UNIVERSITY OF CALIFORNIA

Irvine

Using Static Concurrency Analysis to Understand the Dynamic Behavior of Concurrent Programs

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

by

John Thomas Self

Committee in charge:

Professor Richard N. Taylor, Chair

Professor Debra J. Richardson

Professor Richard W. Selby

1996

The dissertation of John Thomas Self is approved,

and is acceptable in quality and form for

publication on microfilm:

Committee Chair

University of California, Irvine

1996

Dedication

This dissertation is dedicated to my parents, Wayne and Mildred Self, and to my aunt, Margaret Karas. They inspired a lifelong love of learning that led me to this goal.

Contents

List of Figures . . . . **vi**

List of Tables . . . . **vii**

Acknowledgements . . . . **viii**

Curriculum Vitae . . . . **ix**

Abstract . . . . **x**

Chapter 1 Introduction **1**
- 1.1 Terminology 3
- 1.2 Relationship between concurrency and parallel execution 6
- 1.3 Relationship between concurrency and real-time reactive systems . 7
- 1.4 Problems with concurrent programs 7
- 1.5 A hybrid approach 11
- 1.6 Organization 13

Chapter 2 Related Work **15**
- 2.1 What is an execution state-space model? 16
- 2.2 Important characteristics of debugger state-space models 17
- 2.3 Criteria for comparing execution models 19
- 2.4 A framework for comparing concurrent debugger execution models 26
- 2.5 Comparing execution models 28

Chapter 3 A Hybrid Approach **46**
- 3.1 Exploring the state space. 51
- 3.2 Building a TICG to represent a traced execution 53
- 3.3 Using SCA to expand the initial TICG 59
- 3.4 Incremental Forward Generation of States 61
- 3.5 Backward Generation of States 63
- 3.6 Limiting the TICG portion generated or displayed 71
- 3.7 Task Subsetting 72
- 3.8 Data Tracing 73
- 3.9 Correctness of modifications made to TICG model 74

Chapter 4 Examples **79**
4.1 Using the TICG to Understand an Execution 79
4.2 Combining Execution Tracing with SCA 86
4.3 Debugging Data Races 93

Chapter 5 System Architecture **98**

Chapter 6 Conclusion **101**

Bibliography **105**

Appendix A Algorithms used in constructing TICGs **112**
A.1 Converting a trace to an equivalent TICG 112
A.2 Building internal representation for TIGs 116
A.3 Algorithm for task subsetting 123
A.4 Generating the predecessors of a state 124
A.5 Calculating nodes to level N 126

Appendix B Gas station example after attempted corrections **128**

List of Figures

3.1 Two versions of Dining Philosophers. 49
3.2 TICGs built for good and bad versions of two dining philosophers. 50
3.3 instrumentation code inserted at start of each task 57
3.4 instrumentation code inserted at each accept 57
3.5 instrumentation code inserted at each entry call 58
3.6 instrumentation code inserted at the end of each task 58
3.7 instrumentation code inserted for non-rendezvous tasking interactions. 58
3.8 instrumentation code inserted to trace variable accesses. 59
3.9 Similar concurrency paths leading to deadlock 60
3.10 TICG with infeasible states generated by backwards generation. . 64
3.11 Infeasible states generated by backwards state generation in two dining philosophers example. 65
3.12 TIGs for a fork and philosopher. 66
3.13 A feasible state and the infeasible state generated as its predecessor 67
3.14 An infeasible state. 68

4.1 Code for Elevator Controller and Elevator Package 81
4.2 Code for Elevator Simulator Task and Main Program 82
4.3 System output for failed execution of elevator. 83
4.4 Operator task for Initial Version of Gas Station 87
4.5 Pump and Customer Tasks for Initial Version of Gas Station . . . 88
4.6 System output for initial gas station deadlock. 89
4.7 System output shown partial TICG built by SCA for patched gas station. 91
4.8 Code for Data Race Example 94
4.9 System output for failed execution of shared data access example. 95

5.1 System Architecture 98

List of Tables

1.1 Concurrent programming model vs. parallel architectures 6

3.1 Results of experiments to determine number of infeasible states generated for several popular examples. 69

4.1 Elevator TICG size with and without subsetting 84

Acknowledgements

I wish to thank Richard Taylor, my advisor, for giving me the freedom and support to allow me to explore different directions with my research and helping me choose when I needed it. I am also obliged to Debra Richardson and Richard Selby for their guidance in developing this work, and to Michal Young and David Levine for their work on CATS which provides the foundation for my work.

I also want to acknowledge the many friends I have made in the ICS department who have made my many years of graduate school not only tolerable, but one of the happiest times of my life.

This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0218. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Curriculum Vitae

April 9, 1965	Born San Diego, California
June 1988	B.A. in Microbiology, University of California, San Diego
	B.A. in Computer Science, University of California, San Diego
June 1990	M.S. in Information and Computer Science, University of California, Irvine
June 1996 (expected)	Ph.D. in Information and Computer Science, University of California, Irvine
	Dissertation: *Using Static Concurrency Analysis to Understand the Dynamic Behavior of Concurrent Programs*

Publications

John T. Self and Richard N. Taylor
"Using Static Concurrency Analysis to Instrument Concurrent Programs for Dynamic Debugging"
In Barton P. Miller and Charles McDowell, editors, *Proceeding of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 263–266, May 1991. Position Paper.

John T. Self and Richard N. Taylor
"A Hybrid Approach for Debugging Concurrent Software"
In *Proceedings of the Fourth Annual Irvine Software Symposium*, pages 13–24, April 1994.

John T. Self and Richard N. Taylor
"Using Static Concurrency Analysis to Understand the Dynamic Behavior of Concurrent Programs"
Submitted to *ACM SIGSOFT '96: Fourth Symposium on the Foundations of Software Engineering* October 1996.

Abstract of the Dissertation

Using Static Concurrency Analysis to Understand the Dynamic Behavior of Concurrent Programs

by

John Thomas Self

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1996

Professor Richard N. Taylor, Chair

A variety of approaches for debugging non-deterministic concurrent software have been proposed. Most are based on either dynamic trace collection or static analysis. We argue that integration of static and dynamic analysis techniques enables the user to better understand the state space of a faulty concurrent program, and this will aid debugging. This dissertation presents our work in developing a debugging methodology with two components; a framework for representing the state space of concurrent programs, and a set of static and dynamic techniques for exploring this state space. Our focus is currently on Ada programs, but the approach is applicable to any language using the rendezvous model of concurrency.

We provide a visualization of the state space using the Task Interaction Concurrency Graph (TICG). The TICG is a data structure normally built during static concurrency analysis. We capture traces describing the concurrency related events that occur during a particular execution, then build a TICG to represent this execution. The user can then explore this TICG, and expand it further using static concurrency analysis to determine what other executions the could have occurred. We also provide a means of limiting the portion of the state space generated, and of limiting the analysis to a subset of the tasks that comprise the system under test.

This synergistic combination of static and dynamic techniques allows analysis of larger and more complex systems than is possible with conventional static concurrency, yet retains many of the benefits of automation and guidance provided by the formal models used in static concurrency analysis.

Chapter 1

Introduction

Concurrent (multitasking), non-deterministic programs have proven to be very difficult to debug. It has been suggested that the lack of good debugging tools discourages programmers from using available concurrent programming languages, because they fear they may encounter faults which they may be unable to eliminate [BLP91]. There are several new characteristics of concurrent programs that make them especially difficult to debug:

- multiple threads of control.

  Concurrency implies that there is more than one thread of control, and this adds incorrect interactions between tasks to the list of potential sources of errors. These types of error simply cannot occur in programs with a single thread of control, and most programmers are more familiar with sequential programming and debugging.

- non-determinism

  Non-determinism is inherent in most models of concurrency. It can be desirable, adding an opportunity for optimization, but it can also be the source of many problems when it is unintentional. It requires the programmer to think of all potential executions that may result from the non-determinism, rather than the more familiar single input-single execution model of sequential programming.

- lack of repeatability.

  All debugging requires that one be able to examine the state transitions that a program makes as it executes. The common sequential debugging methodology involves re-running the program under the debugger using the inputs which caused the failure. The program state is then carefully examined as the program is repeatedly executed in a controlled environment.

  Non-determinism may make it impossible to replicate a failed execution. Particularly frustrating are failures which occur infrequently during normal execution, and perhaps not at all when run under the debugger, due to timing differences.

In this dissertation we will focus on programs written in the Ada language and use Ada terminology but the techniques are more widely applicable. Any language with rendezvous style synchronization should be easily modeled, while other languages for which reachability graphs can be generated could be dealt with by modifying the framework.

In designing a debugger for concurrent programs one of the most crucial decisions is how to model program executions. Programmers use debuggers to gather information about program executions and use this to locate program faults. Debuggers build models of program executions, and use that to present information to the user about the state of the program.

In conventional sequential programs the state of an executing program consists of the values of all variables and the program counter which indicates what statement is currently executing. Concurrent programs are more complicated because of the multiple threads of control within a single program. Each thread of control may have its own set of variables. More importantly, concurrent programs often have ordering and timing properties that affect correct execution. Sequential programs will generally produce correct results regardless of processor speed. Variations in processor speed can affect the correctness of concurrent programs.

In this dissertation we concentrate on debugging problems related to functional correctness, rather than performance. We will consider a program correct if it produces the correct outputs, even if it runs very slowly. When concurrency is used in an effort to speed up execution determining the performance bottlenecks is crucial, but the techniques used often differ significantly from those for functional correctness, so we will defer that discussion.

1.1 Terminology

Because researchers within the concurrent debugging community have not adopted a single set of terminology, and indeed often conflict, we will define the most important terms here and use them consistently. Note that these may differ from the terms used within a particular paper.

- **Thread** or **Process** – An independent thread of control within a system of cooperating processes. Some use the term thread to refer only to the case where each thread of control shares an address space, but we use the more general meaning here. We will use both terms because they are so commonly used interchangeably.

- **Non-determinism** – A property of many concurrent programs that results in a single program potentially having different executions even when given the same external input data. Non-determinism may be strictly internal, or it may be external where running the program with the same input results in different outputs. It may be intentional (for reasons discussed below) or it may reflect unintended faulty or missing synchronization.

- **Concurrency** – Concurrent programs are structured as systems of independent cooperating threads of control. These threads run independently until these is a need to interact through synchronization or communication. These interactions may take many different forms, including shared variables, message passing, rendezvous, mailboxes, semaphores, or monitors. The relationship between concurrency and parallel execution and real-time systems is discussed below. Concurrent programs often exhibit non-determinism.

- **Probe Effect** – First coined by Gait [Gai86] this term refers to the change in execution of a non-deterministic program when it is run with monitoring enabled. Because monitoring an execution involves some overhead it can change the relative execution speeds of the threads in the program under test. This can result in failures disappearing when the faulty program is run under the debugger, or timing problems that won't appear in the normal environment. If either of these cases occurs it can may be impossible to duplicate a failure under the debugger, meaning the facilities of the debugger can't be used to diagnose the failing execution. The Probe Effect has also been referred to as the "Heisenberg Uncertainty Principle for Software," and the faults which display this characteristic as "Heisenbugs."

The next three definitions are adopted from Miller and Choi [MC88] who borrowed them from the fault-tolerant computing community. These seem to be the definitions most often used by those in the debugging world, but some use the terms differently.

- **Failure** – The first externally visible indication that a program is incorrect. This is usually a wrong value printed out, or a message from the run-time system indicating execution has gone astray.

  Program failures only make sense when we compare them against some intended behavior of the system, such as an explicit or implicit specification.

- **Error** – An erroneous internal state which results in a failure. The error could be an incorrect value for a variable or the program counter executing in the wrong place.

- **Fault** – The algorithmic initial cause of an error. This is the "bug" that leads to the error.

- **SCA (Static Concurrency Analysis)** – A class of techniques for detecting concurrency faults without executing the program. We generally use this term to refer specifically to those techniques based on constructing a model of program concurrency state-space, and examining this model for faults such as deadlock and race conditions.

1.2 Relationship between concurrency and parallel execution

Table 1.1: Concurrent programming model vs. parallel architectures

	programming model	
	sequential	concurrent
uniprocessor	conventional programs	time-slicing
multiprocessor	automatic parallelization	explicit parallelism

Concurrency and parallel execution are related, but orthogonal concerns. Concurrency is a programming model, while parallel execution is an aspect of the machine on which our program runs. As table 1.1 shows it is possible to combine these two factors in any of the four possible ways. Sequential programs running on uniprocessor machines are the most common, and very suitable for many programs. It is often desirable to run sequential programs on a parallel, multiprocessor machine to gain extra performance without having to re-write the code, or deal with the problems of concurrency. Parallel machines often use concurrent code to gain additional large grain parallelism that a compiler may be unable to discern and use for optimization. Finally it may be desirable to write concurrent programs, even if we have only a single processor, for reasons of conceptual clarity that will be discussed below. In this case parallelism can be simulated using a run-time system that interleaves the individual threads of control.

1.3 Relationship between concurrency and real-time reactive systems

It is important to differentiate between real-time, reactive systems and concurrent programs. Reactive systems are those which interact with, and control objects which are external to the computer system. As discussed above a concurrent, non-deterministic programming style may be useful in reactive systems, but there are a few important characteristics that are important for real-time systems, but not for all concurrent systems. As Gligor notes [GL83]

"In real-time programming the programmer must be able to control precisely the sequencing of operations and, possibly, of processes. By contrast, in concurrent programming the goal is to hide the precise sequencing of operations from the user. Similarly, in real-time programming the programmer must be aware of the detailed timing and control characteristics of the external devices, whereas in concurrent programming these characteristics are generally hidden from him."

1.4 Problems with concurrent programs

While concurrency and non-determinism are powerful abstractions, with this power comes additional sources of errors, and further problems in detecting and fixing these errors.

Each thread in a concurrent program can contain any of the faults commonly found in a sequential program (e.g. referencing the wrong variable, array indexing errors, incorrect branch tests, variable out-of-range errors, etc.) If the entire fault is limited to a single thread the methods used for debugging sequential programs may suffice, but even in this case the concurrency may hinder our efforts. For example tracing causality using manual program slicing [Wei82] may be more difficult because the programmer must determine whether any other threads could modify the variables we are slicing on. In addition the error may manifest itself as an incorrect value in a different thread than the one which actually contains the fault.

The Probe Effect limits the usefulness of the conventional debugging approach of stopping program execution and examining program state. Even modifying the program so that it outputs portions of its state as it runs may change timing enough to make the Probe Effect a great hindrance.

Synchronization faults are a category of problems which are not even possible in sequential programs. We can divide this category in two classes: timing errors (race conditions) and liveness errors (livelock and deadlock.)

Timing errors are possible because individual threads proceed at independent and varying rates. This is normally a useful characteristic because it can allow for implementation efficiency. It can be problematic if the programmer does not include proper synchronization, and program correctness depends on sequencing that is not assured by the design of the program. The most common form of timing error is the race condition. This occurs when more than one thread accesses a variable, and at least one of the uses is a write of the variable.

If the write is not synchronized with the reads it may be the case that one of the reads will return a different value depending on whether it occurs before or after the write. This will cause different output values to be computed, and this may make the program results incorrect. This type of error is an unintended source of non-determinism, and can be very difficult to debug because it may occur rarely (perhaps related to interrupts from outside sources) and the Probe Effect may make it difficult to replicate.

Liveness errors such as deadlock and livelock manifest themselves not in incorrect output, but in the computation failing to proceed to completion. Deadlock occurs when all threads are blocked waiting for events which can never occur. Global deadlock is a readily recognizable state of a concurrent system, and can be detected by the run-time system. Absence of deadlock is normally an implicit specification of a concurrent program. Livelock occurs when at least some threads continue to execute without blocking, but no real progress is being made. Livelock is dependent on the specification of the desired program behavior.

Housh and Cuny [HC88] divide interprocess communication errors in message passing programs into three classes:

- Sequencing errors. Occur when the right processes communicate, but in the wrong order.
- Missing communication errors. Occur when only some processes communicate that should.
- Extraneous communication errors. Occur when processes should communicate information, but fail to do so.

Zernik and Rudolph [ZR91] group correctness bugs in shared memory programs into the following four categories:

- Using global variables instead of local ones. This leads to improper sharing.
- Avoiding the minimal required synchronization.
- Locking complex objects. If two processes P1 and P2 need to acquire both locks A and B, we can encounter deadlock if P1 acquire A and P2 acquires B, then they wait to acquire the missing lock
- Not locking complex objects. If the user omits locking complex objects inconsistency bugs may occur when two processes update parts of a data structure in ways that conflict.

Another way to look at the types of errors we encounter is by what types of queries the debugger user may want to make. Feldman [FB88] suggests an example of a parallel program maintaining a tree structure. If a cycle is introduced in the tree, the programmer will want to ask "When did this tree first get an illegal cycle." The difficult parts of the question are: "when", "*this* tree", "first get", and "cycle". Answering "when" requires knowledge about the temporal properties of the execution. Identifying "which tree" we are referring to may be difficult where address spaces overlap, and the names of a particular data item may be different in each thread. Answering when did the cycle "first arise" is difficult because each thread has its own conception of when relative to its progress. The concept of "cycle" requires that we be able to exactly identify what is meant by a cycle. If there is local state in each process that contributes to this definition we must coordinate this information to determine if a cycle exists.

1.5 A hybrid approach

We have taken a hybrid approach that combines trace based debugging with static concurrency analysis. In doing so it gains advantages from both of these techniques. Our system, called SADCPU, implements this synergistic combination. SADCPU is an acronym for Static concurrency analysis Aiding Dynamic Concurrent Program Understanding.

The Task Interaction Concurrency Graph (TICG) is the central model used by our system. We present information to the analyst about both observed and potential executions by visualizing the corresponding portions of the TICG. We build this partial TICG using both static and dynamic techniques. The TICG is a directed graph where each node is referred to as a concurrency state, and each edge represents a concurrency interaction of the program under test. Each concurrency state consists of the current state of each task in the system.

When the tool is used to debug a faulty execution we begin by constructing a TICG that represents the failed execution. We construct this initial TICG from a trace that was collected at run-time. We collect the information about what tasking related events occur during a particular execution by instrumenting the source code of the system under test.

The engineer examines the initial TICG generated from the trace using our animation system. The animation system enables replay of portions of the execution at a much slower rate than the actual execution. This gives the programmer a graphical representation of the order states were visited. The engineer can then inspect individual states to discern the details of the tasking related information for each task, and the tasking transitions leading from that state.

Inspecting the initial TICG gives the programmer information about the particular execution that was traced, but doesn't provide insight into other executions that could have occurred given either different input data, or even with the same inputs but different non-deterministic choices made by the Ada run-time system scheduler. Gaining knowledge of related executions is quite valuable, because there can be more than one execution that leads to a particular failed final state, or there may be another failed state that is closely related to the failure state we observed. We allow the programmer to use static concurrency analysis to expand the graph in both the forward (successor) and backward (predecessor) directions.

Conventional static concurrency analysis often has problems scaling to larger problems due to its inherently exponential nature. We ameliorate this problem by limiting the portion of the graph as discussed above. Another way of attacking this is to limit the number of tasks analyzed to be a subset of the system under test. If some tasks are uninvolved in a particular failure, we can get accurate results with less work by ignoring their interactions.

Our primary focus is on deadlock errors, but we also have integrated information about data access into the TICG model. We record what state the system is in when a particular variable is accessed. In this way we can help the engineer diagnose data race errors.

Our main contributions are as follows:

- Provide understanding of concurrent executions through visualization of an execution model.
- Enable exploration of an execution by using incremental static concurrency analysis under user control.
- Enable debugging of data access anomalies by integrating data access information into the execution model.
- Extend the size of problems handled beyond those of conventional static concurrency analysis by providing incremental expansion and analysis of portions of the system.

1.6 Organization

This dissertation describes a methodology for understanding concurrent programs using a framework based on a concurrency state space model. We present a set of techniques for building, exploring, and annotating the model of a concurrent program, and for understanding how a particular execution relates to the overall structure and behavior of the program as a whole. Chapter 2 introduces the concept of execution state space model, and uses this to compare our system against some others which attempt to address similar problems. Chapter 3 describes our approach in detail, gives information about how and when particular features are likely to be useful, and presents the algorithms used to build and inspect the execution model. Chapter 4 gives some examples of how the techniques could be used to debug particular failures. Chapter 5 describes the implementation of our system. Chapter 6 concludes with a summary of what we have achieved, a discussion of the factors limiting applicability of the approach, and some thoughts on how the system could be expanded in the future.

Chapter 2

Related Work

Debugging tools do not automatically remove faults from programs, but rather aid the programmer in discovering and diagnosing faults so that they may be manually removed. Debuggers enable this discovery process by allowing programmers to examine the state of an executing program. To do this debuggers create execution models and use them to present information to the programmer. Debuggers for concurrent programs must model the temporal behavior of programs in addition to the data and control flow that is modeled by debuggers for sequential programs.

