ソフトウェア工学 86－T 4
（1992. 7. 21）

Graph Models for Static Analysis of Ada Tasking Programs*

Mikko Tiusanen and Tadao Murata
University of Illinois at Chicago, Chicago, IL 60680, USA

Abstract

This paper surveys graph models that have been used to support static analysis of Ada tasking programs, namely Task Flowgraps by Taylor, Task Interaction Graps by Clarke and Long, and Petri Nets. All the models are claimed to be equivalent in this context by displaying their relationship to finite state automata. This equivalence gives an opportunity to use the best of the results of any of the formalisms in any of the others.

1 Introduction

Static analysis of software aims at finding erroneous or unwanted behavioral properties based on the *description* of the software, as opposed to the *execution* of it. it tries to state that evidence of an error is (not) present. This difference is not entirely clear cut, however, since some of the static analysis methods do entail at least a kind of execution: *symbolic execution* and *state space generation* are two examples of these borderline cases.

Taylor [13] initiated the research on static analysis of Ada[1] tasking programs, which has since then grown considerably, see e.g. [2,6,7,9,12,15]. The problem addressed by the research is coping with the nondeterminism introduced by the Ada tasking constructs. Most of the approaches utilize a form of *state space generation* to verify the proper behavior of the tasking program. Very simply, the possible future behavior of the program, its state space, is made explicit. The explicit representation usually takes the form of a graph that has the set of possibly reachable states as nodes and an arc between two states when the other can be reached from the first without other intervening reachable states. The graph is sometimes called a *reachability graph* since a state of the program is reachable exactly when there is a path in the reachability graph to that state.

The reachability graph is attractive since it is a relatively complete characterization of the behavior of the model as long as the state space is finite. Therefore, many of the interesting questions about the behavior can be decided based on the graph. The practical difficulty involved is that the state space of a system, even if finite, can be much larger than can be generated in a reasonable time. A theoretical problem is that a program can conceivably have an infinite number of reachable states, which makes the reachability analysis inapplicable. As we shall see in Section 2 the latter problem can—to an extent—be solved with the age-old method of *abstracting from*, that is, ignoring some aspects of the program.

We shall discuss the graph models for static analysis of Ada tasking programs as follows: first we shall briefly discuss abstractions employed in the static analysis of Ada tasking programs in Section 2. Section 3 describes the simple example that is next used to portray each of the models. Section 7 will then present finite state automata (FSA) as a model is that unifies the other models. The FSA provide a bridge that allows the best of the results obtained by using each of the models to be transferred to benefit the others. We shall omit most of the formal details involved with the models, aiming here simply to give a taste of the models. We refer the reader interested in the more formal side to [14].

2 Abstraction

In static analysis the models do not attempt to keep track of the values of the variables even if these are relevant to the flow of control of the program. For Ada tasking programs, the shared variables, the input queues of entries, the count attribute of entries, and most of the other statements, too, are ignored. Any statement except the *synchronization actions* (e.g. entry calls, select and accept), or *choice actions* (e.g. case and if statements that contain statements of the former group) is abstracted away. That is, the other statements are considered carriers of the flow of control, mere "padding" between the statements relevant to the static analysis. This entails among other things that any ignored statements must terminate in finite time and be correct in the sense of not disrupting the flow of control in any task. Without tracking variables there will be no chance to keep tabs on families of entries or dynamic creation of tasks. Furthermore, it is assumed that there are no sub-

*This work has been supported by the National Science Foundation, Finnish Academy, the Finnish Cultural Foundation (Kalle and Dagmar Välimaa fund), the Emil Aaltonen Foundation, the Alfred Kordelin Foundation, and by the Helsinki University of Technology directly, through Ernst Wirtzen fund, and through the Foundation for Financial Support of HUT. Ps 127.

[1]Ada is a registered trademark of the US Department of Defense.

programs containing synchronizing actions, since static analysis is not necessarily able to usefully track the behavior of, say, a collection of mutually recursive subprograms.

Any choices in either kind of actions are usually modeled as nondeterministic. In state space generation this forces all the alternatives being explored. A typical case where this might lead to spurious errors being reported occurs when an `if` statement is used to determine which of two entry calls to make. The actual choice of path based on the variable values might exclude any erroneous behavior in all the relevant situations.

