A Concurrency Analysis Tool Suite for Ada Programs: Rationale, Design, and Preliminary Experience

MICHAL YOUNG
Purdue University
and
RICHARD N. TAYLOR, DAVID L. LEVINE, KARI A. NIES, and DEBRA BRODBECK
University of California

CATS (Concurrency Analysis Tool Suite) is designed to satisfy several criteria: it must analyze implementation-level Ada source code and check user-specified conditions associated with program source code; it must be modularized in a fashion that supports flexible composition with other tool components, including integration with a variety of testing and analysis techniques; and its performance and capacity must be sufficient for analysis of real application programs. Meeting these objectives together is significantly more difficult than meeting any of them alone. We describe the design and rationale of CATS and report experience with an implementation. The issues addressed here are primarily practical concerns for modularizing and integrating tools for analysis of actual source programs. We also report successful application of CATS to major subsystems of a (nontoy) highly concurrent user interface system.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

General Terms: Design, Reliability, Verification

Additional Key Words and Phrases: Ada, concurrency, software development environments, static analysis, tool integration

This article is a major revision of Young et al. [1989]. This material is based on work sponsored by the Defense Advanced Research Projects Agency under grant MDA972-91-J-1010. Additional support was provided by the Software Engineering Research Center, an NSF Industry/University Cooperative Research Center, and by the National Science Foundation under grant CCR-9010135. The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

Authors' addresses: M. Young, Software Engineering Research Center, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398; email: young@cs.purdue.edu; R. N. Taylor, D. L. Levine, K. A. Nies, and D. Brodbeck, Department of Information and Computer Science, University of California, Irvine, CA 92717-3425; email: {taylor; kari; brodbeck}@ics.uci.edu; levine@louie.timeplex.com.

Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1995 ACM 1049-331X/95/0100-0065 $03.50

1. INTRODUCTION

We describe the design and rationale of a suite of tool components for statically analyzing concurrent programs, with systematic exploration of potential synchronization behaviors (reachability analysis) as a central technique. The Concurrency Analysis Tool Suite (CATS) is designed to meet several objectives together:

Analyzing Implementation-Level Code. Successful use of reachability analysis techniques has been limited primarily to specification and high-level designs, e.g., protocol specifications rather than implementations. CATS is intended to analyze implementation code annotated with partial specifications in the form of assertions. Although the same basic principles apply, there are several pragmatic differences between analysis of high-level designs and analysis of actual implementation code. At the implementation level the relevant synchronization structure of a program is buried in a mass of irrelevant detail. Separating essential structure from irrelevant detail should be as automatic as possible.

Modularity for Integration. A salient characteristic of CATS is a modular design with appropriate interfaces for integration of analysis, testing, and verification techniques in a software development environment. No single technique so dominates others that a monolithic tool or environment for supporting that technique is sufficient for the production of quality software. A primary goal of this work is therefore to produce an open framework in which a variety of analysis and testing tool components can be effectively integrated.

The toolset separates language-specific and language-independent components in a manner that supports integration with language processing tools in an environment. Appropriate decomposition and interfaces are provided to support association of user-specified properties with source code while maintaining this separation. This is most clearly illustrated by a temporal logic model checker divided into an atomic proposition checker and a combiner of subformulae.

Performance and Capacity. Enumerative techniques are widely considered impractical for analysis of concurrent software, because of well-known lower bounds on complexity.[1] Nonetheless we advocate a central role for reachability analysis in analysis and testing of concurrent programs. We report experience applying CATS to an actual, moderate-size concurrent system (a highly concurrent user interface system which was *not* constructed for analysis) as well as several variations on a canonical (toy) program. Although global analysis of large systems will certainly not be feasible, experience to

[1] A lower bound reflects the inherent complexity of a problem, rather than the complexity of an algorithm. The same lower bounds apply thus to other approaches analyzing concurrent systems.

date suggests that exhaustive reachability analysis of natural modules in real-world software systems is both feasible and useful. Analysis of the user interface example described here relies on division along a client/server split. In work reported elsewhere [Yeh 1993; Yeh and Young 1991; 1993; 1994] we are developing approaches to combine these modular analyses hierarchically.