These temporal properties of concurrent programs make the models much more complex than those for sequential programs. The choices made in designing the model decide the expressiveness and efficiency of a concurrent debugger. In this chapter we examine the execution models of many existing debuggers for concurrent programs. In doing this we identify important criteria that describe the essential features of a debugger execution model. This helps us evaluate the execution model we have chosen for our system, and to understand how the functionality of other systems could potentially be integrated with our system. To enhance our understanding of how these other systems compare to ours, we define a reference model that is quite similar to our state space model, and relate the execution models used by these other systems to this.

2.1 What is an execution state-space model?

The purpose of a debugger is to provide a programmer with information that enables him to identify the location of program faults. To do this debuggers must model the state space of an executing program.

Models can be divided into two types: internal and external. Internal models are those built by the debugger for its own use. External models are those presented to the user. These may be separate, or the same model may serve both purposes. Often the external model is a higher level view extracted from the internal model. Our system blurs the line between internal and external models, because it takes what had been an internal model for SCA and presents it to the user as an external model.

For a sequential program the execution state can be defined by the current value of variables and the program counter. This is insufficient for concurrent programs because each thread contains its own program counter, and possibly a separate address space.

As a program executes the state changes many times. The linear record of all the instantaneous states a program went through is referred to as an **execution history.** For a concurrent program this includes state information for all the threads.

Useful programs tend to have many potential executions which differ based on input values. Non-deterministic programs can have different executions even when presented with the same input. We may wish to consider the set of all possible execution histories of a given program, considering all possible inputs, and all non-deterministic variants for each input. We call this set of potential execution histories the **execution state space** of the program.

The execution state space of even trivial programs is extremely large. For example a program that reads in two 32-bit integers and prints out their sum would have at least $2 * 2^{32}$, or over 8 billion potential execution histories, given that each input variable can take on 2^{32} different values. Concurrent programs have even larger execution state spaces, because not only do they have multiple program counters and address spaces, but the temporal ordering of inter-thread communication and synchronization must also be modeled.

Fortunately we rarely need to model the entire possible execution state space of a concurrent program. But the choice of what portion we do represent, and how we model it, greatly influences both what types of information we can provide, and how efficiently we can provide it.

2.2 Important characteristics of debugger state-space models

We would like to collect as much information about an execution as possible. In sequential programs the key attributes of a program state would be the value of the variables, and the program counter (P.C.) One obvious execution state-space model would be all combinations of P.C. and variable values that a program could take on. With this model we are guaranteed that we can provide complete information about any execution. Unfortunately, as described above, this approach is intractable for even small sequential programs. Concurrent programs add temporal properties further complicating and enlarging state-space models.

We may only wish to model the execution history that occurred during a particular failed execution. A simple method to collect the state information described in the previous paragraph would be to copy the values of all variables to disk every time the P.C. changed. This single execution history would be much smaller than the complete state space, but even this may be too large to be practical, especially for a long running program.

By recognizing that typically only a small portion of the state changes at each step in an execution history, we realize that a more efficient way to encode state histories is to record only the changes that occur between states. These records of what changed are called **state deltas.**

An even more efficient modeling technique that works for many concurrent programs is to record only high-level synchronization events, and later reconstruct the entire state. Note that this may be insufficient if the program is not race free. If races occur then knowing only the ordering of synchronization events won't let us locate the start of the error, since it occurred between synchronization states.

Even modeling all the events may be very costly, if we choose to monitor at a very low level. For example Pan and Linton [PL88] give the figure of 1 megabyte of data per second for a slow 1 MIP processor like the VAX 11/780. In contrast Bacon and Goldstein [BG91] reports 1.17 MB/second for a multiprocessor system with 16 processor of 12 MIPS each. If they had used the naive model of Pan and Linton they would have expected 192 MB/second, clearly an unreasonably number.

2.3 Criteria for comparing execution models

As we have seen, an execution model that included all possible information about all facets of a concurrent program execution would be immense, and impossible to collect. Because of this we are forced to limit our models to a subset of all the information. What portion of the overall complete state space we choose to model controls what types of information we can provide to the user. How we represent this information has a great impact on the efficiency and overall usability of debugging tools. To aid in deciding what information to include, and even more importantly what we must leave out, here are some criteria for evaluating execution models.

- Number of execution histories modeled.

  A fundamental consideration is whether a model is designed to represent only a single path through the execution state space, or can multiple histories, up to the entire execution state space, be modeled. As an example of this consider two alternatives for debugging an Ada program that has exhibited a deadlock. One possibility is to do a complete static concurrency analysis, which will model all possible executions of the program, and point out any task interactions sequences that lead to deadlock. With this method we could examine the output and find the sequences that end in the deadlocked state we observe. Another way to go about debugging this program is for the Ada runtime system to trace the task interactions of the running program, and print out this information when the system deadlocks. The static analysis scenario gives us information about many possible executions, while task interaction tracing can give us more detail about a particular execution.

SADCPU blends the two, building a simple model of a single traced execution, then expanding it statically to give us information about other, related executions.

- Level of abstraction.

  All these models are abstractions of real executions, and leave out some details while emphasizing others. While concurrent programming is a conceptual model, any particular program we need to debug is written in a specific programming language, and runs on a particular hardware platform.

  In deciding on a level of abstraction for an execution model we must decide whether the model should be closely tied to a particular language and/or hardware platform.

  If we tie the model closely to a particular language we can gain benefits in several areas:

- It will be easy to map particular states back to the program source code. This is the level the programmer is used to dealing with.
- We gain efficiency by having good encodings for the constructs of this language.

A problem we encounter is that the model may not be sufficient if we need to debug a program consisting of threads written in different languages.

An even lower level of abstraction ties the execution model to a particular hardware platform. We may use our knowledge of the number of processors, and the mapping of threads to them, to limit the amount of information we need to collect. This often results in a more efficient implementation. When debugging performance problems in multiprocessor applications it is often impossible to find the source of bottlenecks without considering the hardware, especially the number of processors and the overhead of task switching.

A problem with the machine oriented approach is that we may miss underlying problems with the synchronization structure because they cannot appear when run on a machine with this particular configuration. If we later run the program on a machine with a different configuration it may encounter new failures that we didn't diagnose previously because our model couldn't consider them.

A higher level, language independent model maps events to those in the design formalism, which a language based model maps to constructs in the program source code. A machine level model captures executions by what machine states were visited. As an example of this consider a system consisting of multiple threads running on a multi-processor machine, where the runtime system dynamically switches threads between processors. Using a language model we would record events associated with a particular thread, whereas a machine oriented model would record all events that occurred on a particular processor.

For SADCPU we have chosen the intermediate, language based approach, but using a formal model that is conducive to incorporating other languages whose concurrency primitives are similar. Our model relies on the rendezvous model of synchronization, which works well for our primary focus on Ada, but would also be easily adapted to CSP, Occam, or Concurrent C++, all of which use this same concurrency model.

- Selective focus.

  As discussed above state-space models, and even models of a single execution history can be very large. The ability to model a subset of a program's threads can help limit the size of the model. There are two motivations for the user to want to narrow the model to cover only certain suspect threads:

  - Generating the partial model takes fewer resources, making modeling tractable.
  - Removing extraneous information about threads not involved in the current failure makes it easier to concentrate on the relevant information.

  SADCPU includes a feature called "task subsetting" that implements this type of selective focus.

- Scheduler non-determinism.

  Programmers of concurrent systems frequently encounter synchronization errors. Non-determinism allows the runtime scheduler to chose from many possible interleavings allowed by the program's synchronization structure. Often the errors are the result of the programmer overlooking one of these event interleavings. Therefore it is important that the debugging system model the actions of the runtime scheduler to make the cause of the error apparent.

  SADCPU models scheduler non-determinism using an interleaving execution model. The non-determinism is represented by multiple-successor states.

- Modeling of external events.

  Concurrent programs often depend on external events. This could be as simple as the series of values in an input file, or it could be interrupts from an external device in an embedded system. If these external events can change an execution it may be necessary to model them for the debugger to be useful.

  In SADCPU the effects of external events are visible in the partial trace captured, but currently there is no explicit modeling of external events. This could be added when a full debugger is integrated.

- Variability of level of detail.

  There is a usually a tradeoff between efficiency and level of detail. If the user can control this tradeoff it may result in a debugger that is capable of handling complex program interactions, but still providing the fine level of detail necessary to easily identify bugs. For example one strategy might be to start with a high level model, and use this to identify the portion of the program that is most likely to contain the problem. Then a more detailed model could be generated for just that portion.

SADCPU doesn't have multiple levels of detail, however it is capable of focusing on particular portions of the state space and ignoring others. In this way the engineer can use a very fine detail while keeping the analysis tractable.

- Uncertainty in run-time data.

  In systems that record execution histories we may have imperfect or incomplete information about the execution. For example we may know that two events occurred in close temporal proximity, but not the exact ordering. In this case the programmer needs to know what parts of the model may be inaccurate, and a method of dealing with this needs to be provided. Building submodels for each possible ordering is one way of doing this.

  SADCPU has uncertainty in its model for data accesses, and this is made clear to the user. We can also handle uncertainty in the form of incomplete trace data for tasking events by determining which series of events could have occurred and which are clearly infeasible.

- Representation of temporal properties.

  Because synchronization faults comprise many of the problems in concurrent programs, and these are manifested in the temporal ordering properties, how we model temporal properties is very important. A well chosen model of these properties will help the programmer easily locate the fault, while an accurate but poorly chosen model will hamper user understanding.

As an example, two vastly differing approaches for representing time in concurrent programs are animation versus explicit time representation. Animation presents states one after another, with the programmer remembering what changed. On the other hand there are many other systems which represent multiple states within a graph structure, with time explicitly encoded.

SADCPU combines both of these models. It uses an interleaving model which represents time as paths through a graph, but we also provide an animation feature which highlights the path followed by a particular execution.

- Support for manual inspection versus model compactness.

  When choosing a model, there is often a conflict between readability versus compactness for machine interpretation. Humans typically use redundant information when trying to understand what they are looking at while automated tools are often much more efficient and the algorithms easier to design when more compact data structures are used. If we intend to use both human inspection of the generated model, and automated tools that derive information from the model we must decide which to optimize.

  In the SADCPU system we opted for a compact representation that requires careful human inspection. The individual states are more complex than some other models, but the reduction in number of states generated, and which therefore need to be inspected, seems to help offset the additional cognitive workload for each state.

2.4 A framework for comparing concurrent debugger execution models

The models we are comparing come from concurrent debugging tools that differ greatly in both the types of programs they attempt to debug, and in the style of debugging they provide. Because of this the models also differ significantly. To aid in the comparison of these dissimilar models we will define a single state space model to use as a standard or reference model and use it as the basis for the discussion. The extensions that are necessary to the reference model to incorporate the information found within a specific debugger execution model represent the additional expressiveness found in that model.

Our reference framework is an adaptation of the Concurrency History Graph (CHG) described in [Tay83]. The CHG itself is a model of the potential execution histories of a concurrent program, but with all program details except the concurrency related interactions abstracted away. We can use this model to compare debugging techniques if we extend it to represent other information we wish to consider beyond the concurrency states. By determining what portion of this annotated CHG structure each technique makes available to us, and with what costs and restrictions, we have a way to compare the merits and shortcomings of each system profiled.

This CHG model is quite similar to the TICG model used by SADCPU. It is less compact, but more readable. As will be discussed later we initially investigated this model for use in SADCPU, rejecting it because its semantics are less well defined. The ease of understanding makes it valuable for use as the reference model, and the imcomplete semantics only become problematic when very fine detail is considered, which is unnecessary for the comparison given in this section.

2.4.1 The Concurrency History Graph Model

In this section we give an brief and informal review of the Concurrency History graph model described by Taylor in [Tay83].

Taylor's approach begins by representing tasking Ada programs as sets of rooted directed flowgraphs. These annotated flowgraphs are similar to those used in compilers for dataflow analysis, but they are reduced by removing information not related to tasking activities.

Each **state node** of a flowgraph corresponds to a tasking related statement (entry call, accept, select, select-else, delay, abort, task begin, task end, and subprogram begin, subprogram end, subprogram call, block begin and block end) but only where the subprogram or block may directly or indirectly perform a synchronization activity. The nodes in these flowgraphs are connected by edges representing control flow.

A **concurrency state** is a ordered n-tuple, where each element is a state node from one of the n flowgraphs representing this program.

A **successor** function is defined on each **concurrency state** which generates all possible concurrency states that could follow this one, given the semantics of the Ada tasking model.

A **concurrency history** is a sequence of of concurrency states starting with the initial start state, and where each state in the sequence is a **successor** state according to the function given above.

A **proper concurrency history** is a concurrency history with no loops.

A **complete concurrency history** is the set of all possible proper concurrency histories for a given program.

A **concurrency history graph (CHG)** is a graphical representation of the complete concurrency history for a program. Using this representation eliminates the need for full storage of identical states which appear in more than one history. Taylor's algorithm builds this data structure and uses it in analyzing the synchronization properties of Ada programs.

We will use the **concurrency history graph (CHG)** as the basis of a model for representing the execution state space of a concurrent program. We will extend this state-space model to represent different types of information that might be collected or generated during the debugging process, and use this resulting model to compare the execution models of existing debuggers.

2.5 Comparing execution models

In this section we present a comparison of the execution models of existing debugging tools. Our comparison will be two-fold; first we will explain the type of information provided by the tool and the execution model used by the technique to derive it, and then we will discuss the extensions to our CHG-based model that would be necessary to represent this type of information. We also evaluate the execution model the criteria discussed in section 2.3.

To organize the comparison we divide the debugging techniques we are considering into categories based on the type of information they are trying to present to the programmer. This is the most appropriate criteria for partitioning the systems because the type of information to be displayed controls what information we need to model.

Other choices we could have made for partitioning the systems include:

- **When model is built** When is the technique active, and when is the model built? In this case we would partition the systems into compile time static analysis, run-time monitoring, and post-mortem examination. Run-time monitoring presents particularly difficult constraints on how much resources we can consume building the model. We chose not to use this as the main criteria because several of the techniques build their models in multiple stages, spanning these divisions.
- **Type of programming model** There are often differences in models based on the type of programming model it supports (message passing, shared memory, massively parallel versus smaller numbers of processes.)
- **Intended usage** Depending on the type of fault we are trying to locate, different information may be needed for diagnosis.

Our partitioning of debugging systems by type of information represented yields the following divisions:

- **Deterministic Re-execution**

  Tools within this category allow the user to recreate a failed execution, with the assurance that non-deterministic choices will be resolved the same way as the original execution.

- **Predicate based debugging**

  These systems build a model of expected program behavior by defining predicates describing intended and unintended behaviors of the system under test. Actual program executions are compared to the model and information about interesting occurrences is provided to the user.

- **Causality/Partial Order presentation tools**

  Partial orders provide a convenient way of characterizing the causality of program events. In a single execution any two events, a and b will have a particular observed total ordering, either a precedes b or b precedes a. It may be the case that either of these ordering is possible in different executions. Partial orders capture the causality relationship that exists if the program synchronization structure constrains the ordering of two events. For example if a is a message send and b is the receipt of that message, then a must always occur before b. This relationship is captured by Lamport's [Lam78] "happened before" relationship, denoted $a \rightarrow b$. Partial orders are a powerful abstraction for understanding the concurrency structure of a program, and many tools exist to provide this information.

- **Data based debugging**

  In many failed programs the fault first manifests itself as an error in a data value. This leads naturally to debugging by examining the values of variables in an execution, and the program events and states that affected the current value of that variable.

- **Specialized portrayals**

  This category encompasses several specialized portrayals of the concurrency attributes of a program execution. These are often valuable as an adjunct to the techniques above, providing a different view of the same basic information, but specialized for one execution environment or type of bug.

2.5.1 Deterministic Re-execution

The systems in this category collect information at run-time to enable the user to recreate a failed execution post-mortem. The models describe a single execution in sufficient detail to ensure that non-deterministic choices are made the same way, and that the replayed execution is a faithful model of the failed execution.

Instant Replay style debuggers

Instant Replay [LMC87] style debuggers save the relative order of significant events during execution, but not the data. By constraining the re-execution such that the synchronization events occur in the same order as during the original execution, it is guaranteed that the data values are the same, provided the external environment is equivalent. This means we get equivalent replays without having to save the values of the variables at each point.

LeBlanc's original Instant Replay models an execution as a series of reads and writes to shared memory variables. The shared objects are locked by semaphores, and each has an associated version number. Each thread has a "process history tape" that records the version number of the object before the read. When a shared object is written the version number associated with an object is incremented.

During replay, the runtime system holds up a process attempting to do a write until all processes that saw the current version of the shared object have completed their reads. An important requirement on this system is that the individual accesses to object must be race free. This means that races occur when coarse grain objects are accessed in different orders, but that the individual shared memory accesses that make up a single coarse grain access are always correctly locked.

Netzer [Net93] describes an enhancement to LeBlanc's Instant Replay that allows it to work on programs where with finer grain shared accesses. This work eliminates LeBlanc's requirement of race free coarse grain operations, and also results in a 2-4 order of magnitude improvement in reducing the number of events that must be traced. Netzer's model views program execution as a triple $P =< E, T, D >$ where E is a finite set of events, T is a temporal ordering relation that show the relative order in which events execute, and D is a shared-data dependence relation that shows when one event can causally affect another.

To maintain this model the optimal tracing algorithm maintains several data structures containing information about an execution.

An *access history* for each shared variable S is a list of events, one per process, that most recently accessed S.

Each event is identified by $< process\#, serial\# >$ pairs, and each process maintains a local counter with which it assigns serial numbers to its events.

Each access history consists of the writer, showing the last event which wrote S, and the *readSet*, showing the last event in each process that read S.

Each process also maintains a vector timestamp of event serial numbers, similar to that of Fidge [Fid91].

These data structures are used by a race detection algorithm to determine which accesses can possibly occur in a non-deterministic order. These are exactly the accesses that need to be traced, since all other accesses will occur in the same order during replay without any special intervention required.

Choi and Stone [CS91] describe an incremental tracing strategy, which collects coarse traces, called logs, during execution and generates detailed trace at replay time using the coarse traces generated at runtime. This allows a trade-off between overhead spent collecting traces during runtime, and the resources required post-mortem to generate detailed information.

Their execution model is based on the *prelog* and *postlog* information collected by Miller[CMN91], which is discussed below. Prelogs collect information at the start of a region about the values of variables that might be read accessed in that interval. Postlogs record what variables changed within that region.

During replay the system regenerates needed MOD and USE information when it is needed by the user. The longer the interval between prelog and postlog, the longer replay will take.

Bacon and Goldstein [BG91] describe a replay system that implements tracing by using the cache-coherency hardware found in shared-memory multiprocessor systems. Because they assume a RISC architecture they are guaranteed that each data reference corresponds to exactly one instruction, and each instruction corresponds to at most one data reference. Their execution model consists of those memory writes that cause interprocessor dependencies. These are exactly those accesses that can cause non-determinism. They can capture these writes because the cache-coherency hardware will detect that a shared cache line is being written to, and will broadcast an INVALIDATE transaction on the bus. When this occurs the tracing system logs the CPU number and the instruction counter. The resulting log has a sequential record of all significant bus transactions. During replay we can use this log to create a sequential replay that will accurately recreate the logged execution. Because this must be run sequentially it will run much slower ($O(N)$, where N is the number of processors the logged execution was run on.) They claim this is often acceptable if we use checkpointing. They also present another scheme using partial orders that allows parallel replay, but requires additional hardware.

Elshoff [Els88] has implemented a debugger for the Amoeba operating system. The debugger has an event-based execution model. Most of the events relate to the message passing that is the major form of interprocess communication in Amoeba. However, Amoeba also has an asynchronous signaling primitive that makes it impossible to use Instant Replay style tracing to implement a comprehensive replay facility. When signaling is used the user may not be able to replay the exact execution. Instead the debugger informs the user of this, and instead provides manual control of the scheduler, so the user can try different scenarios to see which one is most likely.

The ability to control the scheduler is also used if checkpoints are taken. The Amoeba debugger can restore state to a checkpoint, then allow the user to explore a different execution sequence from the one that was observed.

Integrating this information into reference model.

To enable this kind of replay of shared memory programs our CHG-based reference model would need to have each concurrency state annotated with data reference events. A trace would then be a single path through the CHG. Amoeba style replay with uncertainty would be modeled by indicating by branching in the CHG, which would indicate that more than one successor state is possible.

Checkpointing and Rollback

Another strategy for enabling replay is checkpoint/rollback. Instead of recording a continuous stream of information about events that have occurred in an execution, checkpoint/rollback systems take frequent snapshots of an execution.

Pan and Linton describe the RECAP [PL88] system which implements reverse execution using a checkpoint/rollback approach. Reverse execution allows the programmer to "scroll" forward and backward through an execution history. At each point the programmer can examine the state. Pan and Linton make the point that pure replay systems are impractical for very long running programs. If a failed execution has run for several days, even if the replay system adds no overhead the user won't want to wait for the re-execution to reach the state where the error occurred. Their system logs reads of shared memory, including the value read, introducing a large overhead. This also produces a very large amount of log data, and only the most recent checkpoints are kept. This limits reverse execution to recent events.