In essence, these abstractions attempt to include all possible behaviors at the cost of adding some that are not possible in the original program. Any tasks are usually also assumed to be activated simultaneously, though a later activation can be simulated using suitable new entries and calls to these (see [6]).

3 Example

We shall apply each of the models to the well-known program of Helmbold and Luckham describing an automated gas station [4] (ellipses denote omissions):

> "An automated gas station consists of an operator with a computer, a set of pumps, and a set of customers. ... The operator handles payments, and schedules the use of pumps with the aid of the computer. ...
>
> Each customer arrives at the gas station wanting a random amount of gas from a random pump. The customer first goes to the operator and prepays for the pump he wants to use. Then the customer goes to the pump and starts it. When the customer is finished, he turns the pump off and collects his change from the operator.
>
> The pumps have to be activated by the operator before they can be used by the customers. Each time a pump is activated, it is given a limit (the prepayment) on the amount of gas that can be pumped. When the pump is shut off, it reports the amount of gas dispensed to the operator. The pump then waits until it is activated again.
>
> Whenever he is ready, the operator may either accept a prepayment from a customer or receive a report from a pump. ... If the pump is not already in use, it is activated with the proper limit. When a pump reports a completed transaction, the current prepayment record for that pump is retrieved from the computer. The charges are computed and any overpayment is refunded to the customer. If another customer is waiting for the pump, it is reactivated with that customer's prepayment (retrieved from the computer)."

```
task body Customer is -- statement 1
  begin
    loop
      Operator.Prepay
      Pump.Start
      Pump.Finish
      accept Change
    end loop
  end Customer
```

Figure 1: Abstract gas station program: `Customer` [12].

```
task body Pump is -- statement 10
  begin
    loop
      accept Activate
      accept Start
      accept Finish do
        Operator.Charge
      end Finish
    end loop
  end Pump
```

Figure 2: Abstract gas station program: `Pump` [12].

```
task body Operator is -- statement 20
  begin
    loop
      select
        accept Prepay do
          Pump.Activate
        end Prepay
      or
        accept Charge do
          Customer.Change
        end Charge
      end select
    end loop
  end Operator
```

Figure 3: Abstract gas station program: `Operator` [12].

To make the Ada program more compact and perhaps easier to grasp we shall use the abstract version presented in [12], given in Figures 1–3. This consists of the relevant tasking commands only, and is further simplified by dropping the computer of the operator and assuming there is only one pump and one customer. Naturally, these simplifications make also the analysis task much easier. As we shall see, this program has a very benign kind of deadlock, since it is unavoidable: a fairly short simulation of the program will also find the deadlock. Since the example is so simple, it does not display all the relevant aspects of the models. Any comparison among the models should be done more carefully than simply based on their performance on this particular example.

4 Task Flowgraphs

The task flowgraph model by Taylor [13] was the first to be suggested for the purpose of static analysis of Ada tasking programs. In essence, it consists of the graphs describing the possible flow of control in each of a fixed collection of tasks (the *task flowgraphs*), and of a method of combining these to obtain a graph giving the possible flow of control of the whole set, the *concurrency graph*. A task flowgraph is obtained e.g. by abstraction from the full flowgraph of the task produced by an Ada compiler. In some cases there is a need to represent a statement by more than one node, for example, accept statements and entry call statements, since these involve waiting for rendezvous and then its completion. More specifically, an accept statement is represented by three nodes called awaiting-call, accept-engaged and accept-end, and an entry call by two nodes called call-pending and call-engaged in [13].

The task flowgraphs are then combined to get the concurrency graph. A node of the concurrency graph is labeled with the tuple containing the current state of each task. There is an arc between two nodes exactly when the Ada semantics allows one task alone or two tasks in unison to change there state to effect the corresponding change in the node labels. Initially, all the tasks are inactive: only states that can be reached from the initial node are included.

The flowgraphs of the tasks of the example are given in Figures 4, 5, and 6, and the concurrency graph is in Figure 7. This application of the model to the example is slightly better than a straightforward application would be. We have avoided generating additional states in the flowgraphs for the loop constructs. Also, we have merged the awaiting-call states with the select state for the accept statements immediately following the select or the or statements.

