Understanding concurrent Ada programs: Relating high-level Petri nets to programs*

Ryan Stansifer
Department of Computer Sciences
University of North Texas
Denton, Texas 76203–3886

Internet: `stansifer@cs.unt.edu`

Telephone: (817) 565–2811

Fax: (817) 565–2799

October 14, 1994

*This research was supported by a faculty research opportunity grant from the University of North Texas.

1 Introduction

In this paper we show how a Petri net model of concurrent programs can be useful in software development and reverse engineering. The key is a faithful model of the programming language semantics that relates directly to the programs. We have developed a Petri model of a substantive subset of the Ada language [14, 15]. This paper demonstrates the worth of the model by showing that the model can be used in applications for which relating to the program source is important. This enables program developers to gain immediate and effective feedback about their own programs and enables key properties to be extracted automatically from existing programs. This is important in reverse engineering. A system using this model is implemented as part of the VERT (verification of real-time) system [1].

By analyzing a program being developed by a programmer, it is possible to provide feedback about the behavior of the program. Computer tools can bring inconsistencies, incongruities, unexpected interactions, and possible errors to the attention of the programmer. A good compiler does much of this, but more can and should be done, especially in the case of concurrent programs. Understanding and programming in concurrent programming languages appears to be significantly harder than in sequential languages. At the same time more and more programming is for multi-processing systems.

We are concerned specifically with the programming language Ada [16]. Ada is one of the few languages with high-level constructs for concurrent and real-time programming. Ada has a construct called a "task" which allows separate threads of execution to be done (potentially) in parallel. Tasks communicate and synchronize through what is called "rendezvous," a form of remote procedure call. The language design was commissioned by the United States Department of Defense, which has mandated its use in many applications. Since it is widely used, we have taken Ada as the initial language to study, although other high-level languages would be suitable as well. The same techniques described here can be applied to other concurrent languages as well.

We have been modeling Ada's tasking mechanism with special bipartite graphs, called Petri nets [9, 10]. Petri nets have been extensively studied in their role of representing the dynamics of concurrent systems. There are algorithms for structural analysis, construction of reachability trees, and so on. Petri net models of Ada programs have been developed before. One of the most sophisticated models is reported in [4]. This model used timed Petri nets and was able to handle the semantics of the Ada `delay` and `abort` constructs. Other Petri nets models for Ada and other languages exist [12, 13, 3]. But none concern high-level nets. The thesis of this paper is that high-level nets, especially our approach, makes the models easier to relate to the original programs. We actually use a relatively rare form of colored Petri nets that has not been used to model concurrent programming languages before.

We are interested in the worst-case behavior of concurrent programs and have used the Petri net models of Ada programs to detect the sources of possible deadlock. Its naturalness makes it easier to relate the model to the program. It is much more valuable to alert the programmer to the particular circumstances in which the program may deadlock, than to merely state that the model of the program indicates possible deadlock.

2 Ada

The Ada programming language has a construct called a "task" which allows execution to be done (potentially) in parallel. Tasks can be created statically and dynamically. Ada "entry calls" and "accept" statements are the primary means of synchronization of tasks and communication among tasks. A task calls an entry of another task by an entry call statement, which resembles a procedure call. A task accepts an entry call by executing an `accept` statement. Either task is blocked until the other is ready to rendezvous. It is possible for several tasks to call the same entry. In this case the calls are queued. After rendezvous the tasks continue separately.

Not all tasks need be created in parallel with the start of some static block or procedure. Tasks can be created dynamically at any point during the execution of the program. This is accomplished in Ada using pointers. A pointer or, in Ada terminology, an access type to the task must be declared. An instance of the task is begun when a pointer to the task is "allocated" by the `new` construct. These and other constructs are captured, and a system for translating them to Petri nets exists [1].