Feldman has implemented the IGOR [FB88] system for checkpointing of sequential programs, and argues that it can also be used for parallel ones. This system relies on the virtual memory system of a modified UNIX system. This kernel supplies a function that can be called periodically to write "dirty" memory pages to disk. IGOR also includes mechanisms for restoring the external state that existed in the original execution.

The Distributed State Restore (DSR) [GGLS91] mechanism uses checkpointing, with integrated replay tracing and causality tracking, to allow the user to restore a program to a consistent state. In fact, this state need not be exactly one that was encountered during the execution recorded, the only requirement is that it be consistent with the partial order observed during the execution. The progress when execution is continued from the restored state may be controlled by the user such that it is no longer consistent with the original execution, if the programmer wants to investigate alternate executions.

DSR models executions as sequences of global states. Each global state consists a local state interval for each process. A state interval is the series of local events that occurs between non-deterministic events. A consistent global state is one in which the state intervals for each process could have occurred concurrently given the recorded partial order. The partial orders are recorded using Fidge vectors clocks [Fid91].

The user may also choose to focus on only a subset of the processes and the debugger will automatically deal with the missing information.

Integrating this information into reference model.

Our reference model would need to have the checkpoint data as an annotation to the concurrency state. The synchronization events that cause a new concurrency state to be generated seem especially appropriate points to do checkpointing. We might also want to include information about the external environment that existed during the original execution.

2.5.2 Predicate Based Debugging

Predicate based debugging system rely on having the user build a model that describe expected program behaviors. The actual executions are then checked against this model and interesting events are reported to the user.

Bates [Bat88] describes the Event Based Behavioral Abstraction (EBBA) system. EBBA is a framework that enables the user to build top-down models that describe user knowledge of intended system behavior. These models are then compared to actual system behavior and information about differences is made available to the user. EBBA starts with low-level program events, but allows the user to combine this using clustering and filtering to describe higher level events reflecting user knowledge about the programs intended structure. Because these higher level events can be design level concepts, EBBA can support debugging on heterogeneous programs.

The EBBA execution model is the event stream, which is built up into higher level constructs using the user provided models.

Cooper and Marzullo [CM91] describe an approach for detecting global predicates in a distributed system. They are concerned with finding out whether a particular predicate relating state in different processes ever becomes true, for example if each process has an integer-valued variable x, we may want to know if the median value of x is greater than 10. They define three interpretations of this predicate and algorithms for detecting them.

Possibly means that there exists an execution consistent with the observed behavior such that the predicate was true at some point in the execution.

Definitely means that for all executions consistent with the observed behavior the predicate was true at some point in the execution.

Currently indicates that the predicate holds in the current state.

Their approach is intended to debug distributed message passing systems. One consequence of the distributed nature is that at any instant some messages are in transit, and this makes it difficult to make queries about global predicates. In such a system it is simple to recover the local state information that yields a partial order, while collecting a total order will involve stopping some processes to collect information at runtime. Stopping some processes is necessary to derive the *definitely* relationship.

Each partial order encompasses many possible total orders. This means that when they monitor they cannot be certain exactly which of the potential total orders was generated. It may be that some total orders satisfy the predicate while others don't. If this is true then the *possibly* predicate holds. If all potential total orders consistent with the given partial order result in the the predicate being true, then the *definitely* relation holds.

To represent the multiple total orders associated with a detected partial order they use a lattice of global states as their execution model. The lattice represents all possible computations consistent with the observed partial order, and any path through the lattice is a potential total order.

Integrating this information into reference model.

To support this style of debugging we need to include information about when predicates became true. This may not map cleanly to particular CHG concurrency state because the predicate may become true at times other than when synchronization occurs.

2.5.3 Causality/Partial Order presentation tools

As we have seen in the systems presented to this point the partial order relationship that defines causality is a very important piece of information about a concurrent program execution. Many systems utilize partial order information to derive other views of an execution, but the partial order information alone can be a valuable aid for a programmer debugging synchronization related errors. The tools in this section generate and display information about partial orders to the user.

Kimelman and Zernik [KZ93] present an optimal technique for displaying causality in message passing systems by using an on-the-fly topological sort. Their execution model is a graph of events consisting of the message sends and receives with local timestamps . The difficulty in determining causality arises because the distributed nature of the executions they monitor doesn't ensure that event records from different processors will arrive at the central monitoring point in the same order they actually occurred. To maintain their graph they rely on being able to distinguish from the contents of all event records collected up to this point how many predecessors and successors are expected. A predecessor is an event which must precede another event, for example the send of a message must precede the receipt of that message.

Stone [Sto88] has created a graphical representation of partial orders called a concurrency map. Stone's execution model consists of the local histories of each process in a concurrent execution, plus the time dependencies that describe our knowledge of which events must have preceded others. Obviously all events within a single process have a known total order, but we may not have knowledge of the relative ordering of events in different processes. To represent the known total ordering within a process the concurrency map displays a column listing the local history of that process. This column is divided into blocks begun and ended by synchronization events. The concurrency map has arrows between blocks representing known time-ordered dependencies between blocks. If there is not a transitive path between one block and another, then they are potentially concurrent.

Stone describes how the concurrency map could be used in a speculative replay system. If we are guaranteed accurate replay to a certain point in the program the debugger could stop where potential inaccuracy exists due to incomplete information, and display the concurrency map to the user. Since the concurrency map displays all potential concurrency ordering consistent with the information available, the user could utilize this information to decide what path to follow.

Integrating this information into reference model.

Information about what portions of code may have executed concurrently could be integrated with the CHG-based model by adding a table that shows for all the concurrency states generated which pairs of state nodes occurred together.

2.5.4 Data based debugging

The systems discussed so far largely concentrated on the events taking place in a concurrent execution rather than the data. Because the errors detected during a failed execution are often in the data values, the debugging tools in this section provide an alternate view that may often be useful.

Choi, Miller and Netzer [CMN91] describe a method for doing flowback analysis in concurrent programs. Flowback analysis allows the user to determine what program events affected the current value of a variable. Their model use prelogs at the start of a block of code to describe which variables may be written or read, and postlogs to list the values of variables that were written during the block. They use these to generate a dynamic program dependence graph which describes what events affected the current value of a program variable. Actually, when the user begins debugging only the portion of this graph representing the most recent events is available, and the rest is built incrementally upon user request.

Dinning and Schonberg [DS91] present another data oriented approach designed to deal explicitly with race condition errors. Their method is an efficient on-the-fly technique for detecting when race conditions occur in programs with critical sections.

Their method uses a data structure called the *Partial Order Execution Graph* (POEG) that captures the partial ordering relationship imposed on events by the concurrency structure of the program. Their POEG includes lock covers to reduce the number of infeasible races that are reported, by incorporating the knowledge that any two accesses that have the same lock in common cannot have a race between them. This annotated POEG is their execution model with portions of it created as need to check data accesses for races at runtime.

Provided each thread executes deterministically (called internal determinism) their race detecting algorithm has a useful property called *single input, single execution* (SISE). If this property holds, then if the program is run with a particular input and no races are detected, then there will be no races in any execution with that input. There may still be non-determinism between processes, but it can never result in a data race. Dinning and Schonberg describe a static analysis algorithm for detecting when this is true.

Hseush and Kaiser [HK88] describe an approach based on the history of a given variable, rather than a given process. Their execution model consists of the histories of all variables in the execution. Their debugger uses this model to map the observed data path expressions (DPEs) back to the control flow that caused them. The user describes interesting DPEs, and the debugger takes actions (e.g. invoking a breakpoint, forcing execution along a particular path or outputting information about the program state) when they are observed.

Integrating this information into reference model.

Integrating information from data path debugging could be done by creating a map for each variable showing which concurrency states contained accesses to that variable.

2.5.5 Specialized portrayals of concurrency structure.

This section discusses several interesting depictions of information about the structure and behavior of concurrent programs that are different from the more common groupings listed above.

Pancake [Pan93a] describes a user-defined, graphical representation of parallel program structure called the *program phase tree* (PPT). The PPT is derived from the call tree established by the source program. The PPT is motivated by observations of programmers attempting to debug parallel programs. They often hand-generated sketches of the dynamic call graph of the program as they were attempting to understand the execution. Pancake's debugger use the PPT as its model of execution, and automatically derives this information and presents it to the user. The user can customize the portrayal by choosing to eliminate some nodes from the display, or grouping nodes into composite nodes.

Fowler, LeBlanc and Mellor-Crummey [FLMC88] describe a visualization referred to as a *time-space diagram.* The time-space diagram is a two-dimensional graph. One axis represents the temporal dimension, with long intervals of time appearing as long arcs or nodes. The other axis is used for the spatial dimension, which is used to distinguish between processes or objects depending on the focus of interest.

To build this representation they collect information at runtime. The execution model is a directed acyclic graph, where nodes correspond to monitored events, and arcs indicate temporal relationships.

Hough and Cuny [HC88] describe Belvedere, an animation based debugger. Belvedere uses an execution model based on the sequences of GET and PUT message operations and CREATE and DELETE operations on processes and channels. It stores this information in a relational database. Standard database queries are then used to describe "interesting" patterns of communication which are retrieved and animated.

Integrating this information into reference model.

The call graph information needed by PPT could be annotated to each concurrency state. Generation of time-space diagrams and animation could be done on the CHG model, the richness is in the style of presentation, not the underlying data.

Chapter 3

A Hybrid Approach

We wish to provide the user with an organized view of program behavior like that provided by SCA, while gaining the ability of dynamic tracing to deal with large, complex programs. We do this by providing a formal model of the state space of a concurrent program derived from that used by SCA. We avoid the tractability problems that arise from the huge number of states generated by SCA because instead of performing an exhaustive analysis of the complete state space, we generate only a portion of the state space under user control.

The basis of our approach is to develop a partial model of the state space of a concurrent program, allowing the programmer to inspect the model to gain insight into the program. The programmer may then incrementally expand the portion of the model that seems interesting, and avoid generating uninteresting parts.

Our formal model is based on the TICG proposed for SCA by Long and Clarke [LC89]. We chose this model both because it has a well understood semantics [PTY95] , and because of considerable familiarity with it from work with the CATS [YTL^{+}95] system. We also have a set of tools for generating the underlying model from source code which we were able to reuse.

Initially we had investigated using the Concurrency History Graph (CHG) model proposed in [Tay83], which was discussed in Chapter 2. The CHG contains equivalent information to the TICG, but in a less compact representation. The CHG model uses reduced flow graphs (RFGs) to represent individual tasks, and these are based on control flow. The TICG uses TIGs which are based strictly on task interactions. The result of this is that a single TICG state can encapsulate the information present in several separate CHG states. This encoding reduces the number of states required to represent a given program state space, without loss of analysis accuracy. The tradeoff in our work is that the individual TICG states are more difficult for the user to map back to the original program state. We had originally chosen the CHG as our model to increase understandability, but later switched to the TICG because the formal semantics of the TICG model are more clearly defined. We could still replace the TICG with the CHG model, but to do so we would have to define a complete set of semantics for the CHG.

The TICG is a model of the concurrency structure of a program. Each thread of control (in our case an Ada task) is represented by a Task Interaction Graph (TIG.) A TIG abstracts all away information not related to tasking activity. Each node in a TIG represents a region of a task, and each edge represents a task interaction indicating a transition from one region to another. The set of edges leading from a node indicate what tasking actions are possible from that node. A TICG is a model of the behavior of an entire concurrent program, and is constructed from the TIGs of the tasks that make up the program. Here each node is referred to as a state, and is a tuple made up of one node from each TIG. The edges of a TICG represent tasking actions of the program as a whole (beginning or ending of a rendezvous in the case of Ada.) Each edge involves as few tasks as possible, no edge represents two independent events occurring simultaneously. One way to view a state of a TICG is as a description of the next possible concurrency activity of each task.

We utilize a modification of the basic TIG model that relaxes the requirement that each TIG node begin with the same synchronization action. This modification, described in [YTL$^+$95] reduces the size of the TIGs, and therefore the TICG, by eliminating duplication of nodes which contain the same code and exit edges. The algorithm for detecting and removing these duplicate nodes is incorporated into the procedure for converting TIGs from the form produced by the Language Processing Toolset front end into a more compact and efficient internal representation. The code for this procedure is shown in Section A.2.

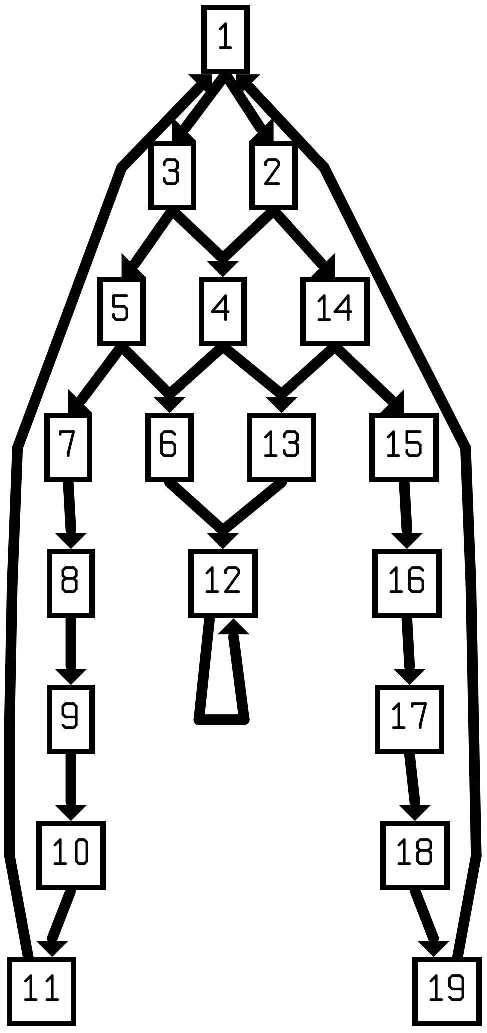
Although both the TIG and TICG models are directed graphs, they represent different types of entities. A TIG is a representation of the static concurrency flow of control of a single task. Understanding a TIG means you grasp the flow of control within a single task, which is relatively easy to discern. A TICG on the other hand represents the dynamic behavior of an entire program, and comprehending it means you are aware of the concurrency actions that take place in any potential execution of the program. This understanding is much more difficult to achieve, but also much more powerful in aiding the program understanding and debugging process.

An illustration of the difference between these can be found in Figure 3.1 This program shows two variants of the familiar dining philosophers example. The version on the left can deadlock if Phil1 picks up Fork1 and Phil2 picks up Fork2. The version is the same, except that the order of picking up the forks.

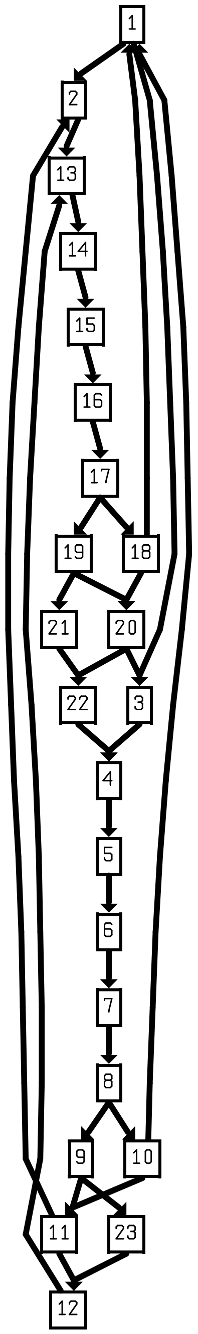
```
procedure two_phil is
   task body Fork1 is
   begin
      loop
         accept UP;
         accept DOWN;
      end loop;
   end Fork1;

   task body Fork2 is
   begin
      loop
         accept UP;
         accept DOWN;
      end loop;
   end Fork2;

   task body Phil1 is
   begin
      loop
         Fork1.UP;
         Fork2.UP;
         Fork1.DOWN;
         Fork2.DOWN;
      end loop;
   end Phil1;

   task body Phil2 is
   begin
      loop
         Fork2.UP;
         Fork1.UP;
         Fork2.DOWN;
         Fork1.DOWN;
      end loop;
   end Phil2;

begin
   null;
end two_phil;
```

```
   task body Phil2 is
   begin
      loop
         Fork1.UP; -- picks up
         Fork2.UP; -- fork 1 first
         Fork1.DOWN;
         Fork2.DOWN;
      end loop;
   end Phil2;
```

Figure 3.1: Two versions of Dining Philosophers. The order of picking up forks is different in Philosopher 2. The version on the left deadlocks, the version on the right doesn't.



Deadlocking version



Non-deadlocking version

Figure 3.2: TICGs built for good and bad versions of two dining philosophers.

The TIGs for the two versions are exactly the same except for Phil2, which has the two forks picked up in a different order. The TICG built for the second, deadlock free, version is much different, as seen in Figure 3.2. This is an example of the power of the TICG as a visualization, because it gives a view into the overall structure of a program, where most debuggers provide only information about events occurring within a task.

Gaining this understanding is much more difficult, but provides us with a powerful model to understand the program. Thus the TICG provides a model that relates the dynamic behavior of a program to the concurrency structure that shapes it. We argue that this is the missing formal model the programmer needs to understand and debug concurrent programs, and that a graphical visualization of this effectively communicates much information about the program under test.

3.1 Exploring the state space.

The concurrency related failures we address are deadlocks and data anomalies. In a deadlock all threads are blocked such that they are unable to progress, and the program is not in an acceptable final state. Freedom from deadlock is usually taken as an implicit specification on all concurrent programs. Concurrency related data access anomalies such as race conditions do not result in structural problems as deadlock does, instead they manifest themselves as incorrect program results. Whether a given result is incorrect depends on the program specification, and we rely on the programmer for this information.

If a program contains faults that can lead to either of these types of failures then the failed states will be represented in the TICG. In the TICG we can also identify the the states which must precede the failed state, and any path from the initial start state to the failed state represents a potential execution which can fail. Thus inspecting the TICG gives us information about executions that can lead to failures.

As described above, a TICG state is made up of one node from each TIG. Each TIG node represents a piece of code from that task. The edges out of a TICG state represent all concurrency interactions possible from that state. The causes of concurrency failures can be divided into two classes; existing erroneous interactions, and missing but necessary interactions. The interactions are represented by the edges of the TICG, but they are controlled by the code for the task, which is represented by the TIG nodes comprising that state. Thus, when using our system to understand and debug an error it is necessary to inspect both states and edges of the generated TICG. The failure occurs in a particular state, which is reached by one or more edges. The code which is faulty, and which requires correction, is represented by a state whose edges ultimately lead to the failed state.

Conventional reachability based SCA generates all states which can be reached from the start state, then examines each generated state to see whether it is a failure state. The usual method of generating a TICG for SCA is to create an initial state representing the state of the program during elaboration, then generating successor states by trying to advance each task that is attempting to make an entry call by checking if the called task is in a region where it can accept the call. If so, a new state is generated representing the state of the program after the rendezvous has occurred, and this successor generation algorithm is repeated on each new state that is generated. Normally the successor generation is continued until there no new states can be generated. This results in a state for every possible concurrency state the program could visit.

Because the full TICG represents all possible executions, it is possible to detect all potential failures. Unfortunately, as we noted previously, building the entire TICG is often infeasible. In place of generating the full TICG representing the complete state space of a concurrent program, we suggest incrementally generating portions of the state space under user control. We will now demonstrate how these additional states can be generated and how this can assist in debugging.

3.2 Building a TICG to represent a traced execution

As mentioned earlier the problem of reproducing infrequently occurring problems in concurrent systems led to the development of trace based debugging. Traces describing important concurrency behavior are collected during each execution. The trace itself is a sequential list of the concurrency interactions that took place as the program executed. When an execution fails the trace has captured enough information to allow a post-mortem investigation of the program states visited. A simple examination of the list of events which occurred may give the programmer sufficient information to discern what went wrong, and how to correct this. This type of information is presented by many prior debugging research prototypes, and in some advanced production debuggers.

Our use of the trace is somewhat different. When an execution fails we use the trace information to construct a partial TICG which represents the failed execution we observed. We present the user with a graphical representation of the states observed during execution. If each state is visited only once then the TICG will be a linear list, but more commonly it is a graph containing cycles if the program visited the same concurrency state more than once. This is a significant advance over tools which can only give a sequential list of task interactions that occurred, because it can give insight into the dynamic structure of program behavior with respect to repeated processing steps. While the dynamic behavior is controlled by the static structure of the individual tasks, the loops in this TICG can represent something more fundamental about an execution. One preliminary way the user may profitably use this initial TICG is to attempt to match up the observed behavior in terms of loops to that which was intended. The causes of failure may be discernible where the system has paths that don't fall into the expected and intended processing paths.

To aid this process of discovery we have implemented a simple trace animation facility. After the initial TICG has been displayed the user can view an animation that sequentially highlights the nodes visited in an execution. The rate at which the animation proceeds is user selectable.