5 Task Interaction Graphs

A Task Interaction Graph (TIG) aims at reducing the size of concurrency graph by reducing the number of nodes of the constituent task flowgraphs. The code of a task is partitioned into *regions* that will then become the nodes of the TIG. The arcs between regions are labeled with a beginning or an end of a possible synchronization action. A region is a subgraph of the full flowgraph from a statement immediately following the task begin, an accept statement, or an entry call statement, up to and including the accept statement, entry call statement, or task end, whichever is first encountered along the flow of control. The region that starts after the task begin is called the *initial region*; a region that ends with the task end is called a *final* region. Note that a task region can have many exiting statements due to the non-determinism introduced by e.g. select statements. Also, a *pseudocode* is associated with each region: this is the Ada code of the region together with $ENTER(x)$ and $EXIT(x, y)$ pseudo-statements that act as place holders for the arcs to other regions y through relevant synchronization actions x.

The Task Interaction Concurrency Graph (TICG), of a program is then obtained much as with flowgraphs: the explicit reference to Ada semantics is replaced by a simpler rule, however. The tasks can make a move exactly when the arc labels of the TIGs out of the current regions of the tasks *match*. Obviously, the labels match if and only if one is an entry call to the accept of the other and both represent either the start or end of the statements, or, otherwise, both labels are equal. The last case can be used to model the delayed activation of some task or termination of some or all the tasks. The initial node is labeled with the tuple of the initial regions.

The above definition has a subtle error that causes it to miss deadlocks under certain circumstances. The choice actions have not been treated properly, since the model ignores a distinction between so-called *internal* and *external* choice. Internal choice refers to decisions that a task internally makes that affect its communication behavior, external choice refers to the joint decision between tasks determining which communication takes place of those possible.[2] If a task executes, say, exactly one of a pair of accept statements based on an internal decision, say, the value of a local variable, the TICG would allow either of these communications to take place, which might lead to missing a deadlock where there is no task calling the only available entry. It is not only up to the calling tasks to choose among the accept statements. In effect, the called task can appear to *refuse* to accept the call. Obviously, all the choices should appear as properly labeled arcs out of a region to other regions: these arc labels should not match any other ones, so as not to force them to synchronize. Please note that the choice involved in select statements with accept statements only *is* handled appropriately, since any one of the branches actually can be chosen.

[2]This is one of the problems that process algebras, starting with CCS by Robin Milner [8] have addressed in detail.

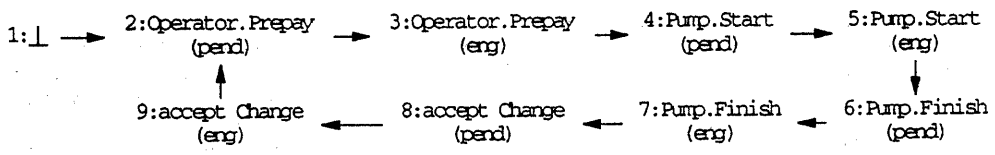

Figure 4: Flowgraph of Customer.

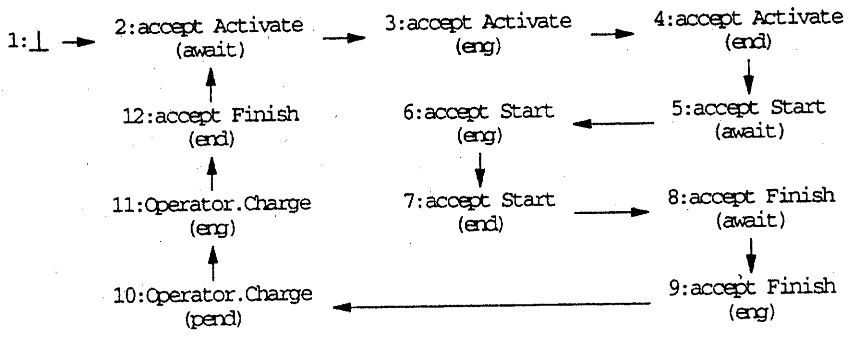

Figure 5: Flowgraph of Pump.