The details of the translation from each construct in Ada to colored Petri nets do not concern us here. One of the strengths of the model is that its high-level concepts correspond closely to the Ada program. We need only a very simple Ada program to illustrate this natural correspondence. Figure 1 shows a small Ada program called `Dine`. This program has a feeble resemblance to the well-known dining philosophers problem. In `Dine` five tasks are started concurrently: the main task; two philosopher tasks, `Hegel` and `Kant`, and two utensil tasks. The main task does nothing—it consists of the `null` statement on line number 39. Each philosopher task exhibits four phases. In the first phase, the philosopher thinks. This is represented by printing some philosophical thought using the standard Ada package `Text_IO`. After the effort of the first phase, the philosopher becomes hungry and begins to negotiate for the two utensils required to begin eating. After the utensils are acquired,

```
--  The dining philosophers
with Text_IO;
procedure Dine is
    -- declarations of task entry points
    task Hegel;  task Kant;
    task Fork1 is entry up;  entry down; end Fork1;
    task Fork2 is entry up;  entry down; end Fork2;
    -- definitions of "philosopher" tasks
    task body Hegel is
    begin
        Text_IO.put ("dialectic");  Text_IO.new_line;
        Fork1.up;
        Fork2.up;
        -- eat
        Fork2.down;
        Fork1.down;
    end Hegel;
    task body Kant is
    begin
        Text_IO.put ("categorical imperative"); Text_IO.new_line;
        Fork2.up;
        Fork1.up;
        -- eat
        Fork1.down;
        Fork2.down;
    end Kant;
    -- each fork is like a binary semaphore
    task body Fork1 is
    begin
        accept up;
        accept down;
    end Fork1;
    task body Fork2 is
    begin
        accept up;
        accept down;
    end Fork2;
begin
    null;  -- main task does nothing
end Dine;
```

Figure 1: Ada program `Dine`

then the third phase, eating, can begin. The fourth and final phase is the release of the two utensils. Although we think of the forks as passive, each utensil is also represented in the program as a task. These tasks enforce the fork's ontological status as a physical and indivisible entity.[1] And so only one philosopher can hold a fork at a time. In the program `Dine` a fork can only be used once. It would be more realistic to allow the fork to be reused, but this complicates the example needlessly.

The tasks interact in that philosophers request a fork by calling entry `up` of task `Fork1` or task `Fork2`. On line 12 and line 13 in figure 1, `Hegel` requests `Fork1` and `Fork2`. On line 21 and line 22, `Hegel` requests `Fork1` and `Fork2`. But `Fork1` is willing only once, on line 30, to accept a call on the entry `up`. Rendezvous, in this simple case, is only a synchronization of the calling philosopher and `Fork1`; no information is exchanged. This synchronization represents the philosopher's acquisition of `Fork1`. The philosopher is free to continue (with one fork), and no philosopher will be given the privilege of acquisition. The accept statement on line 35 is the same. The accept statements on line 31 and 36 represent the return of the fork.

Although one of the philosophers, starves the program `Dine` suffices to illustrate how high-level nets relate to the source code. Other language constructs illustrate the advantages of our particular colored Petri net model over other models (see [14]). This can be partially appreciated by looking at the Petri net in figure 2. The five separate threads of control are represented graphically in the figure. What was expressed awkwardly in Ada by the program `Dine` and recapped ambiguously in English above can be conveyed intuitively by the graphic representation in figure 2. This is due partly to the nature of Petri nets in general. But it is also due to the particular Petri net structure chosen, and it is due to the novel model of concurrency we have developed.

3 Modeling concurrency

The problem in modeling Ada programs with dynamic tasks, subprograms and generics is reusing the model built for a single construct (a task, a subroutine, or a generic template) while taking care that the semantics of each instance is kept distinct. Every instance of a dynamic task, a subprogram, or a generic may potentially rendezvous with some other task, and so disaster can result if the identity of the instances is confused. This problem is overcome by using color-preserving subnets. This is the special property we require of

[1] In this paper we reject the Hegelian belief in the unreality of separateness in favor of a more Kantian "*Ding-an-sich*" theory of reality.

the underlying Petri net mechanisms and is why we reject ordinary Petri nets and ordinary colored Petri nets. The use of color is crucial in representing program segments with just one subnet instead of multiple subnets. This permits the Petri net elements to correspond closely to the original program.

The general idea behind modeling concurrency is that the definition of the language specifies some constraints on the order in which the statements in the program can be executed. Any total ordering of this partial order is a possible sequence of statements that a uniprocessor might choose to execute a program. By enumerating all total orders, one arrives at all the different choices the run-time system might make in the execution of the program. (In testing the program one would want to examine all possible orders. The difficulty is that some particular computing environment might not ever choose some of the paths. It may be impractical to test all the paths, in practice. In this case a "randomly" generated collection of paths is desirable.)

The semantics of program `Dine` is represented by the Petri net in figure 2. The five distinct tasks are clearly evident as columns in the figure.[2] The complete arcs representing the entry calls from the philosopher tasks to the forks have been omitted to avoid cluttering the figure.

Figure 2 is a bipartite graph. The transitions are represented by rectangles, and the places are represented by circles. We think of the program as beginning with a token in the place labeled BGN_{Dine}. The action represented by a transition is ready to be "fired" or executed if tokens exist in all the transition's predecessor places. The transition consumes all the input tokens, and all the transition's successor places receive newly created tokens. In this manner the enabling and firing of transitions continue. Any firing sequence of this Petri net is a possible execution of the statements in program `Dine`.