Collecting the trace is what allows us to do this post-mortem inspection. In our prototype we collect information by modifying the source code with additional statements to record concurrency state changes, which are logged to a trace file. Our instrumentation tool is a modification of the ARTEMIS tool written by Owen O'Malley at the University of California, Irvine. The process of collecting the trace can perturb the program execution, resulting in the well known "Probe Effect" [Gai86] where failures disappear when the program is run with tracing enabled. Minimizing the instrusiveness of program tracing has been the subject of much research in recent years [Bru91, Gor91]. Collecting traces by instrumenting source code has been sufficient for our initial prototype, because our primary concentration is interpreting the collected data and using it to construct the desired abstraction. Adopting a more efficient trace collection method will be important to minimize both performance degradation and "Probe Effects."

One fundamental concept in our tracing work is that each running task has two identifiers, a static task ID (STID) and a dynamic task ID (DTID). The STID identifies which TIG describes the concurrency structure of this task. The DTID is used to distinguish instantiations of a task type. In the case of a single task the STID uniquely identifies it, but in the case of a task type with multiple instantiations we must use the DTID.The DTID's are collected from the Ada runtime system.

Our instrumentation tool modifies the parameter list of each accept statement to add a parameter for the DTID of the caller. Each entry call is similarly modified to pass in its DTID. In addition another parameter is inserted that identifies the particular statement that is making the entry call. This is necessary because the same entry can be called by many different statements within a single task.

Within the body of each accept statement we insert code which records the DTID and statement ID for both the calling and accepting task. In this way we can map each traced rendezvous to the corresponding TICG transition edge. When a new task instantiation is started the STID describing this task is recorded so that this information can be used later.

The SELECT statement is the source of most non-determinism in Ada programs. Recording which accept alternatives are taken in a selective wait gives us most of the information we need, but in addition to tracking the normal rendezvous we must record when several other tasking interactions of interest occur, including when:

- DELAY alternatives are taken in selective waits or timed entry calls
- ELSE parts are taken in selective waits or conditional entry calls
- TERMINATE alternatives are taken in selective waits

We add code to record these tasking actions, and model these as a type of pseudo rendezvous (this is an extension to the basic TICG model.) While this additional modeling is not necessary for accurate static concurrency analysis, it adds a great deal of clarity to the model when it is inspected by the programmer.

Code added by instrumentation system.

- **start task** At the start of each task body the following instrumentation code is inserted

  STMT_ID is an integer that is the statement ID that corresponds to the body of this task.

```
begin
  declare
    Artemis_Child: Artemis.Dynamic_Task_ID_Type;
  begin
    Artemis.Start_Task( STMT_ID, Artemis_Child);
```

Figure 3.3: instrumentation code inserted at start of each task

The runtime portion of the monitoring system takes the following actions:

- It generates a new DTID, and calls the Verdix Ada runtime system to store this ID within the Task Control Block for the current task, so that we can retrieve it later.
- Writes a Start_Task event into the log, recording the STID and DTID, thus allowing us to later establish the mapping between a given DTID and the corresponding STID.
- Sets the out parameter corresponding to Artemis_Child so that the task will be aware what its DTID is.

- **accept statement**

```
accept  ENTRY_NAME( Artemis_Task_ID : in Artemis.Dynamic_Task_ID_Type;
                    Call_Stmt_ID : Artemis.Statement_ID_Type ;
                     ORGINAL_PARAMETERS ) do
  Artemis.Rendezvous_Entry(Artemis_Task_ID,  STMT_ID, Call_Stmt_ID);
                    ORIG_CODE;
  Artemis.End_Rendezvous(Artemis_Task_ID,  STMT_ID, Call_Stmt_ID);
end  ENTRY_NAME;
```

Figure 3.4: instrumentation code inserted at each accept

- **entry call**

```
TASKNAME.ENTRY_NAME ( Artemis.Current_Task, STMT_ID, ORIG_PARAMETERS)
```

Figure 3.5: instrumentation code inserted at each entry call

- **end of task**

```
   Artemis.Finish_Task( STMT_ID);
exception
   when others =>
      Artemis.Exception_Exit_Task( STMT_ID );
end TASK_NAME;
```

Figure 3.6: instrumentation code inserted at the end of each task

STMT_ID is the same one given in Start_Task. The exception code is so that we are notified if a task is aborted on an exception.

- **Non-rendezvous tasking related interactions**
  - 'delay' alternatives in selective waits and timed entry calls
  - 'else' parts taken in selective waits/conditional entry calls
  - TERMINATE alternatives are taken in selective waits

```
Artemis.Start_Subprogram( FAKE_SUBPROG_ID );
```

Figure 3.7: instrumentation code inserted for non-rendezvous tasking interactions.

Each of these is recorded by generating a Start_Subprogram event with an ID that is generated for this statement.

- **Data access** When the user indicates that accesses to a particular variable should be traced we generate the following statements:

```
Artemis.Execute_Statement( PRE_STMT_ID);
 ORIGINAL_ASSIGNMENT;
Artemis.Execute_Statement( POST_STMT_ID);
```

Figure 3.8: instrumentation code inserted to trace variable accesses.

We generate a separate PRE and POST STMT_ID for each accesss to the variable.

When building the TICG from a collected trace we follow the algorithm given in Figure A.1.

3.3 Using SCA to expand the initial TICG

Our system also allows the user to incrementally expand the TICG to include other states which the program's structure would allow, but which weren't observed in this execution.

One motivation for exploring portions of the state space which represent executions other than the failed one is to prevent related faults from causing more failures in the future. These additional failures can occur in several ways. One common mistake made in debugging concurrent programs is "fixing" the symptom (a failure), while missing the underlying fault. As seen in Figure 3.9 there are two manifestations of this problem, both of which can be seen when the TICG is examined. The first case occurs when there are multiple paths to

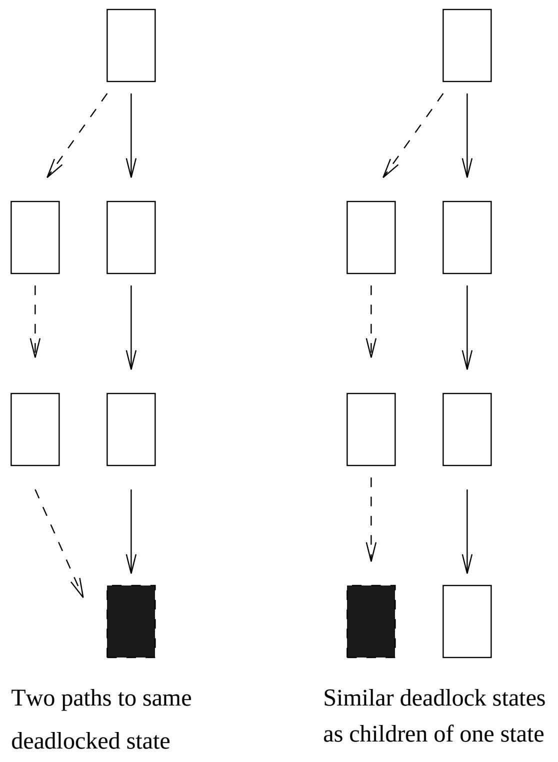


Figure 3.9: Similar concurrency paths leading to deadlock

a particular failed state. In this case eliminating the path that was taken on the failed execution will not stop the problem from occurring. This condition can occur based on the ordering of task transitions in threads which have no enforced partial order relationship. Another case is when there is a similar state which is deadlocked, and which differs from the original failed state by a small amount, perhaps only a few task transitions. This can happen when tasks not directly involved in the deadlock affect the behavior of the deadlocked tasks.

These kinds of faults are seen when using the Concurrency Analysis Tool Suite (CATS.) CATS [YTL+95] builds and does deadlock analysis of complete TICGs. When CATS detects a potential deadlock state it reports the error both in terms of the final state of each task, as well as a description of the states that led to this deadlock. If there is more than one sequence of concurrency states that leads to the same failed state we will see multiple reports with the same final state. We also often see different deadlock states where much of the context leading to the failed state is similar, but the final failed states differ in exactly which tasks are deadlocked.

3.4 Incremental Forward Generation of States

Using incremental state generation to avoid infeasible states The states generated during conventional SCA are a superset of all states that could be visited in any execution of the program. It is a superset because it usually contains many infeasible states that could not occur in any real execution. These are generated because after the data and non-tasking control flow information are abstracted away it may be impossible to tell whether the program logic prevents certain rendezvous from occurring. In this case SCA makes the pessimistic assumption that the tasking action is possible, to avoid missing any potential errors.

Abstracting away the data and control flow was necessary to minimize the number of states, but it introduced inaccuracies. If we do our successor generation under user control, the user can cut off exploration of paths that he knows to be unexecutable. By avoiding following paths that contain infeasible states, we limit the number of states generated, and this allows us to investigate a larger portion of the feasible states.

Using incremental analysis to check corrections or enhancements As mentioned previously concurrency faults occur because synchronization code (represented by a state) is incorrect, leading to incorrect task interactions (represented by edges,) which ultimately lead to failures (represented by a state.) To correct the program, we must identify and modify the faulty state, to either eliminate a faulty transition or to add a missing transition.

It may be that we incorrectly postulate the initial cause. In this case we would modify the code for a state we believe to be faulty, but there may be other paths to the failed case we observed, which do not pass through our hypothesized faulty state.

To help detect this type of incorrect diagnosis we would begin our analysis with a state that existed on the failed path, but prior to the one we have identified. We then build forward from this state. We should end up generating the hypothesized faulty state and the observed failure state, but we may also generate other failure states along paths that don't pass through our hypothesized faulty state. In this case these additional paths represent other executions that could fail.

We may also wish to do partial SCA when we have made enhancements to a previously correctly functioning concurrent program, to ensure that our changes have not introduced new faults.

3.5 Backward Generation of States

Additional states and paths through the TICG can be generated by running a variant of the SCA algorithm in reverse, generating potential predecessor states rather than successors.

There are several cases where this could be useful.

- If we have changed our program in a way that we believe eliminates the possibility of reaching the deadlock state observed in a failed execution, we may wish to do analysis on the revised program to make sure that the failure state we observed is indeed no longer reachable.
- We observe a failure where we don't have enough trace information to definitely determine what exact state the program failed in, or how it got there. This occurs when we must collect traces of programs that run for very long periods of time, because we will be unable to retain the entire trace. Instead we utilize a fixed size circular buffer that discards the oldest information. We could also use this method if we did not enable the tracing facility on the execution which failed.

  In this case we may be able to hypothesize a failed state that is consistent with our observation of the execution, and now we use backward analysis to determine how that state could be reached. This type of analysis where we hypothesize a particular failed state and then search backwards looking for paths where this could happen bears some similarity to safety analysis using Fault Tree Analysis. In both cases any path from the start state to an undesired state indicates problems with the program that must be investigated.

Infeasible states generated during backwards state generation. When building backwards we generate additional unreachable states, in addition to those which are generated when building forward. These states never lead back to the program start state. These infeasible states are infeasible because reaching them would require other tasks to take mutually exclusive paths.

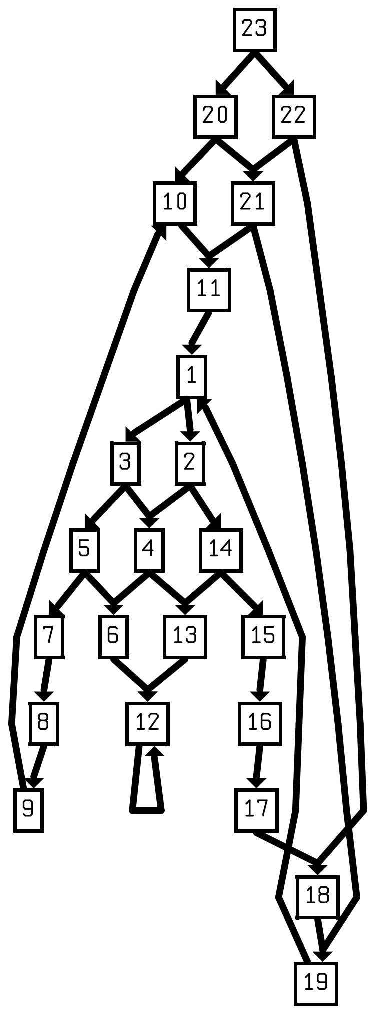


Figure 3.10: TICG with infeasible states generated by backwards generation. TICG for two deadlocking version of dining philosophers example with infeasible states generated by backwards generation. Compare this to the version without additional infeasible states shown in Figure 3.2 on page 50.

State 20	two_phil.phil1	S+(two_phil.fork2.down)
	two_phil.fork1	E-(down)
	two_phil	
	two_phil.fork2	S-(down)
	two_phil.phil2	E+(two_phil.fork1.down)
State 21	two_phil.phil1	E+(two_phil.fork2.down)
	two_phil.fork1	E-(down)
	two_phil	
	two_phil.fork2	E-(down)
	two_phil.phil2	E+(two_phil.fork1.down)
State 22	two_phil.phil1	E+(two_phil.fork2.down)
	two_phil.fork1	S-(down)
	two_phil	
	two_phil.fork2	E-(down)
	two_phil.phil2	S+(two_phil.fork1.down)
State 23	two_phil.phil1	S+(two_phil.fork2.down)
	two_phil.fork1	S-(down)
	two_phil	
	two_phil.fork2	S-(down)
	two_phil.phil2	S+(two_phil.fork1.down)

Figure 3.11: Infeasible states generated by backwards state generation in two dining philosophers example.

An example of this is present in the dining philosophers example given in Figure 3.1 If we generate the TICG for the deadlocking version with two diners we get some additional TICG states beyond those generated by the forward example. This new TICG is shown in Figure 3.10 The state information for the new, infeasible states (number 20, 21, 22 and 23) is shown in Figure 3.11 All of these states are quite similar, so we will illustrate state 21 as an example.

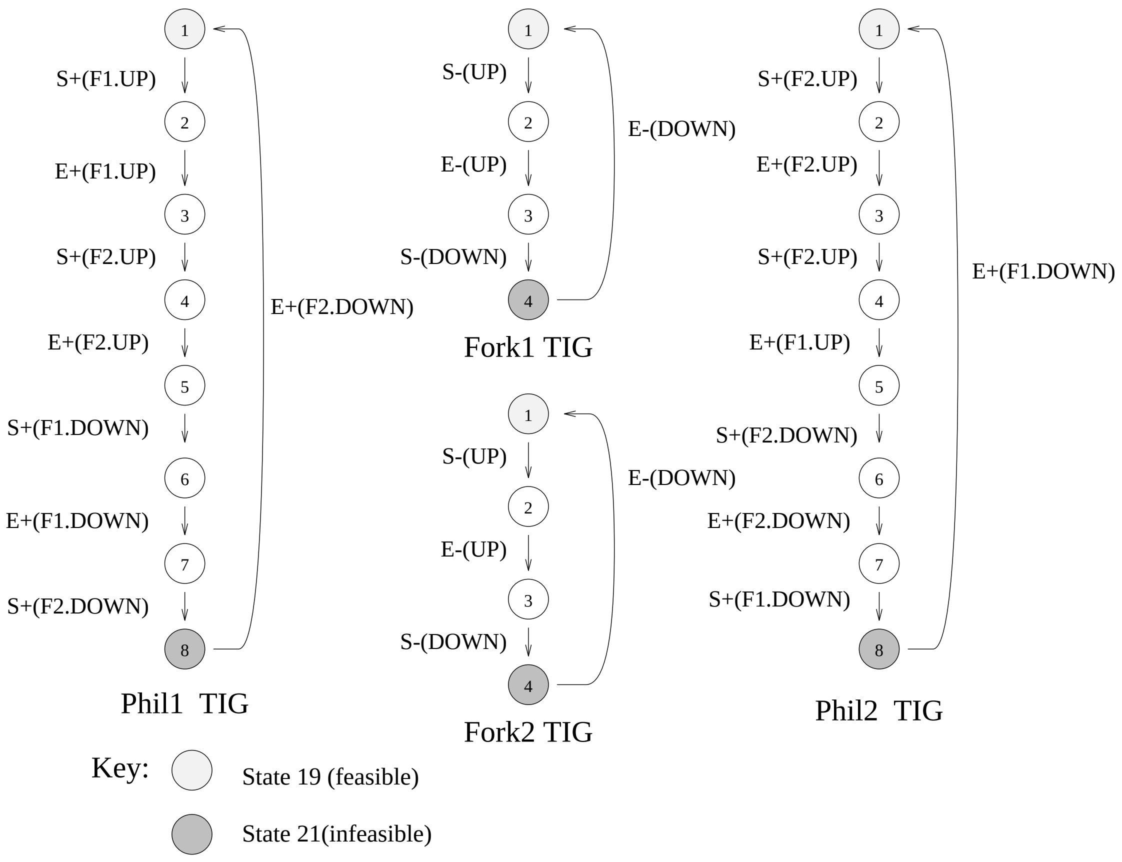


Figure 3.12: TIGs for a fork and philosopher. We generate states in the backward direction by following the edge representing an entry call and the corresponding accept from the edge targets to the source.

State 19
(feasible)

two_phil.phil1	S+(two_phil.fork1.up)
two_phil.fork1	E-(down)
two_phil	
two_phil.fork2	S-(up)
two_phil.phil2	E+(two_phil.fork1.down)

State 21
(infeasible)

two_phil.phil1	E+(two_phil.fork2.down)
two_phil.fork1	E-(down)
two_phil	
two_phil.fork2	E-(down)
two_phil.phil2	E+(two_phil.fork1.down)

Figure 3.13: A feasible state and the infeasible state generated as its predecessor

State 21 was created by backwards generation from state 19, and the state information for these states is shown in Figure 3.13. Backwards state generation proceeds by trying to determine a rendezvous that could have occurred to put the system into the current state. From examining the TIG states for the individual tasks, in state 19 it appears that Phil1 may be just finished putting down Fork2. This state is shown in terms of the source code in Figure 3.14 This state is infeasible because in order for Phil1 to put down Fork2, it must have held Fork2 at that time. But it is impossible for Phil1 to hold Fork2 while Phil1 holds Fork1, which is indicated by the current state of Phil2.

Generation of these additional infeasible states is not limited to this example. To obtain an initial approximation of how many of these additional infeasible states are generated, we did complete SCA of several small examples, then generated all predecessor states using our backward generation algorithm, and repeated the generation until no new predecessor states were generated. The results of this exercise are shown in Table 3.1

```
  task body Fork1 is
  begin
     loop
        accept UP;
⇒       accept DOWN;
     end loop;
  end Fork1;

  task body Fork2 is
  begin
     loop
        accept UP;
⇒       accept DOWN;
     end loop;
  end Fork2;


  task body Phil1 is
  begin
     loop
        Fork1.UP;
        Fork2.UP;
        Fork1.DOWN;
⇒       Fork2.DOWN;
     end loop;
  end Phil1;

  task body Phil2 is
  begin
     loop
        Fork2.UP;
        Fork1.UP;
        Fork2.DOWN;
⇒       Fork1.DOWN;
     end loop;
  end Phil2;
```

Figure 3.14: State 21 is infeasible, but backwards SCA generates it because it only considers actions one pair at a time.

Table 3.1: Results of experiments to determine number of infeasible states generated for several popular examples.

program	forward feasible states	total including infeasible	percent infeasible
one elevator	1235	1636	25
two elevator	6676	10459	36
two philosophers	19	23	17
four philosophers	511	527	3
gas station	649	969	33
remote temperature sensor	651	755	14

The results in this table demonstrate that the number of infeasible states generated varies widely. There is not even a correlation with increasing the size of each example, because with the elevator example the percentage of infeasible states went down with increasing problem size, while the percentage of infeasible states increased as the elevator example was made larger. This difficulty in predicting the number of states generated mirrors that found for forward generation in [Lev93].

These additional infeasible states differ from the infeasible states generated in the forward direction in that they are not artifacts of having removed data-flow information, but instead they result from not considering how the concurrency structure constrains when certain tasking actions can occur. Thus the problem is not that the information is unavailable, but rather that we haven't fully exploited it.

Many of these states can be pruned by a user aware that they can be generated. The user may also write filters that recognize patterns describing sequences of rendezvous that can't occur. Finally it may be possible to write an algorithm that will allow us to avoid generating some of these infeasible states, but there are many subtleties to this calculation, especially in the case where a task has more than one accept for the same entry. This noticeably limits the applicability of such an approach, and reduces the number of infeasible states that can be avoided.

3.6 Limiting the TICG portion generated or displayed

We have previously mentioned that it is necessary to limit the number of states generated, and we allow the user to control which states are generated. It is usually necessary to further limit the number of states displayed to be fewer than those generated because of the limited amount of screen area available for display. It is possible to display a few hundred states on a 20 inch monitor in a readable size, while SCA can produce a graph of several thousand states in less than a minute. The layout algorithm we use to draw the graphical depiction of the TICG can place a few hundred nodes in a few seconds, and attempting to display many more nodes (say 1000) can result in a long delay waiting for the layout to be calculated.