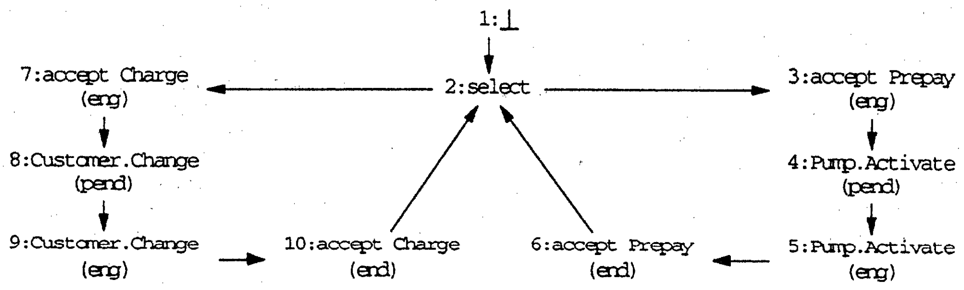

Figure 6: Flowgraph of Operator.

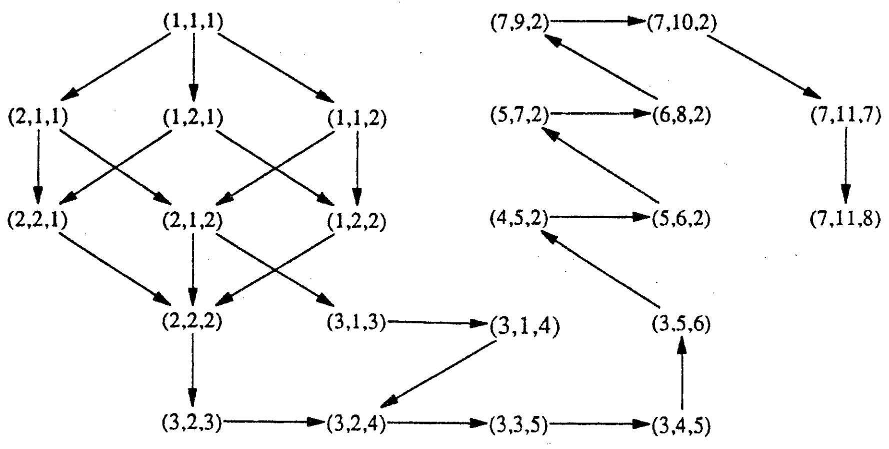

Figure 7: Concurrency graph of the example. The concurrency states consist of the states of `Customer`, `Pump`, and `Operator`, respectively.

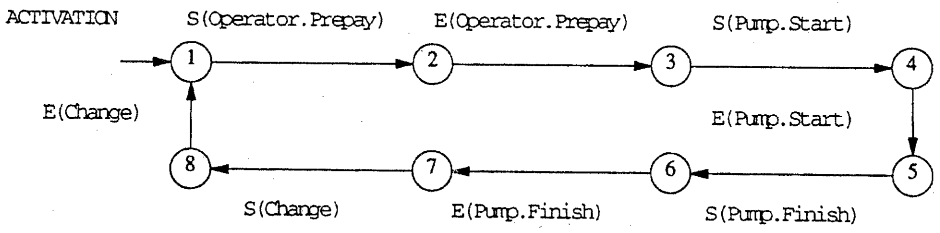

Figure 8: TIG of `Customer`.

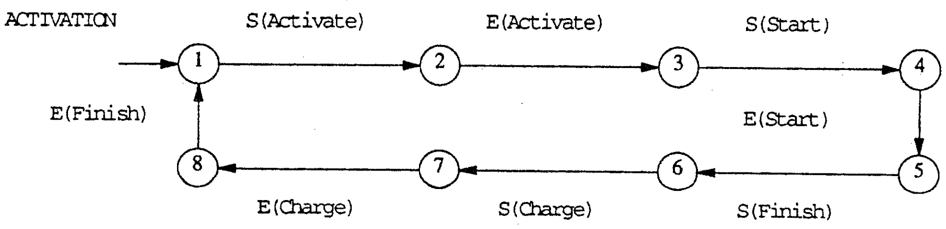

Figure 9: TIG of `Pump`.