2. ANALYSIS APPROACH

The term "reachability analysis" is used to describe construction of a state-transition model of larger modules (or a complete system) from models of individual processes. The composite state-transition model is often called a "reachability graph." These models typically highlight synchronization structure and abstract away other details of execution. Reachability analysis has been applied to Petri nets and CSP-like and CCS-like state machine models, among others [Apt 1983; Fernandez et al. 1992; Morgan and Razouk 1987; Peterson 1981; Taylor 1983b]. We use the term "static concurrency analysis" for reachability analysis of finite-state models extracted from program source code, as described in Taylor [1983b].

A primary use of reachability analysis is verification of properties of the synchronization structure of software, e.g., freedom from deadlock, freedom from starvation, and mutual exclusion. With respect to these properties, reachability analysis provides the same level of assurance as formal verification. Reachability analysis can also be used to support other verification and validation techniques, as we discuss in Section 5.

Reachability analysis suffers from two major kinds of problems. First, the details abstracted away in the simplified models may be essential to the correctness of software. Omitting these details often has the effect of producing spurious error reports. Approaches to providing useful analysis of a simplified model of programs differ significantly between design verification (e.g., communication protocol modeling) and analysis of programs, since in the latter case it is not feasible to place the burden of extracting the "right" abstraction entirely on the user. Second, the size of a global model will usually grow as the product of the sizes of individual process models. Moreover, basic complexity results [Ladner 1979; Smolka 1984; Taylor 1983a] imply that there is no universally applicable short-cut without further sacrificing accuracy. The so-called "state explosion problem" has been addressed in a number of ways, including compact encodings (e.g., binary decision diagrams, partial order models) and hierarchical analysis methods.

General Analysis Model. A variety of analysis techniques can be framed in the general model presented in Figure 1, which decomposes analysis into generation and testing of possible behaviors. An abstract representation of the system is constructed, and from this a representation of possible behaviors is constructed. This representation of possible behaviors is checked against a specification of acceptable behaviors, and violations of the specification are reported. This basic framework is general enough to describe testing (where the model is a program; the representation of behaviors is a set of actual runs; and the checking procedure is the test oracle), reachability

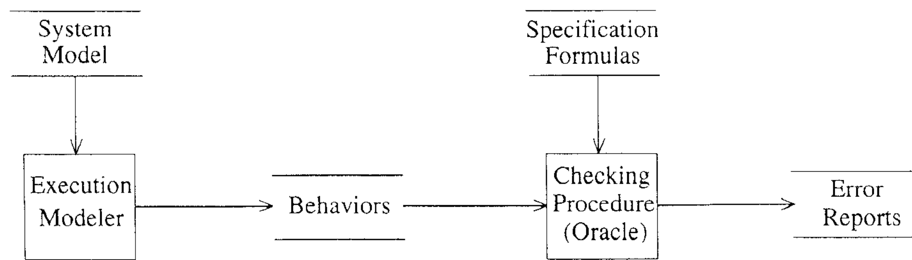

Fig. 1. Generic skeleton of a program analysis technique

analysis (where the representation of behaviors is a state-transition graph), and constrained expression analysis [Avrunin et al. 1986; Dillon et al. 1988] (where the representation of behaviors is a set of inequalities, and the checking procedure is a linear inequality solver), among many others. Although very broad, the model is not so general as to be devoid of content. The general model is strongly biased toward operational (possibly abstract) interpretations of programs; it does not describe purely structural analyses like type checking and interface consistency checking, nor derivation of programs together with their correctness proofs.

With only a slight stretch of the imagination, one can place some kinds of formal verification in this framework: the model is the program, augmented by auxiliary variables or control predicates; the specification includes assertions in the program; and the representation of actual behavior consists of a set of theorems derived from the program, axiom schemata, and rules of inference. The program is accepted if each assertion in the specification also appears in the representation of actual behavior. This characterization does not have much to do with the way formal verification is practiced, but it may help in thinking about how formal verification can be combined with other techniques.

The general model of Figure 1 need not reflect the actual organization of a tool, but we will argue below that it is a good logical decomposition for reachability analysis tools. The following section describes a particular set of tool components which are organized along the lines of this model.

3. AN ANALYSIS TOOLSET FOR CONCURRENT ADA PROGRAMS

CATS is a set of tools built according to the general concurrency analysis model just described. It is initially targeted for concurrent Ada programs. CATS was designed particularly for integration with other testing, analysis, and verification tools in a software development environment. Basic design decisions for reachability analysis are discussed first, in Section 3.1. The system architecture is described in Section 3.3.