The actual definition of firing a transition is more complex since the color of the tokens also plays a role. (These details are given in [14].) The approach to color can best be summarized by viewing color as the name of a potential processor. Color-creating transitions represent forks; color-consuming transitions represent joins. Paths preserving color represent sequences of actions suitable for running on a single processor. Color can then be used to remember where the flow of control returns to after rendezvous or subroutine calls.

Color-preserving paths in the model cannot be restricted to some particular color (or processor), nor is it intuitive to do so. What is required is a mechanism for a subnet to preserve a color, any color. Such a mechanism exists in a formulation of colored Petri nets that uses variables.

[2]The graphs are laid out automatically using an ad hoc algorithm.

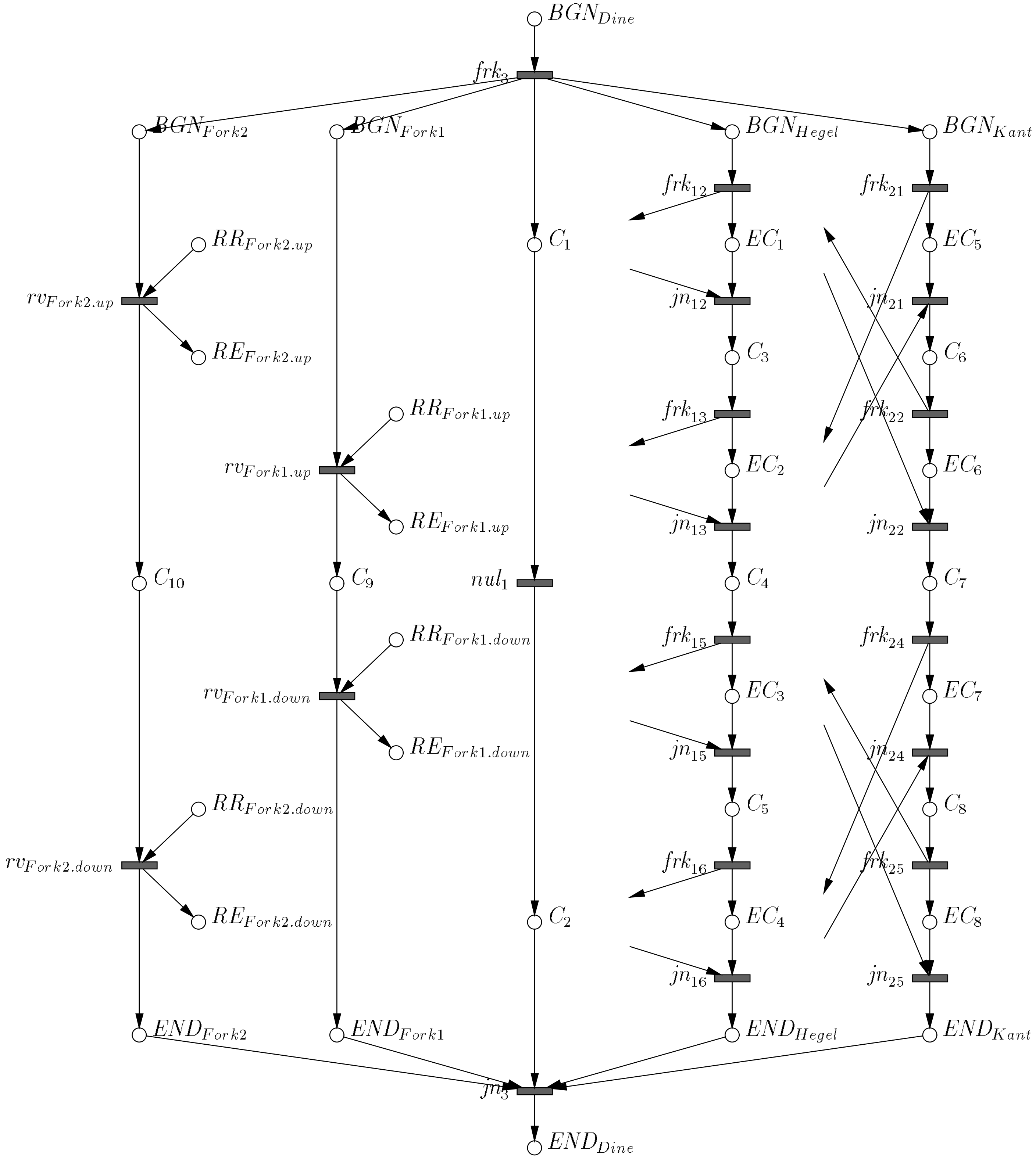

Figure 2: A portion of the Petri net model for Ada program `Dine`

transition	*family*	*attributes*
nul_1	null stmt	line no: 39, containing block: Dine, unique id: 1
frk_3	fork	line no: 3
jn_3	join	line no: 3
frk_{12}	fork	line no: 12, containing block: Hegel
jn_{12}	join	line no: 12, containing block: Hegel
frk_{13}	fork	line no: 13, containing block: Hegel
jn_{13}	join	line no: 13, containing block: Hegel
frk_{15}	fork	line no: 15, containing block: Hegel
jn_{15}	join	line no: 15, containing block: Hegel
frk_{16}	fork	line no: 16, containing block: Hegel
jn_{16}	join	line no: 16, containing block: Hegel
frk_{21}	fork	line no: 21, containing block: Kant
jn_{21}	join	line no: 21, containing block: Kant
frk_{22}	fork	line no: 22, containing block: Kant
jn_{22}	join	line no: 22, containing block: Kant
frk_{24}	fork	line no: 24, containing block: Kant
jn_{24}	join	line no: 24, containing block: Kant
frk_{25}	fork	line no: 25, containing block: Kant
jn_{25}	join	line no: 25, containing block: Kant
$rv_{Fork1.up}$	rendezvous	task name: Fork1, entry point: up, line no: 30
$rv_{Fork1.down}$	rendezvous	task name: Fork1, entry point: down, line no: 31
$rv_{Fork2.up}$	rendezvous	task name: Fork2, entry point: up, line no: 35
$rv_{Fork2.down}$	rendezvous	task name: Fork2, entry point: down, line no: 36

Figure 3: The interpretation of each transition

place	*family*	*attributes*
C_1	composition	unique id: 1
C_2	composition	unique id: 2