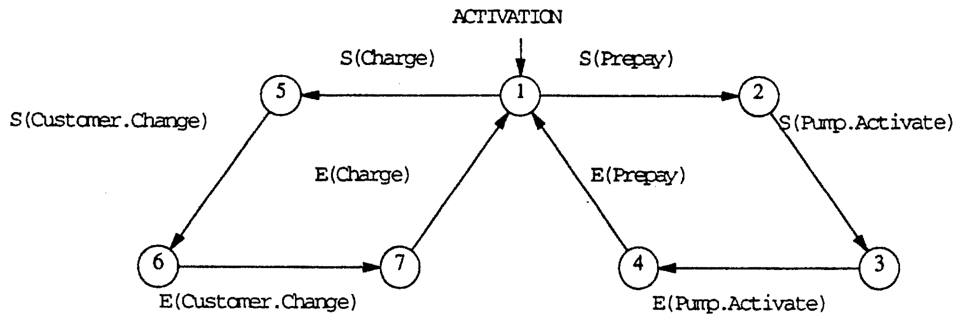

Figure 10: TIG of Operator.

$$\begin{tikzcd}
(1,1,1) \arrow[r] & (2,1,2) \arrow[r] & (2,2,3) \arrow[r] & (2,3,4) \arrow[r] & (3,3,1) \arrow[d] \\
 & (6,7,5) & (6,6,1) \arrow[l] & (5,5,1) \arrow[l] & (4,4,1) \arrow[l]
\end{tikzcd}$$

Figure 11: TICG of the example. The TICSs comprise the states of Customer, Pump, and Operator, respectively.

The TIGs of the tasks of the example are given in Figures 8, 9, and 10, and the TICG is in Figure 11. Since there are no internal choices involved in this example, its model is accurate up to the usual abstractions.

6 Petri Nets

Due to space limitations, we shall simply assume the reader to be familiar with the basic concepts and definitions of Petri nets. The necessary understanding can be obtained e.g. from [10].

An approach to the translation of Ada tasking programs to Petri nets for the static analysis is described in depth in [12]. The translation rules are defined so that the resulting Petri net is *safe*, that is, there is at most one toke in any place in any reachable marking. The Petri net model of the example is given in Figure 12. We shall not present the reachability graph of the Petri net in Figure 12 because of its size: it has 96 nodes. It should be apparent that the Petri net reachability graph so large contains redundancies. One source of redundancy are the many *intermediate* states in the Petri net model, states not relevant for the tasking behavior. For example, the translation of the select statements has three steps: first the select transition fires, then the appropriate accept is selected, and only as a separate step is the accept executed. When the reachability graph is then generated the effect of the intermediate states is multiplied, since all the interleavings of the transitions of each task with those of the other tasks must be considered. Basically, the problem here is that of too fine *atomicity* of the transitions: something that intuitively is a single action, atomic, is broken up into consequtive transitions.

It is, however, a fairly simple matter to remove most of this redundancy: the net can be transformed by removing of loop places and transitions, removing of begin places and transitions, moving the token in the initial marking to the output place of the begin transition, removing the select place and transition together with the transition testing for a pending call, and removing end places (end-case, end-select, end-if, end-block, end-accept) and transitions that have one input and output place. After any removal, the remaining net must naturally be reconnnected appropriately [11].

Tu, Shatz and Murata consider in [15] using Petri net *reductions* [1], both generally applicable rules and ones specifically chosen for the translations from Ada, to solve the particular problem of finding deadlocks. The reductions simplify the Petri net while preserving the deadlocks in its behavior. The application of the reductions to the Petri net of the example produces the Petri net in Figure 13, which has the reachability graph of Figure 14. Since this has a deadlock, the original net has one also.

7 A Canonical Model: FSA

For the purpose of static analysis of Ada tasking programs by producing the reachability graph, the above models turn out to be fundamentally equivalent. This