3.1 Design Considerations for Reachability Analysis

The design of CATS instantiates the general framework of Figure 1 with tool components in a manner driven by lessons from previous implementations of

similar systems, namely, a clear logical separation of modeling from analysis, and a model representation whose semantics are distinct from the semantics of the software being modeled. The following paragraphs discuss these considerations, how they were influenced by other reachability analysis systems, and how they were applied to the design of CATS.

Separating Modeling from Analysis. Tight coupling of modeling (construction of a reachability graph) with analysis is tempting. For instance, one could easily combine generation of successor states with a check for deadlock, perhaps avoiding some redundant computations. An earlier prototype tool for analyzing concurrent Ada programs, constructed at the University of California, Irvine, did just that [Wampler 1985]. In some other reachability analysis tools, analysis of a reachability graph is kept strictly separate from generation of the graph. In the PNut system for analysis of Petri nets [Morgan and Razouk 1987; Razouk 1987], reachability graphs are built by one program and analyzed by a completely separate program, with communication between the two only via the file system or Unix pipes. Important benefits accrue from this separation:

—The analysis component can be used with different modeling approaches. In PNut, a complete reachability graph can be generated by the Reachability Graph Builder (RGB) tool, or a single trace can be generated by the Petri net simulator. A trace is a degenerate reachability graph, so the Reachability Graph Analyzer (RGA) tool can be applied to the output of either modeling tool.
—The analysis component may be used with different models. A temporal logic model checking tool constructed at Carnegie Mellon [Clarke et al. 1985] has been applied to models of sequential circuits as well as concurrent software, and it has also been interfaced to other finite-state verification systems, e.g., the concurrency workbench [Cleaveland et al. 1990].
—Both the modeling component and the analysis component are likely to be simpler.

This factoring of analysis from modeling need not imply a sequential two-phase operation in which all modeling precedes all analysis, although current operating system substrates make that the easiest way to compose tool components. The goal is to maintain a logical separation of modeling and analysis, and a clean interface between them, without ruling out tight integration and feedback from the analysis component to the modeling component.

Ideally the modeling tool should have both configuration options and "hooks" for analysis. The example above, of checking for deadlock as each state is generated, can be accommodated by a hook for checking local properties of each state; a number of other safety properties can be checked in this way, and state information required only for those checks need not be incorporated in the reachability graph. If only local checks are required, only the minimal amount of state information to ensure termination in the modeling engine need be stored. Reductions that preserve particular specified

properties but not others can likewise be divided between the modeling and analysis component in a way that does not violate their logical separation, e.g., the modeling component can provide operations for abstracting away selected actions and minimizing a model; and the analysis tool can determine which actions must be preserved.

Well-Defined Internal Representations. Modeling tools for Petri nets or other models with simple operational semantics have obvious advantages over tools for directly modeling more complex phenomena such as concurrent software written in a language like Ada or CSP. The usual approach to modeling software is first to build a simplified model of the complex artifact, and then to model executions of the simplified representation. An important lesson learned from earlier prototypes noted above is that the advantages of this translation accrue only if the simplified model has an operational semantics independent of the original artifact. That is, one must be able to decompose the question

Does the modeling tool model behaviors of the software accurately?

into the two simpler questions

Is the simplified representation an accurate[2] *representation of the original artifact?*

Does the modeling tool model behaviors of the simplified representation correctly?

If the operational semantics of the simplified representation is defined by reference to the original artifact (e.g., "*node type X represents an Ada entry call*,") this decomposition is lost. This makes it more difficult to assure oneself that the modeling process is accurate or to diagnose the problem when it is not. Without an independent semantics for the simplified representation, development of the modeling tool depends on reasoning about the original artifact. When the combination of translation and modeling fails, the tool developer cannot easily localize the problem to an inadequate representation on the one hand or a failure of the modeling tool on the other.