BGN_{Dine}	block begin	block name: Dine
BGN_{Hegel}	block begin	block name: Hegel
BGN_{Kant}	block begin	block name: Kant
BGN_{Fork1}	block begin	block name: Fork1
BGN_{Fork2}	block begin	block name: Fork2
END_{Dine}	block end	block name: Dine
END_{Hegel}	block end	block name: Hegel
END_{Kant}	block end	block name: Kant
END_{Fork1}	block end	block name: Fork1
END_{Fork2}	block end	block name: Fork2
$RR_{Fork1.up}$	rendezvous request	task name: Fork1, entry point: up
EC_1	entry call	line no: 12, containing block: Hegel, unique id: 1
$RE_{Fork1.up}$	rendezvous end	task name: Fork1, entry point: up
C_3	composition	unique id: 3
$RR_{Fork2.up}$	rendezvous request	task name: Fork2, entry point: up
EC_2	entry call	line no: 13, containing block: Hegel, unique id: 2
$RE_{Fork2.up}$	rendezvous end	task name: Fork2, entry point: up
C_4	composition	unique id: 4
$RR_{Fork2.down}$	rendezvous request	task name: Fork2, entry point: down
EC_3	entry call	line no: 15, containing block: Hegel, unique id: 3
$RE_{Fork2.down}$	rendezvous end	task name: Fork2, entry point: down
C_5	composition	unique id: 5
$RR_{Fork1.down}$	rendezvous request	task name: Fork1, entry point: down
EC_8	entry call	line no: 25, containing block: Kant, unique id: 8

Figure 4: The interpretation of some of the places

color	*attributes*
black	name: Dine, line no: 3
red	name: Hegel, line no: 3
orange	name: Kant, line no: 3
yellow	name: Fork1, line no: 3
green	name: Fork2, line no: 3
blue	name: Fork1, line no: 12, containing block: Hegel
indigo	name: Fork2, line no: 13, containing block: Hegel
purple	name: Fork2, line no: 15, containing block: Hegel
cyan	name: Fork1, line no: 16, containing block: Hegel
tan	name: Fork2, line no: 21, containing block: Kant
violet	name: Fork1, line no: 22, containing block: Kant
pink	name: Fork1, line no: 24, containing block: Kant
brown	name: Fork2, line no: 25, containing block: Kant

Figure 5: The interpretation of the colors

In these colored Petri models transitions correspond to actions of the program. Each basic action, like an assignment statement, will be represented by a transition. Colors are used to keep track of where a thread of control is initiated. So colors will be lexical sites in the program. A thread of control must preserve its color so that upon termination the appropriate program statement can determine which of possibly many instances has terminated.