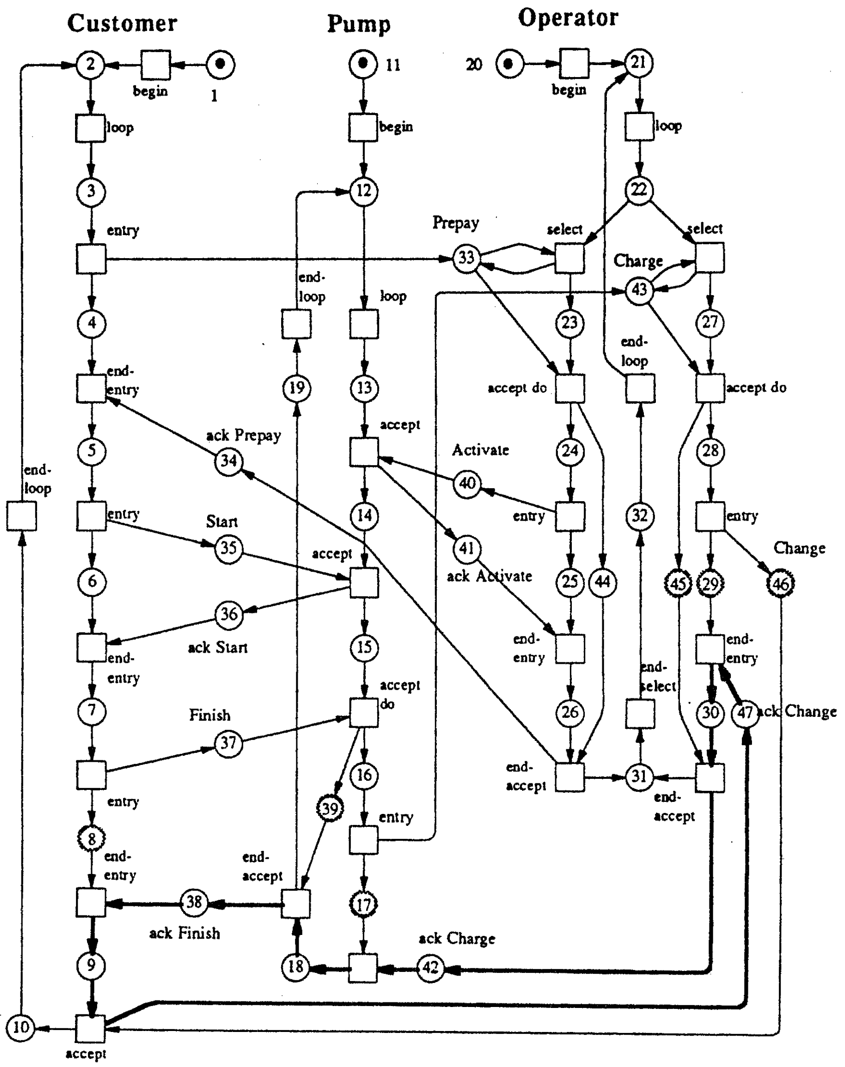

Figure 12: The Petri net of the example. The places 1, 11, and 20 are marked with one token initially, the shaded ones in the deadlock. Note the tokenless circuit through the postset of the deadlock places shown by the heavy lines. This represents a circular waiting condition: `Customer` waiting for `Pump.Finish`, `Pump` for `Operator.Charge`, and `Operator` for `Customer.Change`.

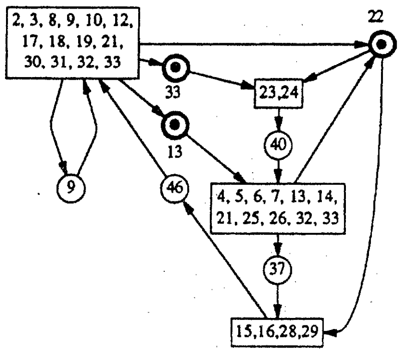

Figure 13: The reduced example net. The numbering of places and transitions refers to Figure 12. Places 13,22,and 33 are marked initially, 46 in the deadlock. Note that place 9 can never have a token, making its output transition dead.

$$\begin{tikzcd}
\{13,22,33\} \arrow[r] & \{13,40\} \arrow[r] & \{22,37\} \arrow[r] & \{46\}
\end{tikzcd}$$

Figure 14: The reachability graph of the reduced example net.

is a simple consequence of the fact that all of them are closely related to finite state automata (FSA). In order to be able to perform state space generation, the state space must be finite. Moreover, it turns out that (with the exception of the flowgraph model which incorporates the Ada semantics as such) the operations performed on the task models to get the combined behavior as a graph can be performed using fairly simple FSA operations: inverse homomorphisms and computing FSA to accept the intersection of the languages of two given FSA, as given in e.g. [5, p. 59–60]. Specifically, combining a collection of TIGs to get a TICG can be done by repeatedly combining these pairwise as follows.