One simple approach would be to just generate nodes in the order than SCA would normally generate them, and stop the generation when a set number of nodes has been generated, but we take a different approach. Rather than just generating a fixed number of nodes when the user requests that more nodes be generated or displayed, we instead generate a certain number of levels of nodes. We call this number of levels the "display depth." With a display depth of three, we would generate all the children of the selected node, all the children of those nodes (grandchildren of the original node) and finally all the children of this second level of nodes (great-grandchildren of the original node.) The algorithm for calculating what nodes to include in a display for level N is given in figure A.5.

3.7 Task Subsetting

When doing the forward or backward state generation we can choose to analyze a subset of the tasks in the program. To ensure the accuracy of analysis conventional SCA always considers the potential movements of all tasks from any given state. It may be the case that the programmer believes that some tasks are not responsible for the failure being diagnosed. In this case it may be possible to limit the size of the generated TICG by ignoring the interactions of those tasks that are considered "uninteresting." This may result in inaccurate modeling, and missing faulty states, but this technique can be very useful because it makes possible the generation of more of the states involving the interesting tasks.

When doing task subsetting our method is to modify the TIG representation to avoid interactions involving uninteresting tasks. It would seem that simply eliminating the TIGs that describe uninteresting tasks would be sufficient, but there are a few intricacies that must be taken into account. If we remove an uninteresting task that makes an entry call on an interesting task, the interesting task can block indefinitely waiting to accept an entry call from a task that has been removed. If this entry of the interesting task is only called by uninteresting tasks then we may safely remove both the uninteresting task and the entry from the interesting task. If this entry is also called by interesting tasks we can not remove the entry, and instead we must retain the task which is otherwise uninteresting. To determine which entries can be removed, and which tasks specified by the user must be retained because they call interesting task entries we utilize the algorithm shown in Figure A.3.

Task subsetting is similar to parceling [You89]. Both require a knowledgable user, and incorrect choices can lead to inaccurate modeling. In all cases where there is an ambiguity of whether a node can be safely removed we give the user control, and rely on his knowledge of the program.

3.8 Data Tracing

Concurrency data anomalies occur when the results of a program vary based on the relative speeds of the tasks. If one task produces a value that another reads we must have synchronization to ensure that the producer task has finished writing the value before the reader tries to access it. If this locking is missing or incorrect then the reading task will get an incorrect value. If we have collected information about the concurrency states that exist when a particular variable is accessed that can help us diagnose missing or faulty synchronization.

Race conditions are suspected when the program sometimes yields incorrect results for a given input. There should be a well defined set of program concurrency states in which the engineer expects the variable to be accessed. A correct locking strategy results in data being accessed only in these correct states. Thus if we can determine in which states data is accessed, we can help the engineer diagnose these problems. We provide this information by adding instrumentation to record the concurrency state that exists when a variable is accessed (either read or written.) In general, only the tasks that read and write the variable in question must be examined, and the states of other tasks can be safely ignored.

To provide information about the concurrency state that exists when a variable is accessed, we add the tracing statements shown in Figure 3.8. This information is read back when the initial TICG is constructed from the trace of a failed execution, and we annotate the TICG with this data access information. It is a possible for one variable access to be associated with more than one TICG state. There can't be any other tasking actions of the task that contains the variable being accessed, but other tasks can rendezvous while the data is being accessed. In this case we annotate all the states. This is potentially extremely useful information in the case of shared variables because if the user sees two separate variable accesses with overlapping concurrency states this is strongly indicative of a data race.

3.9 Correctness of modifications made to TICG model

In this section we address the correctness of our modified TICG model. We start with two assumptions:

1. The basic TICG model is correct

   The TIG/TICG model of concurrency has received considerable study [PTY95] [YTL+95], and we chose it because it has a well documented semantics. We will assume that the basic model accurately represents the Ada program for which is was generated. Given this assumption, it remains fair to ask whether the modifications we have made to the model have introduced inaccuracy.

2. We are concerned only with the accuracy of deadlock detection and we will not consider the implementation and checking of Temporal Logic Assertions. We may wish to add these features in the future, but they are beyond the scope of our current research.

3. We assume the correctness of the deadlock checking algorithm used with the TIG/TICG model, which is described in [YTL$^+$95].

The deadlock checking algorithm treats deadlock as a local rather than a global property. This means deadlock is checked at each state independently, and a system is declared deadlock free if and only if all the states in its complete TICG are deadlock free.

Given that we are assuming the accuracy of both the basic TICG model and deadlock checking algorithm for a particular state, this reduces the problem of deciding whether our modeling is accurate to the question of whether we produce the same states as standard SCA would. If so, our modeling is also accurate. The two forms that such inaccuracy could take are:

- Missing states - States present in the TICG generated by SCA, but not by our approach. Any missing states could contain deadlocks, and if they aren't generated the deadlock checker won't be able to detect and report the potential deadlocks.

- Extra states - States present in our model that aren't present in the TICG generated by SCA. If we assume that SCA generates a safe overestimate of the real state space of a program, then these additional states must infeasible. Therefore they can't contain any real deadlocks, but could generate spurious deadlock reports. Since basic SCA already often produces spurious deadlock reports, these additional ones can be a problem in causing the analyst more work to show that they are infeasible. This is a less serious problem than missing a real deadlock, but still something we would hope to minimize.

We avoid one potentially large source of inaccuracy by using the same algorithm for generating the successors of an individual state in the forward direction as described in the basic model. We will now describe the effects of our modifications of the basic TICG model, and how these affect the accuracy of deadlock detection.

What follows is a list of our modifications to the basic TICG model, which are the potential sources of inaccuracy in our modified approach.

1. Incomplete expansion of graphs

   This change from basic SCA could cause real deadlocks to be missed, if a state that is not generated contains a deadlock. This is unavoidable, since we are explicitly trading the completeness of analysis for the ability to deal with larger systems. Given this limitation, there are two questions that arise:

   - Do we correctly generate deadlock reports for the states that we do produce?

Each of the states in the partial graph was generated using the SCA algorithm, and is checked using the SCA deadlock checking algorithm, so yes, our model will give accurate results with respect to the portion generated.

- If we decide to do complete expansion, we get complete results.

  Again, we do our successor generation using the algorithm from SCA, and we keep track of which states have been fully expanded. If we fully expand all states we end up with the graph that would have been produced by pure SCA.

2. Pseudo-rendezvous.

   - Does adding for pseudo-rendezvous for delay-expire cover up deadlocks?

     No. The places where we add these are zero-edge groups, which CATS ignores for purposes of deadlock detection. It assumes that there will always be a timeout that occurs, and doesn't check this task for deadlock.

   - Does it introduce new spurious deadlock reports.

     No. These pseudo-task edges call entries that are always ready to accept, because they have no way to block.

3. Task subsetting.

   - Can task subsetting cause deadlocks to go unreported?

     Yes, it can cause us to miss existing deadlock if and only if we exclude a task that is involved in a deadlock from the "interesting" set.

Imagine a task T that just does "accept E" with no caller for the entry E. This program will deadlock when all other tasks have terminated and T is still blocked. If we remove T we will not get this deadlock report.

Thus accuracy of deadlock detection is dependent on the user choosing the correct set of interesting tasks.

- Does subsetting introduce new infeasible deadlocks?

  Yes, it can, if we override the subsetting algorithm which attempts to avoid the removal of tasks that call entries of interesting tasks.

4. Backwards state generation

   - Can backwards state generation cause missed deadlocks?

     No. Because backwards state generation can only add states to the graph.

   - Can it add infeasible deadlocks?

     Yes. The states we generate going backwards can't be deadlock states because they have the states they were generated from as successors. But, the states that are generated by going forward from these states may be infeasible, and can have deadlocks reported for them.

Chapter 4

Examples

In this chapter we present examples that demonstrate how our methodology can be applied to debugging both deadlock and data-race faults. While these examples are necessarily small to allow explanation in this short paper, the deadlock example has sufficient complexity that it could not be analyzed by the CATS system.

4.1 Using the TICG to Understand an Execution

Our first example is a simple simulation of an elevator system. This example illustrates several important concepts and features of our system:

1. Use of the TICG as a model to understand a particular execution.
2. The reduction state space size obtained from task subsetting.
3. Handling arrays of task types.

The elevator simulator example has been used many several times in formal specification and testing papers. The fault that we will demonstrate was not seeded for this discussion, but rather was accidentally introduced in the original version of this program when it was implemented by a programmer quite experienced with writing concurrent Ada software. The bug manifested itself only when the program was run on a machine with a very high load average. At all other times it ran correctly. The failure results from race condition involving a guard condition that represents a change in elevator state.

In Figures 4.1 and 4.2 the boxed statements in each task show which statements were active when the deadlock occurred. The elevator_sim_task for an elevator has an 'or delay' alternative that it uses to move move the elevator from one floor to another while that elevator should be moving (controlled by the variable Cur_Dir.) Every time the elevator reaches a new floor it calls Controller_Pkg.Controller.At_Floor to notify the controller that it has reached a new floor. The controller checks to see if this floor is the goal floor, and if so sets the Cur_Dir for this elevator to Idle, causing the elevator to stop moving. However under conditions of heavy load it is possible for Elev_Sim_Task to call At_Floor, complete the rendezvous, and have its delay between floors expire again before Controller can set this elevator's direction to idle.

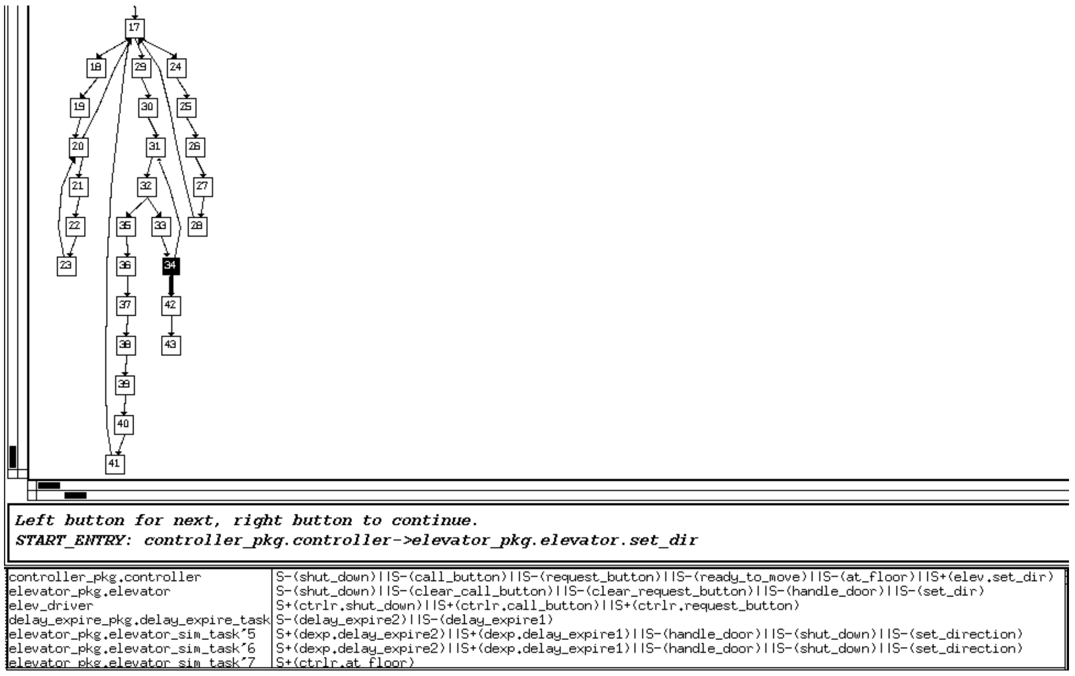
Conventional SCA using CATS was attempted on this example, but CATS was unable to complete the analysis because it generated too many states (well over 50,000) and building the full TICG consumed all available storage. CATS could only successfully handle a single elevator, although in this case it was sufficient to discover the fault. CATS also required that the array of tasks be converted into static tasks. While this exact set of limitations are particular to the CATS system, it serves as an example of how the exponential nature of SCA limits its use on what seems to be a fairly simple example.

```
Elev_Sim : array(1..3) of Elev_Sim_Task;

task body Controller is
  Goal_Floor: boolean;
begin
  loop
    select
      accept At_Floor(Elev:   Elevator_Id;
                      Floor: Floor_Id) do
        Goal_Floor := Is_Goal_Floor(Elev, Floor);
      end;
        -- Is this floor a goal?
      if Goal_Floor then
          -- Open and close the doors and then
          -- restart the elevator
        Elevator_Pkg.Set_Dir(Elev,Idle);
        Elevator_Pkg.Handle_Door(Elev);
      end if;
    or
      -- accept's for Ready_To_Move, Request_Button,
      -- Call_Button and Shut_Down
    end select;
  end loop;
end Controller;

task body Elevator is
begin
  loop
    select
      accept Set_Dir(Elev:Elevator_ID;
                     Dir:Direction) do
        Tmp_Elev := Elev;
        Tmp_Dir := Dir;
      end;
      Elev_Sim(Tmp_Elev).Set_Direction(Tmp_Dir);
    or
      accept Handle_Door(Elev:Elevator_ID) do
        Tmp_Elev := Elev;
      end;
      Elev_Sim(Tmp_Elev).Handle_Door;
    or
        -- accepts for Clear_Request_Button,
      -- Clear_Call_Button and Shut_Down;
    end select;
  end loop;
end Elevator;
```

Figure 4.1: Code for Elevator Controller and Elevator Package

```
  task body Elevator_Sim_Task is
    loop
      select
        accept Set_Direction(Dir:Direction) do
          Cur_Dir := Dir;
        end;
      or
        accept Handle_Door;
        Door_Pkg.Open_Door(My_Id);
        Doors_Open := true;
      or when Doors_Open =>
        delay Open_Door_Delay;
        Door_Pkg.Close_Door(My_Id);
        Controller_Pkg.Ready_To_Move(My_Id);
        Doors_Open := false;
      or when Cur_Dir /= Idle and not Doors_Open =>
        delay Secs_Per_Floor;
        if Cur_Dir = Up then
          Cur_Floor := Cur_Floor + 1;
        else
          Cur_Floor := Cur_Floor − 1;
        end if;
        Controller_Pkg.At_Floor(My_ID,Cur_Floor,Reached_Goal);
        if Reached_Goal then
          Cur_Dir := idle;
        end if;
      end select;
    end loop;
  end Elevator_Sim_Task;

procedure Elev_Driver is
begin
  while not done loop
    text_io.get(command);
    case command is
      when 'r' =>
        controller.request_button(Elev,Floor);
      when 'c' =>
        controller.call_button(Floor,Dir);
      when 'q' =>
        done := true;
  end loop;
  controller_pkg.controller.shut_down;
end Elev_Driver;
```

Figure 4.2: Code for Elevator Simulator Task and Main Program



```
Left button for next, right button to continue.
START_ENTRY: controller_pkg.controller->elevator_pkg.elevator.set_dir
```

```
controller_pkg.controller         S-(shut_down)||S-(call_button)||S-(request_button)||S-(ready_to_move)||S-(at_floor)||S+(elev.set_dir)
elevator_pkg.elevator             S-(shut_down)||S-(clear_call_button)||S-(clear_request_button)||S-(handle_door)||S-(set_dir)
elev_driver                       S+(ctrlr.shut_down)||S+(ctrlr.call_button)||S+(ctrlr.request_button)
delay_expire_pkg.delay_expire_task S-(delay_expire2)||S-(delay_expire1)
elevator_pkg.elevator_sim_task'5  S+(dexp.delay_expire2)||S+(dexp.delay_expire1)||S-(handle_door)||S-(shut_down)||S-(set_direction)
elevator_pkg.elevator_sim_task'6  S+(dexp.delay_expire2)||S+(dexp.delay_expire1)||S-(handle_door)||S-(shut_down)||S-(set_direction)
elevator_pkg.elevator_sim_task'7  S+(ctrlr.at_floor)
```

Figure 4.3: System output for failed execution of elevator. The graph at top shows a portion of the TICG built from the trace. The text box immediately below gives a description of the highlighted task transition which led to failure. The bottom shows the task state window which contains the state from which the failed transition occurred.

In the original version of the elevator the program output a statement every time an event occurs (e.g. a call button is pressed, the elevator reaches a new floor, the doors open or close etc.) To ensure that each message is printed to the console without the danger of two messages appearing with portions interspersed, there is a CONSOLE task which assures exclusive access to the console. The calls to CONSOLE.PRINT_MESSAGE have been removed from the code shown in listing, due to the limited space available, but they were present in the code which was analyzed. With these statements in the code the following statistics were generated:

Table 4.1: Elevator TICG size with and without subsetting

elevators	states	edges
1	1235	2608
2	6676	16305
3	Too many states to complete TICG construction (> 50_000)	
3/no CONSOLE	5242	15613

Using the task subsetting feature to remove the calls to the CONSOLE task eliminated a large portion of the state space, reducing it to approximately the size of the full two elevator example.

We collected a trace and built the initial TICG. The observed execution had the elevators running normally for several requests before the failure occurred. Animating the TICG showed that the program executed along the same paths through the TICG several times, then followed a different path and deadlocked. This final failed state was number 43, and it and state 42 were only visited when the execution failed, whereas normally state 34 led back to state 31, representing the elevator_sim_task floor being reached that is not a goal floor, but in this case the controller attempts to have the elevator stopped after the inter-floor delay has expired.

Explanation of SADCPU display The task state window which appears at the bottom of Figure 4.3 in used to examine TICG states. For each task it lists the potential next actions of that task. The left half of the window give the name of the task, while the right half indicates what next action or actions comprise the transitions from this state. In the task names portion task with names suffixed with a tick mark and a number (e.g. elevator_pkg.elevator_sim_task'7) are instantiations of task types, and the number is the DTID for this task. This does not necessarily correlate to the array index if this is an array of task types, and it is up to the user to determine the mapping back to these anonymous tasks. In the future it would be useful to allow the user to assign meaningful names to these tasks. 'S' and 'E' in the right half of the display indicate the start and end of a task interaction, while '+' and '−' represent entry calls and accept statement respectively. The double vertical bars '||' separate multiple next actions. More than one next action occurs for both tasking non-determinism (e.g. the select statement or timed entry calls) and conditional control flow (if-then or looping.)

This demonstrates a common use of the system, which is to interactively explore the state space. The engineer examines states in turn, matching the behavior described by the trace to that expected from the program design. In this example it is obvious that the other elevator_sim_tasks could follow paths to similar deadlocks, but if the programmer wanted to verify this it could easily be checked by generating the appropriate additional portions of the TICG.

4.2 Combining Execution Tracing with SCA

In the previous example the engineer manually inspected the TICG to diagnose the fault. In this example we demonstrate a different way of using the system, involving more automated analysis. In this case we partially expand the TICG using the common SCA techniques, and then perform automatic deadlock detection.

This scenario demonstrates the following features of our system:

1. Using SCA under user control to expand a partial TICG built from an execution trace
2. Doing automatic deadlock checking on a partial TICG
3. Using SCA to check a proposed corrective change

The program we will use to demonstrate these techniques is a simulation of an automated gas station, derived from that given in [HL85]. The design entities are gasoline pumps, customers who wish to purchase gas, and an automated operator who collects payments from the customers and controls the pumps.

