule 9 (Reduction of Selective-wait Structures) to the original Ada-net if the program contains "select" statements. This results in a self-loopless net. We call this net $\text{NET}_1$ for future reference. Then we apply all the other reduction rules to all applicable nodes. The application of each transition fusion rule (Rule 1, 2 or 3) will remove some transitions and generate some new transitions. We intend to preserve the fusion history to facilitate later analysis. When some transitions are fused, their labels are concatenated. The newly generated transition (the fusion of some transitions) then is labeled with this concatenation, which is composed of some original transition labels. This kind of label concatenation immediately gives us the firing count of each fused transition, (but it does not give the firing order).

Every original transition's label in $\text{NET}_1$ will appear in the label(s) of at least one transition in its reduced net if Rule 10 is not applied. With these labels, it is easy to generate a "firing count vector" during the search for the deadlock in a reachability graph. A "firing count vector" is referred to as an m-dimensional vector (m is the number of transitions in $\text{NET}_1$), each element of which is the number of times that the corresponding transition fired. However, if Rule 10 is applied, some transition labels are loss. As we discussed in Section 4, the removal of a transition by Rule 10 means that this transition has been fired. Thus we generate another m-dimensional vector, $Y = (y_1, y_2, ..., y_m)^T$, $y_i = 1$ if $t_i$ is removed by Rule 10; $y_i = 0$, otherwise.

With these two vectors, $X$ and $Y$, we can obtain the full description of the deadlock marking in the original Ada-net according to the well-known Petri net state equation:

$$M = (C(X+Y)+M_0)$$

where $C$ is the incidence matrix of $\text{NET}_1$ and $M_0$ is the initial marking of $\text{NET}_1$.

The path from the initial state to the deadlock state in the reachability graph of Fig 9 identifies the fired transitions, $T_1$ (04, 18), $T_2$ (04, 15, 16, 17), $T_3$ (06, 17, 18, 21, 22, 23, 24, 32), $T_4$ (105, 127). Thus, we have the following "firing count vector" with respect to $\text{NET}_1$:

$$X=(0.00, 1.00, 1.00, 0.00, 1.00, 1.00, 1.00, 1.00, 1.00)$$. The application of Rule 10 yields another vector:

$$Y=(1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00)$$. By applying Equation (5.2), we can obtain the marking vector that represents the same state as that of String (5.1).

6. 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 that 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 a number of 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. These rules allow us to reduce the size of a program model while preserving the model's deadlock characteristics. Thus, any deadlocks can be identified by first reducing the model and then generating the reduced model's reachability graph. We are continuing to experiment with our prototype tool for the reduction rules. Part of our future work is to investigate the possibility of adding and/or removing reduction rules to obtain other sets of reduction rules that are defined specifically to preserve other types of model properties and analysis. Also, work is needed on more rigorous (both theoretical and experimental) study of the sensitivity of results to the order of application of reduction rules.

REFERENCES


Fig 1 The Toolkit Architecture

Fig 2 The Ada-net of the Gas Station Program at Deadlock

Label Abbreviation:
aeO4: ack-entry-Operator-4
aeP25: ack-entry-Pump-25
Fig 3  The Reduced Net Model of the Gas Station Program

Fig 4  Post-Fusion of Transitions

Fig 5  Pre-Fusion of Transitions

(a) An Entry Call

(b) An Accept with Accept Body

(c) An Accept without Accept Body

(d) A Rendezvous

NOTE ABOUT LABELS:
In Fig 6 (a), (b), (c) and (d), i is the entry call statement number; j is the accept statement number.

Fig 6  The Structures for Entry Call Statements and Accept Statements
Fig 7(a) Multiple Calls to a Common Entry

Fig 7(b) Multiple Calls to a Common Entry

Fig 9 The Reachability Graph of the Reduced Net (Fig 3)

Fig 8 A Reduced Subnet for the Selective Wait Model