First, note that a TIG without the pseudocodes *is* a FSA. Consider two TIGs (FSA), say T and T', and an entry e of T for which there is at least one call in T'. Make copies of the subgraphs of T corresponding to the bodies[3] of each of the accept statements of T for e, one for each of the entry calls in T', and index (subscript) the arc labels $S(e)$ and $E(e)$ in each by the name of the state (region) that corresponding arc of the calling task originates. (Note that for FSA the indexing above is a form of inverse homomorphism.) Each of the copies should have the initial arc originate from the state that the arc of the original subgraph has the arc labeled $S(e)$ originate; all the copies should end with arcs to the same state as the arc labeled with $E(e)$ in the original ends. The original subgraph remains in the FSA so that the construction can be repeated if some other task calls e. Then the arc labels of the calls of T' to e are similarly indexed by the name of the state from which the arc in question originates. A similar but simpler splitting must be performed on the calls, too. The common symbols labeling the arcs will be indexed both by the call state and the accept state. Note that for calls the arc labels are of the form $S(T.e)$ and $E(T.e)$ whereas accepts have labels of form $S(e)$ and $E(e)$. Therefore, in order to achieve the effect of synchronization, the arc labels of the accepts must be renamed to be the same as those of the calls before indexing.

Then reflexive arcs, which do not change the state even if taken, must be added to each state of both T and T', one for each label of the other that is *not* to be synchronized with, and each labeled with exactly this label. This allows both of the FSA perform internal actions or synchronize with yet other FSA without interference from the other. Then construct the FSA accepting the intersection of the languages accepted by the two FSA. The resulting FSA can be combined similarly with other TIGs. Finally, all the remaining subgraphs that correspond to accept statements, with or without bodies, are usually removed from the combined FSA, since the program is often considered *closed*, not open for entry calls from unknown tasks. The result is a FSA describing the behavior of the whole program, that is, a reachability graph.

Copying of the subgraphs can be described as an inverse homomorphism on FSA, as can be adding the reflexive arcs. We shall not go into the constructions involving Petri nets due to space limitations.

The equivalence of the models gives a chance to use the best results of each of the models in all the other models by translating the notions through FSA from one model to the other. It also suggest the possibility of using FSA operations for benefit in the models. The most interesting of these is perhaps the *minimization* of FSA. There is a problem, however, since the minimization typically considers two FSA equivalent exactly when these accept the same language. There are examples of FSA derived from Ada tasking programs such that the two accept the same language, yet one can deadlock even though the other cannot. This is actually closely related to the problem of internal and external nondeterminism discussed above. If one task can refuse to accept a certain call based on an internal decision where another will always accept it, the languages of the two FSA representing the tasks are the same, assuming the internal decision is modeled using ϵ-moves of the FSA. But the first task can deadlock the system by deciding to refuse the call if the rest of the tasks can proceed only if the call was accepted. There has been work done using process

[3]Each accept has a subgraph of at least one state by construction of the TIGs.

algebras, such as CCS, Theoretical CSP, and LOTOS, that directly relates to this, see e.g. [17].

8 Discussion

Whither the static analysis of Ada tasking programs? Some effort will probably go to incorporating the best ideas in any one model to the other models. But after this there will be a definite push needed to extend the scope of the analysis techniques. The natural step is to include—at least to some extent—the variables that have been ignored.

There have been many interesting methods to avoid generating the full reachability graph, while still maintaining most of the relevant properties of the system, e.g. [3,7,16]. Exploiting some of these in a way useful for static analysis is also a very interesting direction.

One of the simple ideas to improve the analysis methods is to admit the insertion of *redundancy* into the description of the program. This could involve data invariants, program invariants, anything that would present a usable description of what the program *should* do as opposed to the program text that describes what it does do. This can then be used to restrict the search of anomalies by letting the analysis program immediately notify of any contradictions between the actual and intended behavior. Sometimes an error in the system causes a bulk of a reachability analysis to be spent on irrelevant states that become reachable only as a consequence of the error.

9 Conclusion

We have presented some of the models that have been applied to the static analysis of Ada tasking programs. We have also related these to the FSA, which provides a way of using the best of each world in any of the others. A choice among the models is then a matter of secondary concerns like the availability of computer programs to support the analysis, taste, background in the theory necessary and the like. We have also discussed some of the suggested methods of improving a reachability based analysis of the models.