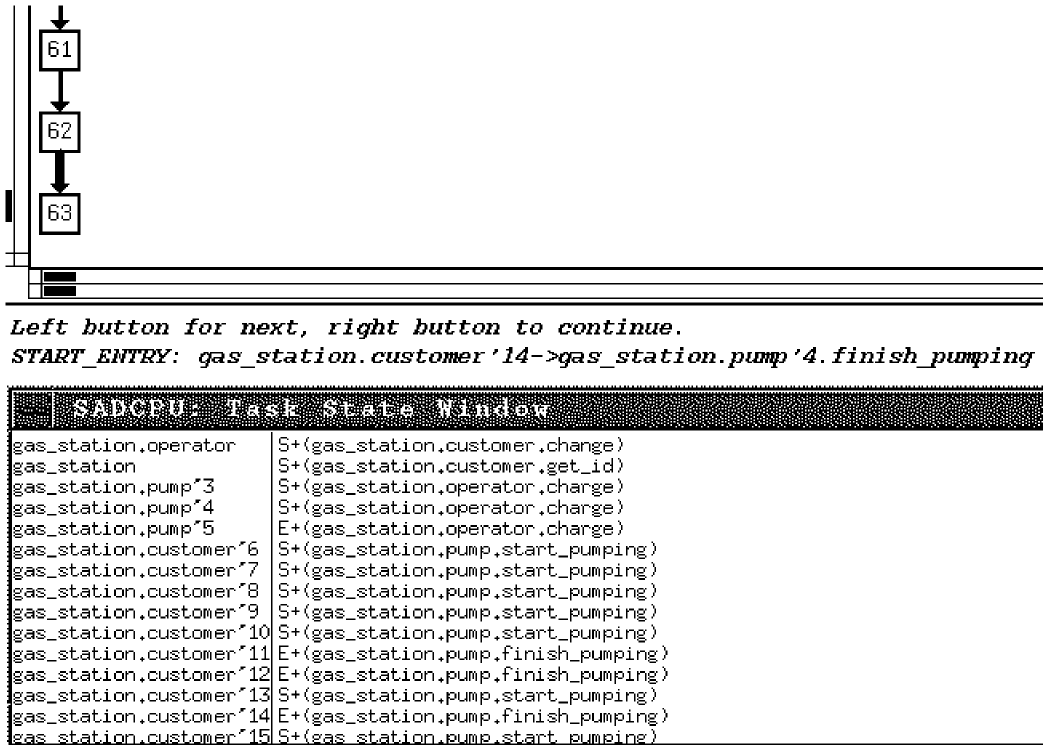
The initial version of the program is shown in Figures 4.4 and 4.5. This version deadlocks, with the failure state shown in our system output for the failed state shown in Figure 4.6. As can be seen in the system output, task OPERATOR is trying to call entry CHANGE of task CUSTOMERS, CUSTOMERS is calling entry FINISH_PUMPING of task PUMPS, and task PUMPS is calling entry CHARGE of task OPERATOR.

```
with COMPUTER, RANDOM;
procedure GAS_STATION is
  type CUST_ARRAY is array( integer range <>) of CUSTOMER;
  type CUST_ARRAY_PTR is access CUST_ARRAY;

  CUSTOMERS : CUST_ARRAY_PTR;

  PUMPS : array(1..3) of PUMP;

  task body OPERATOR is
    AMOUNT_PAID, CUSTOMER_NUMBER : INTEGER;
    CHANGE_AMOUNT, IDLE_PUMP     : INTEGER;
  begin
    loop
      select
        accept PRE_PAY(AMOUNT : in INTEGER;
                       PUMP_ID: in  INTEGER;
                       CUSTOMER_ID: in INTEGER) do
          -- if no previous customer is waiting, activate this pump
          if COMPUTER.EMPTY_QUEUE(PUMP_ID) then
            PUMPS(PUMP_ID).ACTIVATE(AMOUNT);
          end if;
          -- enter a record of prepayment; records for each pump are
          -- serviced in FIFO order
          COMPUTER.ENTER(PUMP_ID, AMOUNT, CUSTOMER_ID);
        end PRE_PAY;
      or
        accept CHARGE(AMOUNT  : in INTEGER;  PUMP_ID: in INTEGER) do
          -- retrieve current record for this pump, i.e., the caller
          COMPUTER.RETRIEVE(PUMP_ID, AMOUNT_PAID, CUSTOMER_NUMBER);
          -- give change to the customer
          CUSTOMERS(CUSTOMER_NUMBER).CHANGE(AMOUNT_PAID − AMOUNT);
          -- if another customer is waiting for this pump, activate it
          if not COMPUTER.EMPTY_QUEUE(IDLE_PUMP) then
            PUMPS(IDLE_PUMP).ACTIVATE(COMPUTER.TOP_AMOUNT(IDLE_PUMP));
          end if;
        end CHARGE;
      end select;
    end loop;
  end OPERATOR;
```

Figure 4.4: Operator task for Initial Version of Gas Station

```
  task body PUMP is
    MY_NUMBER : INTEGER;
    CURRENT_CHARGES, AMOUNT_LIMIT : INTEGER;
  begin
    accept GET_ID(ID : INTEGER) do
          MY_NUMBER := ID;
    end GET_ID;

    loop
      accept ACTIVATE(LIMIT : in INTEGER) do
        AMOUNT_LIMIT := LIMIT;
      end ACTIVATE;
      accept START_PUMPING;
      accept FINISH_PUMPING(AMOUNT_CHARGED : out INTEGER) do
        CURRENT_CHARGES := RANDOM.GET_CHARGES(AMOUNT_LIMIT);
        AMOUNT_CHARGED := CURRENT_CHARGES;
        OPERATOR.CHARGE(CURRENT_CHARGES, MY_NUMBER);
         end FINISH_PUMPING;
    end loop;
  end PUMP;

  task body CUSTOMER is
    MY_NUMBER : INTEGER;
    MY_MONEY, PUMP_NUMBER, AMOUNT_PUMPED : INTEGER;
  begin
    accept GET_ID(ID : INTEGER) do
          MY_NUMBER := ID;
    end GET_ID;

    loop
      RANDOM.DRIVE_IN(MY_MONEY, PUMP_NUMBER);
      OPERATOR.PRE_PAY(MY_MONEY, PUMP_NUMBER, MY_NUMBER);
      PUMPS(PUMP_NUMBER).START_PUMPING;
      PUMPS(PUMP_NUMBER).FINISH_PUMPING(AMOUNT_PUMPED);
         accept CHANGE(AMOUNT : in INTEGER);
    end loop;
  end CUSTOMER;
begin
  COMPUTER.GET_NUMBER_OF_CUSTOMERS;
  CUSTOMERS := new CUST_ARRAY(1..COMPUTER.CURRENT_CUSTOMERS);
  for K in 1..COMPUTER.NUMBER_OF_PUMPS loop
     PUMPS(K).GET_ID(K);
  end loop;
  for K in 1..COMPUTER.CURRENT_CUSTOMERS loop
     -- assign CUSTOMER ids
     CUSTOMERS(K).GET_ID(K);
  end loop;
end GAS_STATION;
```

Figure 4.5: Pump and Customer Tasks for Initial Version of Gas Station



```
Left button for next, right button to continue.
START_ENTRY: gas_station.customer'14->gas_station.pump'4.finish_pumping
```

```
gas_station.operator     S+(gas_station.customer.change)
gas_station              S+(gas_station.customer.get_id)
gas_station.pump'3       S+(gas_station.operator.charge)
gas_station.pump'4       S+(gas_station.operator.charge)
gas_station.pump'5       E+(gas_station.operator.charge)
gas_station.customer'6   S+(gas_station.pump.start_pumping)
gas_station.customer'7   S+(gas_station.pump.start_pumping)
gas_station.customer'8   S+(gas_station.pump.start_pumping)
gas_station.customer'9   S+(gas_station.pump.start_pumping)
gas_station.customer'10  S+(gas_station.pump.start_pumping)
gas_station.customer'11  E+(gas_station.pump.finish_pumping)
gas_station.customer'12  E+(gas_station.pump.finish_pumping)
gas_station.customer'13  S+(gas_station.pump.start_pumping)
gas_station.customer'14  E+(gas_station.pump.finish_pumping)
gas_station.customer'15  S+(gas_station.pump.start_pumping)
```

Figure 4.6: System output for initial gas station deadlock.

In [HL85] Helmbold and McDowell suggest the following series of changes the programmer might make in an attempt to remove the deadlocks from this program.

1. Moving the entry call to the customer out of the accept body of CHARGE. This eliminates this failure, but rerunning the program leads to a different deadlock state.

2. Changing the order of entry calls in task OPERATOR so that the PUMP is activated for the next CUSTOMER before the first CUSTOMER is given his change. Again this fixes the detected deadlock, but leads to a new deadlock state.

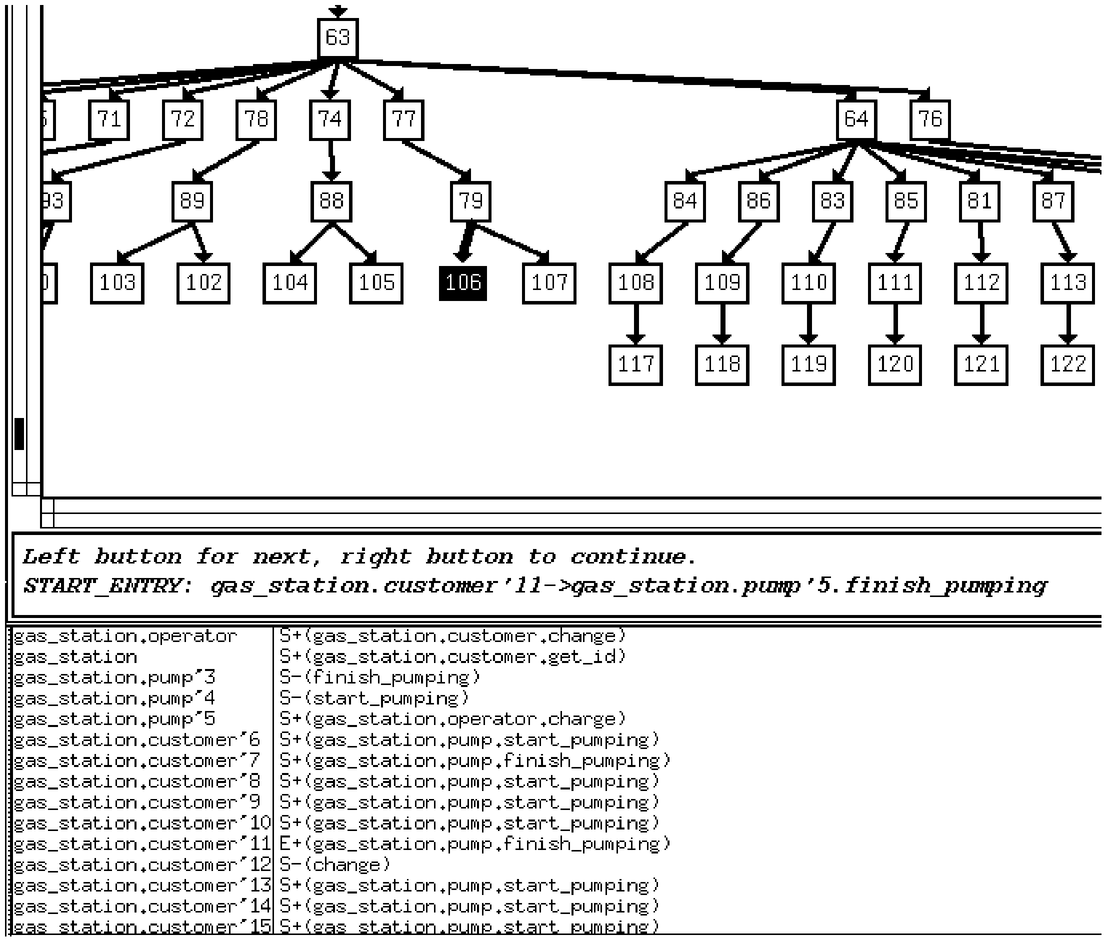
The program with these patches applied is shown in Appendix B

The incorrect diagnoses suggested by Helmbold and McDowell are reasonable, even given the output of our system. The underlying fault in this system is that the OPERATOR maintains records of which CUSTOMER task should receive change in FIFO order according to the order of payment, but several CUSTOMER tasks may PRE_PAY, and then arrive at the pumps in a different order than they arrived at the OPERATOR. This is an example of fixing the symptom (a deadlock) rather than the cause (the fault involving an incorrect assumption about FIFO order.)

Our suggestion for detecting when a suggested "fix" is incorrect is to utilize the partial SCA facilities provided in our tool to check corrections. The user would make corrections to the software, then build the portion of the TICG that represents the state space around the original failure. In this way we can detect some potential failures that occur which are related to the original failure.

In this case we chose to start building forward beginning at a state that equivalent to the one that was was ten states back from the original failed state. From this state we built forward for six levels, building 471 nodes. We then ran the deadlock checker on this partial TICG. It reported a new deadlock node, giving its node number. We then used the Node State Window to find the details of this state. We see that indeed this is a feasible deadlock state, so even after all our attempted fixes, we have missed something. The system output is shown in Figure 4.7.

This report of a feasible deadlock state is extremely valuable, because the failures that occur after the initial set of incorrect fixes are quite infrequent,



```
Left button for next, right button to continue.
START_ENTRY: gas_station.customer'11->gas_station.pump'5.finish_pumping
```

```
gas_station.operator       S+(gas_station.customer.change)
gas_station                S+(gas_station.customer.get_id)
gas_station.pump'3         S-(finish_pumping)
gas_station.pump'4         S-(start_pumping)
gas_station.pump'5         S+(gas_station.operator.charge)
gas_station.customer'6     S+(gas_station.pump.start_pumping)
gas_station.customer'7     S+(gas_station.pump.finish_pumping)
gas_station.customer'8     S+(gas_station.pump.start_pumping)
gas_station.customer'9     S+(gas_station.pump.start_pumping)
gas_station.customer'10    S+(gas_station.pump.start_pumping)
gas_station.customer'11    E+(gas_station.pump.finish_pumping)
gas_station.customer'12    S-(change)
gas_station.customer'13    S+(gas_station.pump.start_pumping)
gas_station.customer'14    S+(gas_station.pump.start_pumping)
gas_station.customer'15    S+(gas_station.pump.start_pumping)
```

Figure 4.7: System output shown partial TICG built by SCA for patched gas station.

so testing may give us a false confidence that our changes have eliminated the problem.

This shows how partial SCA can be combined with debugging. We couldn't have use complete SCA because of the following characteristics of this example:

- Tasks are dynamically created. It is impossible to statically determine how many tasks (in this case CUSTOMERS will be created until the program is run.) Conventional SCA relies on being able to determine this statically.
- The full state space of this program has far too many states to build the complete TICG. Here we limit exploration, not by task subsetting, but by manually choosing to explore only a portion of the execution.

Limitations of current implementation Our current implementation requires additional user intervention to handle the scenario discussed above. When you change a program you must regenerate the TIGs. This process changes both the Statement_IDs and TIG node numbers. This means that both the partial TICG we built no longer have an accurate mapping back to the TIGs and source code. Currently to update this mapping the engineer must rebuild the TICG manually using incremental generation of states until the partial TICG is reconstructed. There is no theoretical reason why the mappings couldn't be automatically updated, but this is not supported by our current implementation.

4.3 Debugging Data Races

The next example illustrates a data race problem similar to those observed in running scientific simulations on multiprocessors. A common application design approach is divide and conquer. Different parts of the application problem are mapped onto different processing elements (in this case tasks rather than physical processors). Each task does some calculation to derive its value. The calculations take varying amounts of time, and as a task finishes it compares its value with the best value calculated so far. If its value is the best so far it saves it as the best value, then goes on to do other work. Once all subparts are complete, the best value is returned.

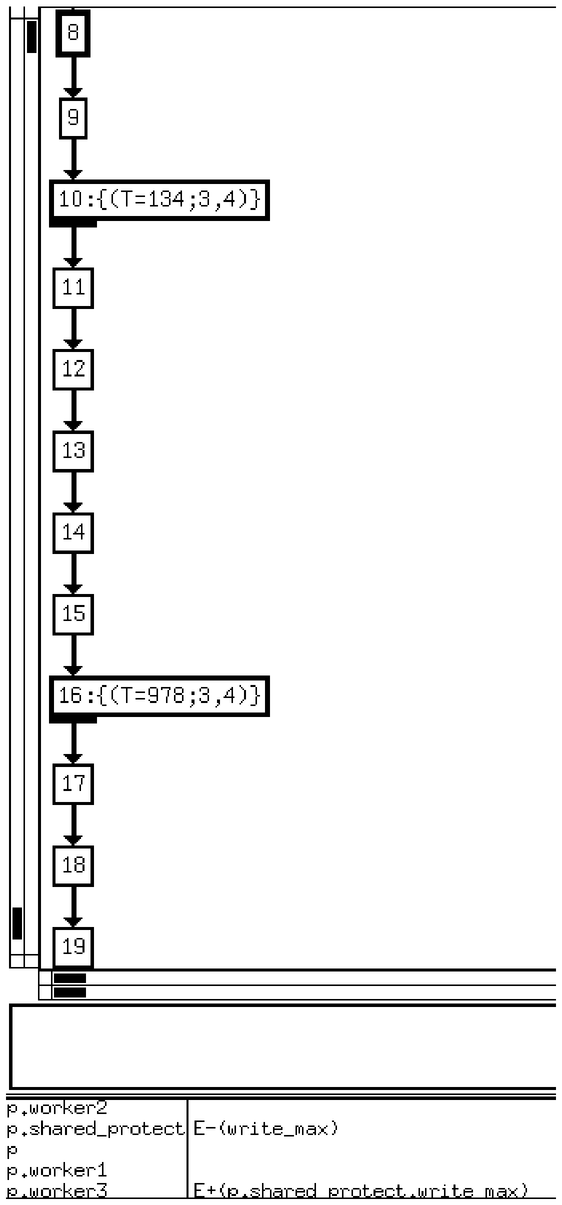
Here we have an improper locking strategy. The individual reads and writes are correctly locked, but in the interval between when one task decides it has the best value, and commits it, it is possible for another task to save a better value, which is then overwritten.

The failure is revealed when the value of the *max* variable is printed out. In this case we have the tracing system instrument both read and write accesses to the *max* variable, which occur in the accept statements for *read_max* and *write_max*. The tracing system then records the concurrency state each time the variable is read or written.

When the user encounters an execution that fails (an incorrect value is printed out for max) SADCPU is started with the trace file generated by the failed execution. A TICG constructed from the trace of a failed execution is shown in Figure 4.9. The states where the variable was read or written are highlighted

```
procedure p is
   NUM_TASKS : constant integer := 3;
   workers : array (1..NUM_TASKS) of worker;
task body shared_protect is
   max : integer :=0;
   num_done : integer := 0;
begin
   loop
         select
            accept read_max( i : out integer) do
                  i := max;
            end read_max;
      or
            accept write_max( i : in integer) do
                  max := i;
            end write_max;
         or
            accept done do
                  num_done := num_done + 1;
            end done;
         end select;
         exit when num_done = NUM_TASKS;
   end loop;
  text_io.put_line("best overall " & integer'image(max));
end Shared_Protect;

task body Worker is
   Best_So_Far, My_Value : integer;
begin
   My_Value := Calc_Local_Value;
   shared_protect.read_max( Best_So_Far );
   if (My_Value > Best_So_Far) then
         shared_protect.write_max(My_Value);
   end if;
   Shared_Protect.Done;
end Worker;

begin -- p
   null;
end p;
```

Figure 4.8: Code for Data Race Example



```
p.worker2
p.shared_protect E-(write_max)
p
p.worker1
p.worker3        E+(p.shared_protect.write_max)
```

Figure 4.9: System output for failed execution of shared data access example.

(they have wider borders than other states.) Clicking on these states shows two pieces of information:

- a statement ID. This tells us which statement made the access, and which variable is involved. There is a separate ID for the PRE and POST access event (see the discussion in section 3.2.)
- a time stamp. The time stamp is a monotonically increasing record of the time an event occurred. It is needed because the same TICG can be visited more than once in an execution. Indeed, as we saw in the elevator example this is often the case.

In the system output showing in Figure 4.9 we have clicked on states 10 and 16, but not state 8. In this case we could have shown the data access information for all states on the screen at once, but often this is not possible because the list of data accesses for a particular state can cause it to overlap a large portion of the graph. In the output shown the TICG is linear, which removes this problem.

Remember from the discussion in section 3.2 that a particular variable access may be associated with more than one TICG state. In this example the PRE and POST event for each access occurred in the same TICG state, which is the most common case.

Examining this TICG reveals a path with the following sequence:

worker'1 ⇒ shared_protect.read_max

worker'3 ⇒ shared_protect.read_max

worker'3 ⇒ shared_protect.write_max

worker'1 ⇒ shared_protect.write_max

This ordering is the source of our incorrect result. Worker'1 read the current value, and decided his new value was best. Worker'3 made a similar decision about his new value. Worker'3 wrote back his value, which was really the best value, and worker'1 then overwrote this with his value.

In this case it would have been helpful to have the value that was read or written stored as part of the access information. This isn't currently part of our system, but would be a valuable addition. One problem to overcome is the difficulty of recording the value of variables of types more complex than integers. For example, a data structure containing pointers would be difficult to trace. Integration with a full debugger could make this more meaningful.

Chapter 5

System Architecture

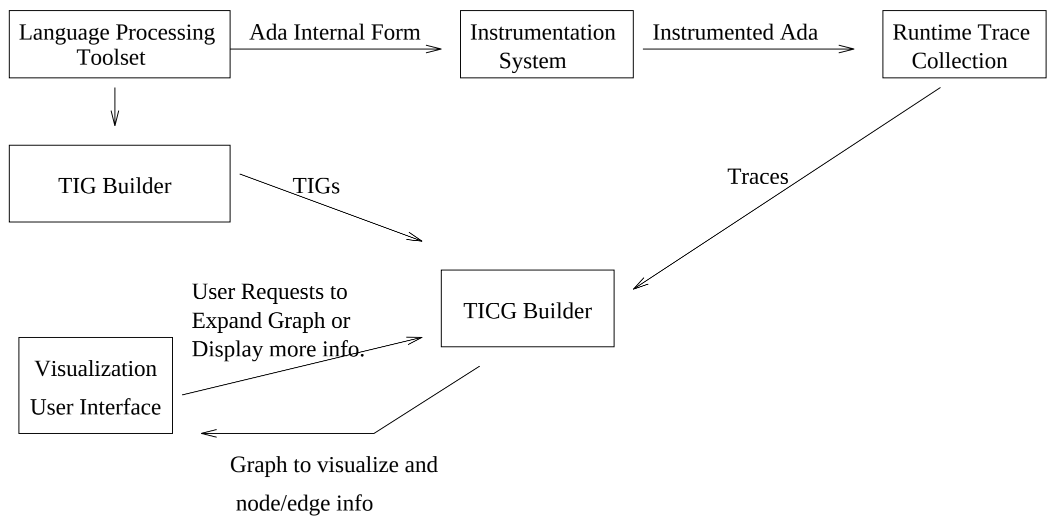

Language Processing Toolset
Ada Internal Form
Instrumentation System
Instrumented Ada
Runtime Trace Collection
TIG Builder
TIGs
Traces
User Requests to Expand Graph or Display more info.
TICG Builder
Visualization User Interface
Graph to visualize and node/edge info

Figure 5.1: System Architecture

We currently have a prototype implementation that supports the style of debugging by TICG examination that we described above. The system runs on Sun SparcStations, and is written in a combination of Ada and C.