Use of a completely independent modeling formalism such as Petri nets or CCS provides the requisite independent semantics, but may do so at considerable cost in clarity and performance. Performance is not an issue when the purpose of the model is to formalize and clarify language semantics, as in the translation of Ada tasking to Petri nets described by Mandrioli et al. [1985],

[2] It is necessary to say precisely what one means by "accurate." One approach is to insist that a model represent all and only the possible behaviors of a piece of software (both *accurate* and *precise*, in the terminology of Taylor [1983b] and Dillon et al. [1988]). However, a model which meets this absolute standard cannot overcome fundamental complexity bounds (e.g., undecidability or NP-hardness), because the relation between model and original artifact amounts to a problem reduction. A reasonable approach is to insist that a model represent all possible erroneous behaviors, but allow the possibility that some impossible erroneous behaviors are also represented. A precise definition of this criterion with sufficient conditions for ensuring accuracy with respect to temporal specification formulae is described in Young [1988].

but expansion during translation is an important consideration when the goal is automated analysis. In some cases the availability of a body of existing theory and off-the-shelf tools may partly or wholly compensate for clumsy translation (e.g., application of Petri net reductions and reachability analysis tools as in Shatz et al. [1990], but in many cases it is worthwhile to customize a modeling formalism to an application domain. This specificity need not compromise the independence of the two semantics.

3.2 The TIG Model of Ada Tasking

The Task Interaction Graph model of Ada tasking was introduced by Long and Clarke [1989]. Task interaction graphs (TIGs) represent individual Ada tasks. A task interaction concurrency graph (TICG) is a reachability graph representing the concurrent execution of a set of tasks making up a program.

To make this article self-contained and to clarify our modifications to the model (including how certain "optimizations" may be performed during construction of TIG models from Ada code), we present the original and modified models here with translations from Ada. In the interest of clarity, we describe the TIG model as a kind of labeled transition system, and our terminology and presentation differ somewhat from that of Long and Clarke [1989].

A task interaction graph (TIG) represents the structure of a single Ada task. Roughly speaking, TIG nodes represent states and sequences of nonsynchronization activities, while TIG edges represent synchronization activities. For present purposes a synchronization activity is an entry call, an accept statement, select, select-else, task-begin, or task-end.

Ada rendezvous must be modeled by two steps (engage and finish) for the general case including synchronization activities within accept bodies. The common special case of rendezvous without nested synchronization can be recognized in a prepass, and for that case the two steps can be collapsed into one.

The TIG model does not explicitly represent execution time or priorities. Partly this reflects a choice to focus on properties that do not depend on the execution platform (e.g., single or multiple processors), and partly it is a compromise to avoid the expense of a more detailed model. A consequence is that it is not necessary to model FIFO service explicitly at Ada entry queues; the orders in which tasks could execute entry calls are identical to the orders in which they could be engaged. If a task could be starved by non-FIFO queuing, it can also be starved at the point just before it makes an entry call. Entry queues do prevent starvation when one assumes fair scheduling (and only then). For example, a round-robin scheduler would prevent a task from waiting forever to make an entry call, and FIFO queuing could then ensure eventual engagement. Fairness properties never change the set of reachable states (e.g., they cannot prevent deadlock). FIFO queuing is more cheaply represented as a fairness assumption external to the explicit graph representation of reachable states.

In order to use a general rule for building reachability graphs from sets of labeled flowgraphs, rather than a case-by-case set of rules for the different

synchronization activities in Ada, we introduce a set of labels similar to those used in process algebras for modeling elementary actions [Baeten and van Glabeek 1987; Bergstra and Klop 1984; Milner 1989].

It is usual in labeled transition systems to introduce a single unobservable action τ, letting all other actions be communication actions. For our purposes it will be more convenient to introduce a whole set of noncommunicating actions H, of which η is a distinguished unobservable action. (No member of H participates in communication, but only η may be removed by such common axioms as $a; \eta; b = a; b$.)

Definition 3.2.1 (*Actions*). Let $L = \Sigma \cup \overline{\Sigma} \cup H$ be a finite set of elements called *actions*. Σ, called the set of *names*, and $\overline{\Sigma}$, called the set of *conames*, are subsets of L such that names $\sigma \in \Sigma$ are in one-to-one correspondence with conames $\overline{\sigma} \in \overline{\Sigma}$. $\overline{\sigma}$ is called the complementary action of σ. By a small abuse of notation, we denote the union of this bijection and its inverse as a function "bar" of one argument, so that $\overline{\overline{\sigma}} = \sigma$. H is the set of noncommunicating actions, and the distinguished element $\eta \in H$ is called the silent action.

We write Σ, $\overline{\Sigma}$, or H where no confusion will arise about which set L they are taken from.