A Petri net marking is a representation of the state of the concurrent system. The initial marking is the initial state and is always the one in which the place BGN_P has one black token. P is the name of the main block (compilation unit). The color black is the site that by convention designates the main thread of control.

In figure 3 all the transitions for the model of the Ada program `Dine` are listed. The transitions are divided into families. Some transitions represent forks, some represent joins, and some represent parts of rendezvous. Some of the transitions represent actions of the programs. The translation captures certain key information in each transition including the site of the construct that gave rise to the transition.

Figure 4 lists 26 of the 36 places in the model of the program `Dine`. The places are also divided into families. Some places represent the point in the Petri net where the model for tasks and subprograms begin and end. Some places represent the "glue" between other pieces. This is the composition family. Still others represent points in the model for rendezvous. Places, like transitions, contain information from the original program. This information allows markings to be interpreted in the context of the program.

Figure 5 lists all the colors used by the model of the program `Dine`. Each color is connected with a lexical site. The colors black, red, orange, yellow, and green are connected to the main program at line 3. The other colors, eight in number, are used temporarily in modeling rendezvous to keep track of the eight sites of entry calls.

The translation of Ada programs to Petri nets has been implemented in the programming language ML [6]. The information illustrated by the three figures 3–5 is automatically created by the translator given the Ada program as input. In the next section we show how to use this information to interpret properties about the program.

4 Deadlock detection

Potential deadlocks in a concurrent program can be uncovered by forming the reachability graph or the interleaving case graph [8] of the model. The reachability graph shows all the states of the program and all the possible ways to reach one state from another. If there are any nodes in the graph from which it is not possible to reach the state representing normal termination of the program, then the program has a potential deadlock. It is conceptually quite easy and, in fact, simple to implement a program to build the reachability graph for a colored Petri net. However, for programs of any practical consequence this approach is not feasible as the number of possible states of the program is quite large. In these cases it is possible to do a random walk through the graph to test for the possibility of deadlock. Furthermore, it may be possible to find structural properties of the graph which guarantee that the net is deadlock free. This has been done in the case of colorless Petri nets [7]. Further work for colored Petri nets needs to be done.

The Petri net model of the Ada program `Dine` is given in figure 2. The reachability graph for the program `Dine` is shown in figure 6. Each node in the graph represents a state in the maximal semantics [11] of the program. Each arc represents the transformation of one state to another by executing some action (transition) of the program.

Each state of the program is represented by some marking in the colored Petri net model. Figure 7 lists the first 17 of the 115 markings. Using the tables of information about the places and colors it is possible to relate each marking to the execution of the program. We will look at a few of these markings.

The first marking, node $N0$, is one black token at place BGN_{Dine}. This means that the execution of the block `Dine`, the outermost and hence main block, is ready to begin. The