Preparation phase Before the program under test is executed several actions must be taken to instrument the program and build necessary artifacts. All of these tools are written in Ada.

The language processing toolset transforms Ada program code into an internal representation. This semantically analyzed internal form is used by as a common intermediate form by the other components of the system. The TIG Builder uses this semantically analyzed representation to construct the TIGs.

The internal form is also used by the instrumentation system to identify where tasking interactions occur. At statements where a task interaction occurs statements calling the monitoring system are added by pretty-printing the intermediate form back into Ada code, which is then compiled.

The runtime trace collection system has two parts: the calls to the monitoring system added by the instrumentation system, and the monitoring system which is implemented as an Ada package. It collects "events" which have several components, the event type (START_TASK, RENDEZVOUS_ENTRY, etc.) the DTID, and a Statement_ID. This Statement_ID is derived from the internal form, and this use of a common internal form makes it easy to map from the collected traces back to the TIG nodes they represent. All of this information is logged to a trace file. Each of these IDs are mapped back to small integers, making the traces fairly small, on the order of 10 bytes per event. This allows us to record 100_000 events in a one megabyte trace. Because we are tracing task interactions, which occur fairly infrequently in most Ada programs, we can completely trace many long running examples. The limited size and number of events may also allow it to be adapted to debugging in a host-target environment. The target execution monitor could either pass the events back to the host in real time, or retain a circular buffer of the most recent events in a portion of system memory, to be unloaded to the host development system after an error occurs.

Post-execution program understanding After an execution has failed the engineer uses the other two components of our system, the TICG builder and the Visualization User Interface, to examine the execution and attempt to understand the execution of the failed program under test.

The TICG builder is the core component of the entire system. It is responsible for generating the states of the TICG. It does this both by translating the trace information into the corresponding TICG states, and by creating additional states under user control using the normal SCA concurrency state generation algorithm described above. The TICG builder is also written in Ada.

The Visualization User Interface is written in C, and depicts the TICG graphically on an X Windows System display. The user interface and TICG builder components exist in separate Unix processes, and communicate using the Q [MHO96] multilanguage interprocess communication system to manage distribution and interlanguage data typing issues. The Visualization User Interface was built by modifying the VCG tool [San94]. Building upon this tool accrues several benefits, most notably a very efficient layout of the TICG, with several hundred nodes plotted in a few seconds, the ability to switch to alternate views of the graph (such as a fisheye view) and the ability to scale the graph display.

The examples from the dissertation and the source code for SADCPU are available via the World Wide Web at `http://www.ics.uci.edu/~self/sadcpu`

Chapter 6

Conclusion

This paper has presented a debugging methodology designed around a framework for representing concurrent program behavior, and a set of techniques for exploring the state space it represents. The advantages of our approach include

- The use of the TICG to provide a formal model of concurrency state space that the engineer can use to understand program behavior
- Providing useful information even in cases where complete SCA is infeasible due to the large state space.
- It is capable of handling dynamic features frequently not modeled by SCA including dynamic task creation, delay statements, conditional and timed entry calls.
- The integration of trace based debugging with SCA, using SCA to further explore an execution.
- Exploring the state space by "backwards" state generation.
- An algorithm for task subsetting, enabling the modeling of a portion of a concurrent system.

- Incorporating data access information into the TICG, allowing diagnosis of missing synchronization which causes data races.

We have shown by example that our methodology can provide useful information in many cases, but there are several limitations which restrict the usefulness of our system:

- **High entry barrier** Effective use of the system requires a significant familiarity with static concurrency analysis. Studies of the users of parallel debugging tools have shown that programmers are very reluctant to begin using tools which require learning many new concepts before they produce results.[Pan93b] Because our methodology is based on a formal model, and understanding that model is crucial to interpreting the tool output, completely removing this barrier seems impossible. One approach we may pursue involves hiding the graphical TICG representation from the novice user, and instead presenting the potential paths as sequences of code statements.
- **Scalability** The current user interface is capable of presenting a portion of the graph of up to several hundred nodes to the user. The user can then scroll to see more nodes as needed. This is usually sufficient to visualize the portion of the TICG that is of interest, because the user may choose what portions of the TICG are displayed at any time. However, as the number of tasks grows very large using the methodology will become difficult. If the program consists of several hundred tasks the size of each state will become too large to understand. Task subsetting can aid greatly in reducing this problem, but some systems will exceed the capabilities of our system. Our system expands the portion of the state space that can be explored well

beyond the capacity of an unaided programmer, and the TICG provides the engineer a model to integrate pieces of information gathered during debugging.

- **Susceptibility to user error** Our technique relies heavily on a skilled user's knowledge of the system under test. There are several cases where our system relies on the user to supply information about the program under test that it can't independently verify:

  - identification of "interesting" tasks
  - pruning infeasible paths
  - deciding which portions of the graph to explore further

  If the user makes incorrect choices for these items then the TICG may not contain the states and paths that describe a fault. This can be problematic as we have argued that faults exist because the user has an incomplete or incorrect mental model of program behavior. However, this seems a necessary limitation since generating the full graph is infeasible. If the user finds that the faulty state doesn't appear then it is always possible to go back and generate the missing states.

Future Enhancements There are several enhancements that will make our system more useful:

- **integration with a full debugger** INSTANT REPLAY [LMC87] style debugging support would allow the user to more fully examine faults which aren't entirely due to concurrency structure.

- **extensions for testing** At this time the system allows for dynamic creation of states only during program tracing. The TICG model could also be used for concurrent program testing by having the user identify interesting portions of the TICG, then controlling the Ada scheduler so that any non-deterministic choices would be forced along these paths.
- **program slicing** The data race understanding possible with our current system is fairly limited because the user must identify which variable to trace. A significant aid would be to use dynamic program slicing [ADS91] which would allow us to identify which statements in the program affect the value of an output variable.

Bibliography

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 60–73, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.

[Bat88] Peter Bates. Debugging heterogeneous distributed system using event-based models of behavior. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22. ACM Press, May 1988.

[BG91] David F. Bacon and Steph Copen Goldstein. Hardware-assisted replay of multiprocessor. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[BLP91] Robert Benner, Lyle Long, and Richard Pelz. Panel discussion on debugging large scale scientific applications. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 12–14. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[Bru91] Bernd Bruegge. A portable platform for distributed event environments. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 184–193. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[CM91] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[CMN91] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analyusis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 91.

[CS91] Jong-Deok Choi and Janice M. Stone. Balancing runtime and replay costs in a trace-and-replay system. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 26–35. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[DS91] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[Els88] I. J. P. Elshoff. A distributed debugger for amoeba. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 1–10. ACM Press, May 1988.

[FB88] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123. ACM Press, May 1988.

[Fid91] Colin Fidge. Logical time in distributed systems. *IEEE Computer*, 24(8):28–33, August 1991.

[FLMC88] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173. ACM Press, May 1988.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software — Practice & Experience*, 16(3):225–233, March 1986.

[GGLS91] Arthur P. Goldberg, Ajei Gopal, Andy Lowry, and Rob Strom. Restoring consistent global states of distributed computations. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 144–154. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[GL83] Virgil D. Gligor and Gary L. Luckenbaugh. An assesment of the real-time requirements for programming environments and languages. In *Proceedings of the Real-Time Systems Symposium*, pages 3–19, December 1983.

[Gor91] Michael M. Gorlick. The flight recorder: An architectural aid for system monitoring. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 175–183. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

[HC88] Alfred A. Hough and Janice E. Cuny. Initial experiences with a pattern-oriented parallel debugger. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 195–205. ACM Press, May 1988.

[HK88] Wenwey Hseush and Gail E. Kaiser. Data path debugging: Data-oriented debugging for a concurrent programming language. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 236–247. ACM Press, May 1988.

[HL85] David Helmbold and David Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[KZ93] Doug Kimelman and Dror Zernik. On-the-fly topological soft – a basis for interactive debugging and live visualization of parallel programs. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages

12–20. ACM Press, December 1993. Appeared as *ACM SIGPLAN Notices 28*(12).

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 44–52, Pittsburgh, PA, May 1989.

[Lev93] David L. Levine. *Structural Reengineering for Static Concurrency Analysis*. PhD thesis, University of California, Irvine, 1993. Available as UCI ICS technical report 93–21.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[MC88] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 141–150. ACM Press, May 1988.

[MHO96] Mark J. Maybee, Dennis H. Heimbigner, and Leon J. Osterweil. Multilanguage interoperability in distributed systems: Experience report. In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996. Also issued as CU Technical Report CU-CS-782-95.

[Net93] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11. ACM Press, December 1993. Appeared as *ACM SIGPLAN Notices 28*(12).

[Pan93a] Cherri M. Pancake. Customizable portrayals of program structure. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 64–74. ACM Press, December 1993. Appeared as *ACM SIGPLAN Notices 28*(12).

[Pan93b] Cherri M. Pancake. Why is there such a mis-match between user needs and tool products? In *1993 Workshop on Parallel Computing Systems*, March 1993.

[PL88] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129. ACM Press, May 1988.

[PTY95] Mauro Pezzè, Richard N. Taylor, and Michal Young. Graph models for reachability analysis of concurrent programs. *ACM Transactions on Software Engineering and Methodology*, 4(2):171–213, April 1995.

[San94] Georg Sander. Graph layout through the vcg tool. In *Graph Drawing, DIMACS International Workshop, GD '94*, pages 194–205, Princeton, New Jersey, October 1994. *LNCS 894*.

[Sto88] Janice M. Stone. A graphical representation of concurrent processes. In Thomas J. LeBlanc and Barton P. Miller, editors, *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 226–235. ACM Press, May 1988.

[Tay83] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[You89] Michal Young. *Hybrid Analysis Techniques for Software Fault Detection.* PhD thesis, University of California, Irvine, 1989. Available as UCI ICS technical report 89–26.

[YTL$^+$95] Michal Young, Richard N. Taylor, David L. Levine, Kari A. Nies, and Debra Brodbeck. A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, January 1995.

[ZR91] Dror Zernik and Larry Rudolph. Animating work and time for debugging parallel programs foundation and experience. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 46–56. ACM Press, May 1991. Appeared as *ACM SIGPLAN Notices 26*(12).

Appendix A

Algorithms used in constructing TICGs

A.1 Converting a trace to an equivalent TICG

Algorithmic Complexity: O(n) where n is the number of trace events.

```
function Build_TICG_From_Trace ( Trace_Name : in string )
                                        return task_interaction_concurrency_graph is
begin
  -- create a TICG that has only the initial state
  Trace_TICG := New_TICG(Trace_TICG, Generic_TIG);
  Initial_State := Create_Initial_State(Trace_TICG);

  -- Find_Or_Create_State checks to see if this state
  -- already exists.  If so, it sets New_State to false
  -- Parent_Key is the state number of either the created
  -- state or previously existing state.

  Find_Or_Create_State(Trace_TICG, Initial_State,
                                Parent_Key, New_State);

  AR.event_io.open(log, AR.event_io.in_file, Trace_Name);

  Parent := Initial_State;

  -- now loop through the trace file, adding states to represent
  -- the task interactions seen in the trace.

  while not AR.event_io.end_of_file(log) loop
    AR.event_io.read(log, event);
```

```
case Event.event is
   when AR.Start_Task =>
     -- Establish the mapping between the DTID and STID
     ART_IRIS.Lookup_Task_Name( Event.Task_Body,
                                Event.Current_Task,
                                        Trace_TICG.Model_Info.all);
   when AR.Rendezvous_Entry =>
     -- this is the start of a rendezvous

     -- The new child state is a copy of the parent state
     Child.States   := Kernel(Parent);

     -- and has no edges leaving it (yet.)
     Child.Edges    := Empty_Edges;

     -- figure out which tasks were involved

     -- calling task may be the main program, which Ada assigns the
     -- semantics of a task.  The main task is always given DTID
     -- number 1

     if integer(Event.Other_Task_ID) = 1 then -- 1 is main task dynamic ID
       Calling_Task_Key := tif.tig_key(Parameters.Main_Task_Tig_Key);
     else
       Calling_Task_Key :=
         ART_IRIS.Task_ID_to_Tig_Key(Event.Other_Task_ID);
     end if;

     -- we don't have to do the check here because the main task
     -- isn't allowed to have 'accept' statments
     Accept_Task_Key := ART_IRIS.Task_ID_to_Tig_Key(Event.Current_Task);

     -- find out what TIG edges were involved by looking in a table
     -- that maps Statement_IDs to edges.  This association was
     -- established when we built our TIG internal form.

     Calling_Edge_Key := Lookup_Edge_Key(Event.Entry_Stmt,
                                  Trace_TICG.Model_Info.all,
                                             Calling_Task_Key);
     Accept_Edge_Key := Lookup_Edge_Key(Event.Acpt,
                                  Trace_TICG.Model_Info.all,
                                             Accept_Task_Key);

     -- now we need to find out what TIG nodes are at the end of
     -- these edges.  In the Child state these nodes will be replace
     -- the nodes that were in the parent.        We stored this information
     -- when we built the TIGs internal form

     Calling_Head_Node := TIF.TIF_edge_to_origin_table(Calling_Edge_Key);
     Accept_Head_Node := TIF.TIF_edge_to_origin_table(Accept_Edge_Key);
```

```
    Child.States(tig_key(Calling_Task_Key)) :=
                                    tig_state_key(Calling_Head_Node);
    Child.States(tig_key(Accept_Task_Key)) :=
                                    tig_state_key(Accept_Head_Node);
    Child.Edges(tig_key(Calling_Task_Key)) :=
                          tig_edge_key(Calling_Edge_Key);
    Child.Edges(tig_key(Accept_Task_Key)) :=
                          tig_edge_key(Accept_Edge_Key);

    -- as mentioned before, it this state already exists, then
    -- Find_Or_Create_State will avoid creating a duplicate.  It
    -- does this by maintaining a hash table of states.

    Find_Or_Create_State(Trace_TICG, Child, Child_Key, New_State);

-- Insert into table of traced states.  This is used when we
    -- animate the trace, so we know which TICG edge corresponds to
    -- a particular step of the trace.
trace_step_to_adg_table(curr_trace_step) := Child_Key;
curr_trace_step := curr_trace_step + 1;

    -- Create the link between the Parent and Child TICG states.
    -- This state may already exist if the trace visited this state
    -- previously, and ended up taking the same transition out.
    -- This couldn't happen when doing static expansion, because
    -- in that case we build all out edges in one step, and never
    -- revisit an existing state to do further expansion.

    if not New_State then
    -- if it isn't we need to see the edge to child state already exists
      Find_Or_Create_Link(Trace_TICG, Parent_Key, Child_Key, Child);
    else
      Create_Edge(Trace_TICG, Parent_Key, Child_Key, Child);
    end if;

    Parent := Child; -- new Parent is our old Child
    Parent_Key := Child_Key;

  when AR.End_Rendezvous =>
    -- This is the end of a rendezvous, we already saw the start.
-- The code is the same as for AR.Rendezvous_Entry
  when AR.Start_Subprog =>
    -- this is the 'or delay' statement expiring, or an 'or else' part
    -- of a conditional entry call.
    Advance_Past_Other_Tasking(Event.Subpgm);
  when AR.Execute_Statement =>
    -- This is a variable access (read or write) of a variable
    -- we are tracing.  Assign_Access_Event associates this access
    -- event with the current Parent state.
    --
    -- We actually see two of these events for each traced variable
```

```
            -- access.  Most of the time they will end up being annoatations
            -- of the same TICG state, but this won't be the case if other
            -- tasks have interactions between the pre- and post- of this
            -- variable access.
            Assign_Access_Event(Parent, Event.Statement);
          when AR.Exception_Task =>
            -- a task just aborted.  We don't handle this very well right
            -- now.  We just make the task as terminated, and whenever we
            -- try to do static generation of states beyond this we
            -- ignore trying to advance the terminated task.
      when others =>
            null;
    end case;
  end loop;
  -- when we save the TICG we output in a format suitable for the
  -- visualization front end
  save_ticg(Trace_TICG);
  AR.event_io.close(log);
  return Trace_TICG;
end Build_TICG_From_Trace;
```

A.2 Building internal representation for TIGs

Algorithmic Complexity: The complexity of converting each TIG is $O(N \cdot EG \cdot EPG \cdot (EG^2 \cdot EPT))$ where