Definition 3.2.2 (*Labeled Flowgraph*). A labeled flowgraph is a 6-tuple

$$G = \langle N, A, L, s, T, l \rangle$$

where N is a finite set of elements called nodes; $A \subseteq N \times N$ is called the edges; $s \in N$ the distinguished start node; $T \subseteq N$ is the set of terminal nodes; $L = \Sigma \cup \overline{\Sigma} \cup H$ is a set of actions; and $l : A \to L$ is a labeling function for edges. We write $n \xrightarrow{x} m$ to indicate that there is an edge from n to m whose label is x.

A TIG is a labeled flow graph with actions (edge labels) corresponding to tasking activities. An Ada program will be represented by a set of TIGs sharing the same set L of actions. The presence of complementary actions in different graphs allows the identification of corresponding synchronization actions in different tasks. By convention we will use names like x from Σ for actions that represent entry calls, and conames like $\bar{x}$ from $\overline{\Sigma}$ for actions representing Ada accept statements. When we describe the TICG (reachability graph) below, we will use bracketed symbols like $[x]$ from H to represent rendezvous. Whereas x and $\bar{x}$ are communicating actions, their joint occurrence as $[x]$ (i.e., a rendezvous) is a noncommunicating action because it does not communicate or synchronize with any third party. This is similar to the usual practice in labeled transition systems of using the distinguished silent action to represent completed communications, except that we preserve the identity of individual rendezvous as elements of H.

Although we limit consideration here to two-party rendezvous between Ada tasks, it is perhaps worth mentioning that the same representation scheme extends easily to synchronization with passive entities like protected records

in Ada 9X (similar to monitors), which could be represented as additional TIGs. Additionally, actions other than synchronization could in principle be represented by elements of H. However, in reachability analysis one usually wants to reduce the number of distinct interleavings and the number of states that must be enumerated by every means available. Shared-variable accesses, including protected-record access, can be more cheaply represented by associating potential access with nodes, with no loss in diagnostic power. (This would not be true of protected records in Ada 9X if their access procedures were permitted to execute potentially blocking operations.)

Long and Clarke include one more component in TIGs, a function C that assigns "pseudocode" (regions of Ada source code) to each node. We omit C from the basic definition above because one of our optimizations will require associating source code with edges rather than nodes. Informally it will still be useful to think of task regions being associated with nodes in a TIG. The task regions for the Ada program of Figure 2 are shown in Figure 3. Construction of a TIG corresponding to task T2 of this Ada program is illustrated in Figure 5. We illustrate only the case of rendezvous without accept bodies to keep the examples as small as possible.

The TIG model as originally defined by Long and Clarke requires that each task region begin with a single synchronization action, i.e., edges incident to a TIG-node correspond to the same synchronization action. This simplifies bookkeeping and improves the accuracy of analyses other than reachability analysis (e.g., the dataflow analysis reported in Long and Clarke [1991]). Duplication of TIG nodes with identical synchronization behavior has a substantial cost, as shown by the experience reported in Section 4.1 (see Table I). The putative bookkeeping advantages of duplication can be obtained at no cost by associating code regions with TIG edges rather than TIG nodes. In the construction below we will actually produce a modified TIG model rather than the original model as described by Long and Clarke, since it is easier to describe translation from the modified model to the original than vice versa. (Implementing either transformation is straightforward.)

Constructing TIGs from Ada. The first step in the construction of a TIG consists of making all task and entry names unique and of removing statements not involving communication. Procedures and functions containing tasking activity are expanded in line.[3] A TIG representation of task T can then be constructed in a syntax-directed manner, as illustrated in Figure 4.

The construction is similar to the usual NFA construction for lexical recognizers (Thompson's construction). However, since the axiom $\epsilon x = x$ for regular languages has no direct analogue for labeled flowgraphs [Kannellakis and Smolka 1990] (see the discussion of internal choice below), we cannot always build up graphs from subgraphs by adding silent moves. In addition to

[3] Recursive procedures with tasking activities cannot be directly represented by TIGs, nor by any finite-state representation, but we can construct representations of recursive procedures by TIGs or other finite-state transition systems that over- or underestimate possible behaviors, in the same way that a finite-state acceptor can be constructed to accept a superset or subset of a context-free language.

```
with text_io,

procedure main is
  task T0,
  task T1 is
      entry P,
      entry Q;
  end T1;
  task T2,


  task body T0 is
  begin
    T1 P;
  end T0;

  task body T1 is
    package boolean_io is new text_io.enumeration_io(boolean);

    done: boolean:
  begin
    loop
      select
        accept P,
      or
        accept Q;
      end select,
      boolean_io.get(done);

     exit when done,
     end loop;
   end T1,

  task body T2 is
  begin
    T1.Q;
  end T2;

begin
  NULL,
end main,
```