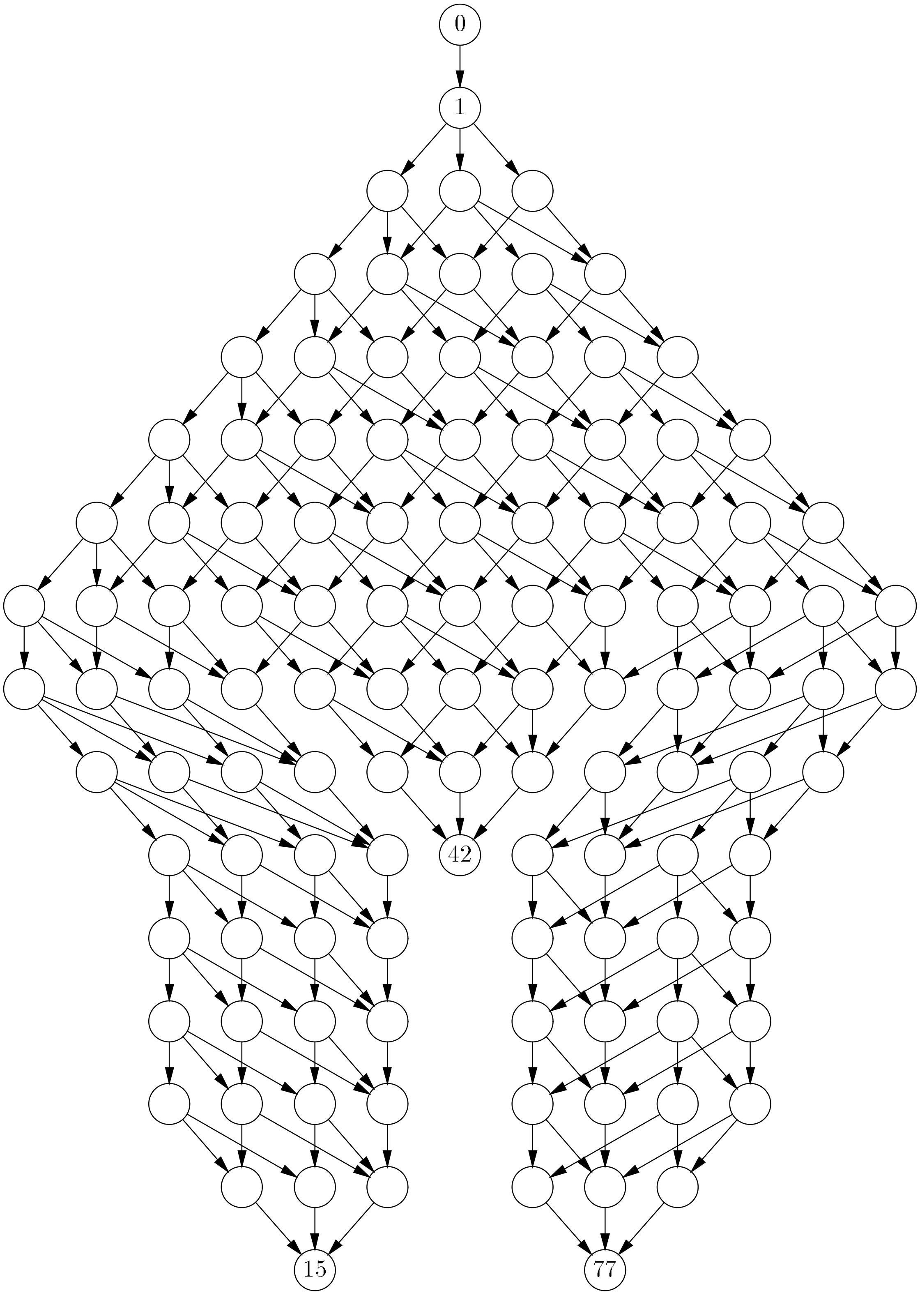

Figure 6: Reachability graph of program `Dine`

node	*marking*
N0	$BGN_{Dine} : \langle black \rangle$
N1	$BGN_{Fork2} : \langle green \rangle$, $BGN_{Fork1} : \langle yellow \rangle$, $BGN_{Kant} : \langle orange \rangle$, $BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$
N2	$BGN_{Fork2} : \langle green \rangle$, $BGN_{Fork1} : \langle yellow \rangle$, $BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $EC_5 : \langle orange \rangle$, $RR_{Fork2.up} : \langle tan \rangle$
N3	$BGN_{Fork1} : \langle yellow \rangle$, $BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $EC_5 : \langle orange \rangle$, $RE_{Fork2.up} : \langle tan \rangle$, $C_{10} : \langle green \rangle$
N4	$BGN_{Fork1} : \langle yellow \rangle$, $BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $C_6 : \langle orange \rangle$
N5	$BGN_{Fork1} : \langle yellow \rangle$, $BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $EC_6 : \langle orange \rangle$, $RR_{Fork1.up} : \langle violet \rangle$
N6	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $EC_6 : \langle orange \rangle$, $RE_{Fork1.up} : \langle violet \rangle$, $C_9 : \langle yellow \rangle$
N7	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $C_9 : \langle yellow \rangle$, $C_7 : \langle orange \rangle$
N8	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $C_9 : \langle yellow \rangle$, $EC_7 : \langle orange \rangle$, $RR_{Fork1.down} : \langle pink \rangle$
N9	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $EC_7 : \langle orange \rangle$, $RE_{Fork1.down} : \langle pink \rangle$, $END_{Fork1} : \langle yellow \rangle$
N10	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $END_{Fork1} : \langle yellow \rangle$, $C_8 : \langle orange \rangle$
N11	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $C_{10} : \langle green \rangle$, $END_{Fork1} : \langle yellow \rangle$, $EC_8 : \langle orange \rangle$, $RR_{Fork2.down} : \langle brown \rangle$
N12	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $END_{Fork1} : \langle yellow \rangle$, $EC_8 : \langle orange \rangle$, $RE_{Fork2.down} : \langle brown \rangle$, $END_{Fork2} : \langle green \rangle$
N13	$BGN_{Hegel} : \langle red \rangle$, $C_1 : \langle black \rangle$, $END_{Fork1} : \langle yellow \rangle$, $END_{Fork2} : \langle green \rangle$, $END_{Kant} : \langle orange \rangle$
N14	$C_1 : \langle black \rangle$, $END_{Fork1} : \langle yellow \rangle$, $END_{Fork2} : \langle green \rangle$, $END_{Kant} : \langle orange \rangle$, $EC_1 : \langle red \rangle$, $RR_{Fork1.up} : \langle blue \rangle$
N15	$END_{Fork1} : \langle yellow \rangle$, $END_{Fork2} : \langle green \rangle$, $END_{Kant} : \langle orange \rangle$, $EC_1 : \langle red \rangle$, $RR_{Fork1.up} : \langle blue \rangle$, $C_2 : \langle black \rangle$
N16	$BGN_{Hegel} : \langle red \rangle$, $END_{Fork1} : \langle yellow \rangle$, $END_{Fork2} : \langle green \rangle$, $END_{Kant} : \langle orange \rangle$, $C_2 : \langle black \rangle$