- N is the number of nodes in the TIG
- EG is the number of edge groups in the TIG
- EPG is the average number of edges per edge group
- EPT is the average number of edges per TIG

```
Procedure Retrieve_TIGs(Internal_Structure: OUT Internal_Tig_Structure) is

 -- The TIG graph structure is represented internally as two arrays.
 -- The node array contains an integer that is the index of the first
 -- child edge of that node.  Each child edge has an index to the next
 -- child edge of its parent.
 --
 -- There is one node array and one edge array per program, rather one
 -- per TIG.  We separately keep track of which nodes are the roots of
 -- a TIG.

 -- The size of the arrays is fixed, to allow rapid access.  We assume
 -- that there will be an average of no more than 50 nodes per TIG.  An
 -- individual TIG can have more than 50 nodes, but the average number of
 -- nodes per TIG must be less than this, or we must bump up this constant
 -- and recompile.  Similarly the average number of edges is limited to
 -- 100 edges per TIG.  In practice we have never seen these limits
 -- exceeded.

 Max_Nodes: CONSTANT adgs.adg_node_key  := adgs.adg_node_key(50 * Num_Tigs);
 Max_Edges: CONSTANT adgs.adg_edge_key := adgs.adg_edge_key(100 * Num_Tigs);

 type tig_node_record is record
   real_tig_node : TIG.TIG_Node; -- external representation of this node
   real_tig_key : TIG_Key; -- index that is the internal rep
   hashval : integer; -- hashing value used for quick comparisons

   -- what nodes do the edges of this node lead to?
   link_exists : link_exists_array := (others => false);
  end record;

  type Unique_TIG_Node_Table is array( tig_key range <> ) of tig_node_record;
  type Unique_TIG_Node_Map_PTR is access Unique_TIG_Node_Table;
  type Unique_TIG_Node_Maps is array(First_Tig_Key..Last_Tig_Key ) of
                                            Unique_TIG_Node_Map_PTR;

  Unique_TIG_Node_Map : Unique_TIG_Node_Maps;
```

```
begin
 PTR:= new Tig_Structure_Body(Max_Nodes, Max_Edges, Max_Entries,
                                                   Max_Groups);
 adgs.Init (PTR.Graph, Min_Node => 1, Max_Node => PTR.Max_Nodes,
            Min_Edge => 1, Max_Edge => PTR.Max_Edges);
 Internal_Structure := PTR;

 -- Place the first node in each TIG on stack of nodes to
 -- be processed, while building map from TIG names to TIG keys
 Roots: declare

  -- used to compare TIG edge lists in node operation

  -- Calc_TIG_Node_Hashval builds a hash value reflecting the number
  -- of Edge Groups and the total number of out edges
  -- this isn't unique, but is quick and easy to calculate, and
  -- will rule out a lot of nodes
  function Calc_TIG_Node_Hashval( TN : TIG.TIG_Node ) return integer is
     EG              : TIG.TIG_Edge_Group;
     EG_Set          : TIG.TIG_Edge_Group_List;
     EG_Edge_Set     : TIG.TIG_Edge_List;
     Eg_Set_Size,
     EG_Edge_Set_Size : Natural;
     Total_Edges     : Natural;
  begin
     EG_Set_Size := Number of exiting edge groups;

     Total_Edges := 0;
     for each Edge Group EG loop
       EG_Edge_Set_Size := Number of Edges in EG;
       Total_Edges := Total_Edges + EG_Edge_Set_Size;
     end loop;
     return Total_Edges * 100 + EG_Set_Size;
  end Calc_TIG_Node_Hashval;

  procedure Hash_TIG_Node( TN : in out TIG.TIG_Node ) is
  begin
     Unique_TIG_Node_Map(i)(tig_key(TIG.Node_Number(TN))).hashval :=
                                          Calc_TIG_Node_Hashval(TN);
     Unique_TIG_Node_Map(i)(tig_key(
                              TIG.Node_Number(TN))).real_tig_node := TN;

  end Hash_TIG_Node;

  procedure Build_TIG_Node_Hash(Graph : in out TIG.TIG_Graph) is
  begin
     for each node N in Graph loop
       Hash_TIG_Node(N);
     end loop;
  end Build_TIG_Node_Hash;
```

```
-- Two edge groups are equal if they have the same edges.
--
-- This check is made more difficult because the edges within
-- an edge group won't necessarily appear in the same order.
-- Thus we have to use two nested loops to check to see
-- if every edge and edge group in EG1 is present in EG2.

function equal_edge_group(EG1, EG2 : TIG.TIG_Edge_Group)
                                                   return boolean is
   N1_EL, N2_EL : TIG.TIG_Edge_List;
   N1_Edge, N2_Edge : TIG.TIG_Edge;
   N1_E_Set_Size, N2_E_Set_Size,
   N1_EL_size, N2_EL_size : integer;
   found_matching_edge: boolean;
   N1_Label, N2_Label : DynamicStrings.DynamicString;
begin
   N1_EL := TIG.Edges(EG1);
   N1_EL_size := TIG.Length(N1_EL);
   N2_EL := TIG.Edges(EG2);
   N2_EL_size := TIG.Length(N2_EL);

   for j in 1..N1_EL_size loop
    N1_Edge := TIG.Retrieve(N1_EL, j);
    N1_Label := TIG.Get_Label(N1_Edge);
    found_matching_edge := false;
    for k in 1..N2_EL_size loop
      N2_Edge := TIG.Retrieve(N2_EL, k);
      N2_Label := TIG.Get_Label(N2_Edge);
      if (TIG.Node_Number(TIG.Tail(N1_Edge)) =
          TIG.Node_Number(TIG.Tail(N2_Edge))) then
        found_matching_edge := true;
        exit;
      end if;
    end loop;

    if not found_matching_edge then
      return false;
    end if;
   end loop;
   return true; -- if we fall through to here then we matched each edge
end equal_edge_group;

-- Two nodes are equal if there edge groups are equal.
--
-- Just like checking individual edge groups for equivalence,
-- This check is made more difficult because the edges groups
-- won't necessarily appear in the same order in both nodes.
-- Thus we have to use two nested loops to check to see
-- if every edge and edge group in EG1 is present in EG2.
--
```

```
-- This means checking two nodes for equivalence is an order
-- O(G^2*E^2) operation where G is then number of edge Group and
-- E is the number of Edges.  This is why we build the hash table
-- to avoid doing this calculation for nodes that are obviously
-- different

function equal_nodes(N1, N2: TIG.TIG_Node) return boolean is
   N1_EG, N2_EG : TIG.TIG_Edge_Group;
   N1_EG_Set, N2_EG_Set : TIG.TIG_Edge_Group_List;
   N1_EG_Edge_Set, N2_EG_Edge_Set : TIG.TIG_Edge_List;
   N1_EG_Set_Size, N2_EG_Set_Size,
   N1_EG_Edge_Set_Size, N2_EG_Edge_Set_Size  : Natural;
   found_matching_edge_group : boolean;
begin
   N1_EG_set := TIG.Exits(N1);
   if (N1_EG_set /= TIG.Null_TIG_Edge_Group_List) then
     N1_EG_set_size := TIG.Length(N1_EG_Set);
   else
     N1_EG_set_size := 0;
   end if;

   N2_EG_set := TIG.Exits(N2);
   if (N2_EG_set /= TIG.Null_TIG_Edge_Group_List) then
     N2_EG_set_size := TIG.Length(N2_EG_Set);
   else
     N2_EG_set_size := 0;
   end if;

   for j in 1..N1_EG_Set_Size loop
     N1_EG := TIG.Retrieve(N1_EG_Set, j);
     found_matching_edge_group := false;
     for k in 1..N1_EG_Set_Size loop
       N2_EG := TIG.Retrieve(N2_EG_Set, k);
       if (equal_edge_group(N1_EG, N2_EG)) then
         found_matching_edge_group := true;
         exit;
       end if;
     end loop;
     if not found_matching_edge_group then
       return false;
     end if;
   end loop;
   return true; -- if we fall through to all we matched each EG
end equal_nodes;

procedure TIG_node_present( N : TIG.TIG_Node;
                        is_new_node: out boolean;
                                  old_node_num : out tig_key) is
   curr_hashval : integer;
begin
   is_new_node := true;
```

```
      curr_hashval := Calc_TIG_Node_Hashval(N);
      for curr_hash_node in Lowest_TIG_Index_Value..
                        tig_key(TIG.Get_Node_Count(Instance_Graph)) loop
        if (Unique_TIG_Node_Map(i)(curr_hash_node).hashval =
                                                    curr_hashval) then
          if (equal_nodes(N,
                  Unique_TIG_Node_Map(i)(curr_hash_node).real_tig_node)) then
            is_new_node := false;
            old_node_num := curr_hash_node;
            exit; -- want to return index of FIRST match
          end if;
        end if;
      end loop;
  end TIG_node_present;

  procedure Insert_Unique_TIG_Nodes ( N : in out TIG.TIG_Node ) is
      is_new_node : boolean;
  begin
      TIG_node_present( N, is_new_node, old_node_num );
      if is_new_node then
        Unique_TIG_Node_Map(i)(tig_key(TIG.Node_Number(N))).real_tig_key :=
                                         tig_key(TIG.Node_Number(N));
      else
        Unique_TIG_Node_Map(i) (tig_key(TIG.Node_Number(N))).real_tig_key :=
                                                      old_node_num;
      end if;
  end Insert_Unique_TIG_Nodes;

  procedure Build_Unique_TIG_Node_Table is new
      TIG.Iterate_Over_TIG_Nodes( Insert_Unique_TIG_Nodes );

  procedure Print_Unique_TIG_Node_Table(Instance_Graph :TIG.TIG_Graph;
                                  tname : string;
                                  index : tig_key) is
  begin
      text_io.put_line("uniq table for TIG " & tname);
      for k in Unique_TIG_Node_Map(index).all'range loop
        text_io.put_line(tig_key'image(k) &
          tig_key'image(Unique_TIG_Node_Map(index)(k).real_tig_key));
      end loop;
  end Print_Unique_TIG_Node_Table;



begin
  i := First_Tig_Key;
  while natural(i) <= Parameters.Num_Tasks loop
      Instance_Graph := Parameters.Dynamic_Task_Map(integer(i));
      Tig_Name := Parameters.Repos_TIGS_Info_Table(
        Parameters.Dynamic_Task_To_Static_TIG_Map(integer(i))).Tig_Task_Name;
        -- We're using the optimized representation of TIGs discussed in
        -- the CATS TOSEM Rationale paper.  This relaxes the "single entrance"
        -- property of the original TIG definition, and makes graphs smaller.
        --
        -- To do this we must determine which nodes are identical according
        -- to this defition.
        --
        -- Iterate over TIG looking for duplicate nodes (ones where the
        -- exits are the same, although the entrances are different.)
        -- We build up the Unique_TIG_Node_Table this way.  Entries there
        -- are the node itself for the first one, the original one for
        -- any duplicates.

        -- generate hash values for all nodes in this graph.
        Build_TIG_Node_Hash(Instance_graph);

        -- now figure out which nodes are unique.
        Build_Unique_TIG_Node_Table( Instance_Graph );
        PTR.Task_Map(i) := Instance_Graph;

        PTR.Task_Names(i) := Tig_Name;
        Tig_Initial_Node := TIG.Start(PTR.Task_Map(i));
        adgs.Create_Node (PTR.Graph, Tig_Initial_Node_Key);

        Graphite_To_TIG_Node_Keys.Insert(TIG.Node_Number(Tig_Initial_Node),
                                 Tig_Initial_Node_Key, Node_Table(i));
        PTR.Node_Map(Tig_Initial_Node_Key) := Tig_Initial_Node;
        PTR.Node_ID (Tig_Initial_Node_Key) := TIG_Initial_Node_Key;
        Node_Stacks.Push(To_Be_Processed, (TIG_Initial_Node_Key, i));
        i := TIG_Key'SUCC(i);
    end loop;
  end Roots;

  -- At this point, states First_Tig_Key..Last_Tig_Key in the
  -- attributable_directed graph structure are the initial nodes of all
  -- the TIGs, i.e., they together form the initial state of the task
  -- interaction concurrency graph.

  -- The algorithm is to start with the initial node of the TIG, push
  -- all its children onto the To_Be_Processed Node_Stack.
  -- We then pop the top entry off the stack, and push all its children
  -- onto the stack.  We continue this process of pushing children on
  -- the stack until the stack is empty (meaning that all nodes have been
  -- processed

  while not Node_Stacks.Is_Empty(To_Be_Processed) loop
    Node_Stacks.Pop(To_Be_Processed, Node_Info);
    Task_Number := Node_Info.Task_Number;
    Parent_Key  := Node_Info.Node_Number;
    Parent_Node := PTR.Node_Map(Parent_Key);
    EG_Set := TIG.Exits(Parent_Node);
    -- loop over each exit group
    for The_Edge_Group in 1..TIG.Length(EG_Set) loop

          EG := TIG.Retrieve(EG_Set, The_Edge_Group);
          EG_Tail_Set := TIG.Tails(EG);

      -- loop over each exit node for the edge group
          for i in 1..TIG.Length(EG_Tail_Set) loop
            Child_Node := TIG.Retrieve(EG_Tail_Set, i);
            The_Edge := TIG.Retrieve(TIG.Enter(Child_Node),1);

            -- if the parent and child are already linked, don't make
            -- another link just like it.
            if not Link Exists between Parent_Node and Child_Node
          If Child_Node already exists then
                Child_Key := Key of Existing Copy of Child_Node;
              elsif this Node is a duplicate except for enter edges then
                Child_Key := Key of Node them Child_Node duplicates
              end if;
            else
                -- we need to create this node
                Create_The_Node;
            end if; -- not link exists
           end loop; -- each edge
          Edge_Group := group_key'succ(Edge_Group);
    end loop; -- each edge group
  end loop; -- node stacks empty

 -- To allow us to do "backwards" generation of concurrency state
 -- predecessors we have to build a graph that is the equivalent of
 -- this TIG, but the edges are in the opposite direction.

 adgs.reverse_graph(PTR.Graph, PTR.Rev_Graph);
 end Retrieve_TIGs;

end Tigs_Internal_Form;
```

A.3 Algorithm for task subsetting

Algorithmic Complexity: O($T^2 \cdot E$) where T is the number of tasks and E is the total number of entries in the tasks.

```
Interesting_Tasks := User_Supplied_Interesting;
UnInteresting_Tasks := AllTasks − User_Supplied_Interesting;
Currently_Uninteresting_Entries := All Entries of Interesting_Tasks;

Procedure Calc_Interesting is
  Procedure Update_Interesting_Entries is
    ∀ UE ∈ Currently_Uninteresting_Entries loop
        ∀ CALLER ∈ CALLERS(UE) loop
            if CALLER ∈ Interesting_Tasks then
              Interesting_Entries := Interesting_Entries ∪ UE
            end if;
  end Update_Interesting_Entries;

Procedure Update_Interesting_Tasks is
    Interesting_Tasks_Is_Stable := True;
    ∀ UT ∈ UnInteresting_Tasks loop
        ∀ E ∈ entries called by U loop
             if E ∈ Interesting_Entries then
                 Interesting_Tasks := Interesting_Tasks ∪ UT;
                 Currently_Uninteresting_Entries :=
                   Currently_Uninteresting_Entries ∪ ENTRIES(UT);
                 Interesting_Tasks_Is_Stable := False;
end Update_Interesting_Tasks;

begin −− Calc_Interesting
    loop
       Update_Interesting_Entries;
       Update_Interesting_Tasks;
       exit when Interesting_Tasks_Is_Stable;
    end loop;
end Calc_Interesting;

Procedure Update_TIGs is
begin
    ∀ INTTIG ∈ Interesting_Tasks loop
        ∀ NODE ∈ NODES(INTTIG) loop
            if NODE is ENTRY_CALL and
               TARGET_TASK ∈ UnInteresting_Tasks then
                  remove_node(NODE);
            elsif NODE is ACCEPT_STATEMENT and
                  ENTRY ∉ Interesting_Entries then
                    remove_node(NODE);
            end if;
end Update_TIGs.
```

A.4 Generating the predecessors of a state

Algorithmic Complexity: Generating the predecessors of a state is O(T^3) where T is the number of tasks in the system.

```
-- Predecessor generation
-- We return the child of this node on the stack Children.
-- Successor generation is the same only we use the forward edges instead
-- of the back edges.

 procedure Gen_Pred (Parent  : open_state_type;
                     Children: in out open_state_stacks.stack;
                     TICG    : task_interaction_concurrency_graph) is

  Int_Form: tif.internal_tig_structure:= TICG.Model_Info.all;

  Child                  : open_state_type;
 Finger1,    Finger2     : tif.tig_back_edge_finger;
 Edge_Type1, Edge_Type2 : TIG_Descriptors.Edge_Kind;
 Entry_Id1,  Entry_Id2  : tif.entry_key;
 Partner1,   Partner2   : tif.tig_key;
 Next_Node1, Next_Node2 : tif.tig_node_key;
 Edge_Key1,  Edge_Key2  : tif.tig_edge_key;
 Node_Key1,  Node_Key2  : tig_state_key;

 begin
  For each Task This_Task in Program loop
   This_Node := TIG Node for This_Task in current state
      While there are more back edges from This_Node loop
       This_Edge := Next back edge from This_Node;
       If This_Edge.Edge_Type = Start_Entry_Call or
         This_Edge.Edge_Type = End_Entry_Call  then
             Partner_Task := Task called by This_Task;
             Partner_Node := TIG Node for Partner_Task in current state;

             While there are more back edges from Partner_Node loop

               Partner_Edge := Next back edge from This_Node;
               If (This_Edge.Edge_Type = Start_Entry_Call and
                   Partner_Edge.Edge_Type = Start_Accept and
                   These edges reference the same entry)  or

                  (This_Edge.Edge_Type  = End_Entry_Call and
                   Partner_Edge.Edge_Type = End_Accept and
                   These edges reference the same entry) then

                   -- create the new child (predecessor) state

                   -- the new state is the same as the old state
                   Child.States   := Kernel(Parent);

                   -- except that the nodes for the tasks that
                   -- have just finished a task interaction are backed
                   -- up to their predecessors.
                   Child.States(This_Task):= Predecessor(This_Node)
                   Child.States(Partner_Task) := Predecessor(Partner_Node)

                   -- the edges are empty, except for the ones
                   -- that represent the tasking action that
                   -- just occurred.
                   Child.Edges    := Empty_Edges;
                   Child.Edges(This_Task) := back edge from This_Node
                   Child.Edges(Partner_Task) := back edge from Partner_Node

                   -- push this state onto the stack of new states
                   -- we will return to the caller.
                   Push(Children, Child);
                 end if;

             end loop;
           end if;

         end loop;
       end loop;
 end Gen_Pred;
```

A.5 Calculating nodes to level N

Algorithmic Complexity: $O(\frac{m^{d+1}-1}{m-1})$

```
procedure mark_to_N_levels(ADG: attributable_directed_graph;
                           Start_Node_Key : adg_node_key;
                           m_array : mark_array_ptr;
                           N_levels : integer) is
  type adg_node_record is record
    key : adg_node_key;
    depth : integer := 0;
  end record;

  package n_stack is new Bounded_Stacks(adg_node_record);

  To_Be_Processed : n_stack.stack(Num_Nodes(ADG)) :=
                                  n_stack.create(Num_Nodes(ADG));
  This_Node : adg_node_record;
  Finger : Edge_Finger;
  Already_Expanded : boolean;
  Current_Depth : integer;

  Procedure Add_Node_To_List(K : adg_node_key; is_expanded : out boolean) is
  begin
    m_array(K).visited := true;
    if (not m_array(K).expanded) then
      is_expanded := false;
    else
      is_expanded := true;
    end if;
  end Add_Node_To_List;

begin
  This_Node.key := Start_Node_Key;
  Add_Node_To_List(This_Node.key, Already_Expanded);

  If Already_Expanded then
    return;
  end if; -- already visited this one
  This_Node.depth := 1;
  Mark_Array(Start_Node_Key).expanded := true;
  n_stack.push(To_Be_Processed, This_Node);
  while not n_stack.is_empty(To_Be_Processed) loop
    n_stack.pop(To_Be_Processed, This_Node);
    Current_Depth := This_Node.depth;
    Mark_Array(This_Node.Key).expanded := true;
    Finger := Edges(This_Node.key, ADG);
    While not Empty(Finger, ADG) loop
      This_Node.key := Head(Finger, ADG);
      This_Node.depth := Current_Depth + 1;
      Add_Node_To_List(This_Node.key, Already_Expanded);
      if not Already_Expanded and This_Node.depth <= N_Levels then
        n_stack.push(To_Be_Processed, This_Node);
      end if;
      Finger := Tail(Finger, ADG);
    end loop;
  end loop;
end mark_to_N_levels;
```

Appendix B
Gas station example after attempted corrections

```
with COMPUTER, RANDOM;
procedure GAS_STATION is
  type CUST_ARRAY is array( integer range <>) of CUSTOMER;
  type CUST_ARRAY_PTR is access CUST_ARRAY;
  CUSTOMERS : CUST_ARRAY_PTR;

  PUMPS : array(1..3) of PUMP;

  task body OPERATOR is
    AMOUNT_PAID, CUSTOMER_NUMBER : INTEGER;
    CHANGE_AMOUNT, IDLE_PUMP     : INTEGER;
  begin
    loop
      select
        accept PRE_PAY(AMOUNT : in INTEGER;
                       PUMP_ID: in  INTEGER;
                       CUSTOMER_ID: in INTEGER) do
          -- if no previous customer is waiting, activate this pump
          if COMPUTER.EMPTY_QUEUE(PUMP_ID) then
            PUMPS(PUMP_ID).ACTIVATE(AMOUNT);
          end if;
          -- enter a record of prepayment; records for each pump are
          -- serviced in FIFO order
          COMPUTER.ENTER(PUMP_ID, AMOUNT, CUSTOMER_ID);
        end PRE_PAY;
      or
        accept CHARGE(AMOUNT  : in INTEGER;  PUMP_ID: in INTEGER) do
          -- retrieve current record for this pump, i.e., the caller
          COMPUTER.RETRIEVE(PUMP_ID, AMOUNT_PAID, CUSTOMER_NUMBER);
          -- use new local variables declared in the operator body
          CHANGE_AMOUNT := AMOUNT_PAID - AMOUNT;
          IDLE_PUMP     := PUMP_ID;
        end CHARGE;
        -- if another customer is waiting for this pump, activate it
        if not COMPUTER.EMPTY_QUEUE(IDLE_PUMP) then
```

```
          PUMPS(IDLE_PUMP).ACTIVATE(COMPUTER.TOP_AMOUNT(IDLE_PUMP));
        end if;
        -- give change to the customer
        CUSTOMERS(CUSTOMER_NUMBER).CHANGE(CHANGE_AMOUNT);
      end select;
    end loop;
  end OPERATOR;

  task body PUMP is
    MY_NUMBER : INTEGER;
    CURRENT_CHARGES, AMOUNT_LIMIT : INTEGER;
  begin
    accept GET_ID(ID : INTEGER) do
         MY_NUMBER := ID;
    end GET_ID;

    loop
      accept ACTIVATE(LIMIT : in INTEGER) do
        AMOUNT_LIMIT := LIMIT;
      end ACTIVATE;
      accept START_PUMPING;
      accept FINISH_PUMPING(AMOUNT_CHARGED : out INTEGER) do
        -- compute actual charge as a random number less than the limit
        CURRENT_CHARGES := RANDOM.GET_CHARGES(AMOUNT_LIMIT);
        AMOUNT_CHARGED := CURRENT_CHARGES;
        OPERATOR.CHARGE(CURRENT_CHARGES, MY_NUMBER);
         end FINISH_PUMPING;
    end loop;
  end PUMP;

  task body CUSTOMER is
    MY_NUMBER : INTEGER;
    MY_MONEY, PUMP_NUMBER, AMOUNT_PUMPED : INTEGER;
  begin
    accept GET_ID(ID : INTEGER) do
         MY_NUMBER := ID;
    end GET_ID;

    loop
      -- randomly decide amount of gas needed and choice of pump
      RANDOM.DRIVE_IN(MY_MONEY, PUMP_NUMBER);
      OPERATOR.PRE_PAY(MY_MONEY, PUMP_NUMBER, MY_NUMBER);
      PUMPS(PUMP_NUMBER).START_PUMPING;
      PUMPS(PUMP_NUMBER).FINISH_PUMPING(AMOUNT_PUMPED);
         accept CHANGE(AMOUNT : in INTEGER) do
        -- check to see if we received the right amount of money
      end CHANGE;
    end loop;
  end CUSTOMER;

begin
```

```
  COMPUTER.GET_NUMBER_OF_CUSTOMERS;
  CUSTOMERS := new CUST_ARRAY(1..COMPUTER.CURRENT_CUSTOMERS);

  for K in 1..COMPUTER.NUMBER_OF_PUMPS loop
    -- assign PUMP ids
    PUMPS(K).GET_ID(K);
  end loop;

  for K in 1..COMPUTER.CURRENT_CUSTOMERS loop
    -- assign CUSTOMER ids
    CUSTOMERS(K).GET_ID(K);
  end loop;
end GAS_STATION;
```