We are, as can easily be seen from the contents of the paper, prejudiced towards the use of Petri nets, but not without good reasons. Petri nets are fairly easy to grasp as long as one avoids trying to understand all the details of all the sometimes *ad hoc* variations of the basic model. If nothing else can be said for the Petri nets then the fact that the Petri nets with finite state spaces provide a compact and intuitive notation for FSA, each net denoting its reachability graph. Moreover, the net also describes the *concurrency* of the program or system, something that is lost in the reachability graph alone. The analysis programs are generally fairly easy to write since the model as such is simple. The difficulties lie more in the creative use of all the aspects of the simple model to achieve a goal. It should be clear that the Petri net models can be made both obviously correct to anyone interested and at least as efficient in the analysis as the other models.

Much work remains yet to be done in the field of static analysis of Ada tasking programs. We hope that this paper will stimulate further research and developments into all of the models described, many new ones, and their application to this particular problem. The work on this subject has a strong practical motivation: to help understand and design concurrent and distributed software. It is encouraging to find such a unity and diversity among the models employed so far. The unity displayed supports the soundness of the current research since all the models have a common goal to aim at to direct them; the diversity supports the comprehensiveness of the research since different vantage points produce different views, any one of which might be crucial for further progress.

References

[1] Berthelot G, Roucairol G: *Reduction of Petri Nets.* Lecture Notes in Computer Science Vol. 45: Mathematical Foundations of Computer Science 1976, Mazurkiewicz A (ed), Springer-Verlag, Berlin 1976, 202–209.

[2] Dillon LK, Avrunin GS, Wileden JS: *Constrained Expressions: Toward Broad Applicability of Analysis Methods for Distributed Software Systems.* ACM Transactions on Programming Languages and Systems 10 (1988) 3, 374–420.

[3] Godefroid P: *Using Partial Orders to Improve Automatic Verification Methods.* Proceedings of Workshop on Computer Aided Verification, Rutgers University, June 1990, DIMACS Technical Report 90–31.

[4] Helmbold DP, Luckham D: Debugging Ada Tasking Programs. IEEE Software 2 (1985) 2, 47–57.

[5] Hopcroft JE, Ullman JD: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 1979, 418 p.

[6] Long DL, Clarke LA: *Task Interaction Graphs for Concurrency Analysis.* Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, May 1989.

[7] McDowell CE: *Representing Reachable States of a Parallel Program.* Technical report 89–17, Computer Research Laboratory, University of California, Santa Cruz, California, 1989.

[8] Milner R: *Communication and Concurrency.* Prentice-Hall 1989, 260 p.

[9] Murata T, Shenker B, Shatz SM: *Detection of Ada Static Deadlocks Using Petri Net Invariants.* IEEE Transactions on Software Engineering SE–15 (1989) 3, 314–326.

[10] Murata T: *Petri Nets: Properties, Analysis and Applications.* Proceedings of the IEEE 77 (1989) 4, 541–580.

[11] Sermersheim B: *Enhancemenets, Optimizations, and Testing of Software in the TOTAL System.* Master's Project, University of Illinois at Chicago 1990.

[12] Shatz SM, Cheng WK: *A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior.* The Journal of Systems and Software 8 (1988), 343–359.

[13] Taylor RN: *A General Purpose Algorithm for Analyzing Concurrent Programs.* CACM 26 (1983) 5, 362–376.

[14] Tiusanen M, Murata T: *Models for Static Analysis of Ada Tasking Programs.* Technical report UIC–EECS 92–7, University of Illinois at Chicago, Department of Electrical Engineering and Computer Science 1992.

[15] Tu S, Shatz SM, Murata T: *Theory and Application of Petri Net Reduction to Ada Tasking Deadlock Analysis.* Submitted for publication, 1990.

[16] Valmari A: *Stubborn Attack on State Explosion.* Computer-Aided Verification 90, Series in Discrete Mathematics and Theoretical Computer Science, Vol 3, American Mathematical Society, Providence, Rhode Island, 1991.

[17] Valmari A; Tienari M: *An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm.* Proceedings of IFIP WG 6.1 Protocol Specification, Testing, and Verification, Stockholm, June 1991.