Figure 7: Some of the 115 markings

node	*marking*
N15	END_{Fork1} : $\langle yellow \rangle$, END_{Fork2} : $\langle green \rangle$, END_{Kant} : $\langle orange \rangle$, EC_1 : $\langle red \rangle$, $RR_{Fork1.up}$: $\langle blue \rangle$, C_2 : $\langle black \rangle$
N42	C_{10} : $\langle green \rangle$, EC_6 : $\langle orange \rangle$, $RR_{Fork1.up}$: $\langle violet \rangle$, C_9 : $\langle yellow \rangle$, EC_2 : $\langle red \rangle$, $RR_{Fork2.up}$: $\langle indigo \rangle$, C_2 : $\langle black \rangle$
N77	EC_5 : $\langle orange \rangle$, $RR_{Fork2.up}$: $\langle tan \rangle$, END_{Fork2} : $\langle green \rangle$, END_{Fork1} : $\langle yellow \rangle$, END_{Hegel} : $\langle red \rangle$, C_2 : $\langle black \rangle$

Figure 8: The dead markings

color of the token is not significant in this case as black is the one and only color that can be assigned to executing the main block.

The second marking, node $N1$, is one black token at place C_1, one yellow token at place BGN_{Fork1}, one green token at place BGN_{Fork2}, one orange token at place BGN_{Kant}, and one red token at place BGN_{Hegel}. This marking is the result of completing the action of transition frk_3. This transition represents the initiation, in parallel, of the five tasks `Dine`, `Kant`, `Hegel`, `Fork1` and `Fork2`.

Finally, we look at node $N11$, a point at which the program has made some progress. In this marking we have one black token at place C_1, one yellow token at END_{Fork1}, one green token at C_{10}, one orange token at EC_8, one brown token at $RR_{Fork2.down}$, and one red token at BGN_{Hegel}. This marking indicates that `Fork1` is ready to terminate. The other fork has completed the rendezvous at line number 35 and is now ready to accept its entry `down`, i.e., the fork is still in use. Task `Hegel` is ready to begin but has not made any progress. Task `Kant` is ready to rendezvous with `Fork2` at entry `down`.

The orange token at place EC_8 is a device used by the model to preserve colors during rendezvous. This means that the task `Kant` is in the middle of the entry call at line 25. It is an example of low-level model information that is of no particular use to the programmer. We know from the interpretation of the color brown and the place $RR_{Fork2.down}$ that the task `Kant` is ready to rendezvous with `Fork2` at entry point `down` from the entry call at line 25. To avoid confusing the programmer who may care little about the underlying model, the raw marking should be processed and the state of the program reported in terms of the program itself.

A normal program makes continual progress until the main program is ready to terminate. This would be indicated by a marking in which a black token is found at the end of the main program. Such a marking is not found among the 117 possible markings for `Dine`. This is a serious problem as it means the program cannot in any circumstances terminate normally. In general, the potential for deadlock is evident by the nodes of the graph that have outdegree 0. These are the so-called "dead" markings.

In the reachability graph of the Ada program `Dine` there are three dead markings. These markings are shown in figure 8. Again, using the tables of information about the Petri model we can interpret these markings in terms of the program.

$N15$ Tasks `Fork1`, `Fork2` and `Kant` are all ready to terminate. The main task is waiting for `Hegel` to finish so that the whole program may terminate. The task `Hegel` is blocked waiting for `Fork1` to accept the entry call `up`.

$N42$ The main task is waiting on all four other tasks so that the whole program may terminate. The tasks `Fork1` and `Fork2` are both blocked waiting to accept a `down` entry call, i.e., both forks are in use and are waiting to be returned. The task `Kant` is blocked waiting for `Fork1` to accept the entry call `up`. The task `Hegel` is blocked waiting for `Fork2` to accept the entry call `up`.

$N77$ Tasks `Fork1`, `Fork2` and `Hegel` are all ready to terminate. The main task is waiting for `Kant` to finish so that the whole program may terminate. The task `Kant` is blocked waiting for `Fork2` to accept the entry call `up`.

Markings 15 and 77 are really the same situation with the role of the philosophers reversed. Marking 42 is the classic danger in the dining philosophers problem. Each agent has acquired a resource but needs the resource acquired by the other in order to continue. Thus the system is deadlocked.

Although the current system is not designed to interpret the markings, this is not difficult to do. All the necessary information is carried as part of the elements themselves. In this way markings of the underlying Petri model can be used to explain the circumstances around deadlock in an Ada program.

5 Conclusion

Automated analysis of a computer program with the intention of reconstructing the high-level concepts that motivated the original programmer is an impossible goal. But automated analysis of programs with the intention of revealing key properties is possible. In this paper we describe an analysis system for Ada programs. We describe how the system can be used to understand the concurrency in Ada programs in terms that are quite concrete. This is useful in developing, compiling, verifying and reverse engineering Ada programs.

Representation of concurrent programs as colored Petri nets reduces both the complexity of the net models and the effort required to analyze the models. In this representation there is a one-to-one mapping between threads of control and colors. We are convinced that the naturalness of the model makes it a good candidate for modeling the semantics of concurrent programs. Some evidence for this is the easy manner with which it is possible to describe a marking in the model as a state in the Ada program.

No computer system can ever answer the really interesting questions like:

> Does this program ever deadlock?

or

> Is this task designed to be used in these circumstances?

Getting what information that is possible automatically makes human programmers more productive. We have shown in this paper a computer system designed to answer questions like:

> What are the sets of events in which the program may deadlock?

or

> Is this task used in these circumstances?

The methodology introduced in this paper is a step towards automated analysis of concurrent software systems. Further study must be done to test how feasible the approach is for large programs. We currently are examining parallel algorithms for performing the analysis of colored Petri nets [5].

The system as implemented so far provides a lot of data about the program but does not cater to the programmer and is not integrated in a software development or analysis environment. Based on the model, we envision temporal logic querying [2, 17] and meaningful responses tied graphically to the program text. The key to understanding existing programs and debugging developing programs is to provide fast, convenient, and useful information about the behavior of the program.

References

[1] M. Beaven, B. Elmore, D. C. Marinescu, and R. Stansifer. VERT—Verification of real-time programs. In *Proceedings of the 15th Annual International Computer Software and Applications Conference*, pages 618–625. IEEE Computer Society, Los Alamitos, CA, September 1991.

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

[3] M. Gourgand and D. Hill. Petri nets modelling on transputers with OCCAM 2. In *Proceedings of the 1990 European Simulation Symposium*, pages 143–147. 1990.

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

[5] D. C. Marinescu, M. Beaven, and R. Stansifer. *A parallel algorithm for computing invariants of Petri net models*, pages 136–143. IEEE Computer Society Press, Los Alamitos, CA, December 1991.

[6] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[7] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlock using Petri net invariants. *IEEE Transactions on Software Engineering*, SE-15(3):314–326, March 1989.

[8] E. Olderog. *Nets, Terms and Formulas.* Cambridge University Press, Cambridge, 1991.

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

[10] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, Berlin, 1985.

[11] A. Salwicki and T. Müldner. On the algorithmic properites of concurrent programs. In E. Engeler, editor, *Logic of Programs*, volume 125 of *Lecture notes in computer science*, pages 169–197. Springer-Verlag, Berlin, 1981.

[12] 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(5):343–359, December 1988.

[13] S. M. Shatz, K. Mai, C. Block, and S. Tu. Design and implementation of a Petri net based toolkit for Ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):424–441, October 1990.

[14] R. Stansifer, M. Beaven, and D. C. Marinescu. Modeling concurrent programs with colored petri nets. *Journal of Systems and Software*, 26(2), August 1994.

[15] R. Stansifer and D. Marinescu. Petri net models of concurrent Ada programs. *Microelectronics and Reliability*, 31(4):577–594, 1991.

[16] United States Department of Defense. *Reference Manual for the Ada Programming Language*. United States Government Printing Office, 1983.

[17] W. G. Wood. A case study of the application of temporal logic to system specification. In *Technical Review 1988*, pages 122–140. Software Engineering Institute, Carnegie Mellon University, 1988.