Fig. 2. A simple Ada program.

introducing nodes and labeled edges, we combine single-entry, single-exit flowgraphs by merging nodes. The notation merge(n, S) means that the start node s of S is identified with node n. All edges $s \xrightarrow{x} m$ are replaced by edges $n \xrightarrow{x} m$. Similarly, the notation merge(S, n) means that the terminal node t of S is identified with n, and edges $m \xrightarrow{x} t$ are replaced by $m \xrightarrow{x} n$.

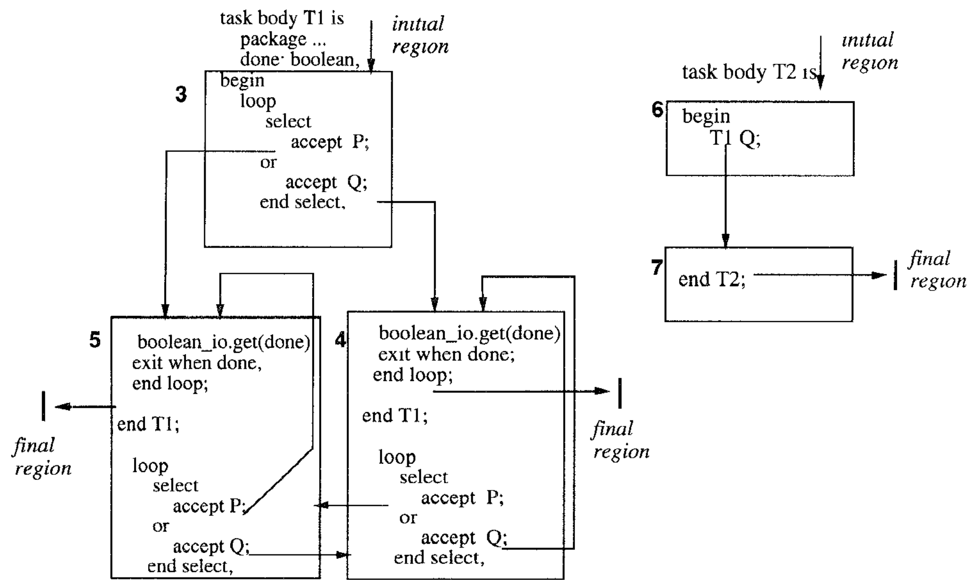

Fig. 3. Task regions in the TIG model of the Ada program of Figure 2. Regions are single entry, which causes duplication of the loop body.

Construction 3.2.3

—call Q *where* Q *is a task entry*. Introduce two nodes m and n (the initial and final node, respectively) and a single edge $m \overset{Q}{\rightarrow} n$. Let $Q \in \Sigma$. An accept is represented similarly, except the entry name is represented by $\overline{Q} \in \overline{\Sigma}$. See Figure 4(a).

—$S_1; S_2$. Merge the final node of S_1 and the initial node of S_2. See Figure 4(b).

—while C loop S_1; end loop. Introduce a new node m. Replace initial and terminal nodes as follows: merge(m, S_1), merge(S_1, m). Loops with exits in other positions must first be transformed to while loops by "rolling" and duplicating any code before the exit. See Figure 4(d).

—select S_1 or $\cdots$ or S_n end select; *where each of the* S_i *is an* accept *alternative*. Introduce two nodes m and n, and replace the initial and final nodes of each S_i by m and n: merge(m, S_i) and merge(S_i, n). See Figure 4(e).

—if $\cdots$ then S_1; else S_2; end if; Introduce two nodes m and n, and replace the initial and final nodes of S_1 and S_2 by m and n: merge(m, S_1), merge (S_1, n), merge(m, S_2), merge(S_2, n). Note this is identical to the rule for select. We will distinguish internal nondeterministic choice (if) from external nondeterministic choice (select) by attributing nodes, below. See Figure 4(c–f) for a more intuitive view of how this distinction is maintained.

—task body $\cdots$ is $\cdots$ begin S; end; The labeled flowgraph for S becomes the labeled flowgraph for the task, with the initial and terminal nodes for S distinguished as initial and terminal nodes for the task.

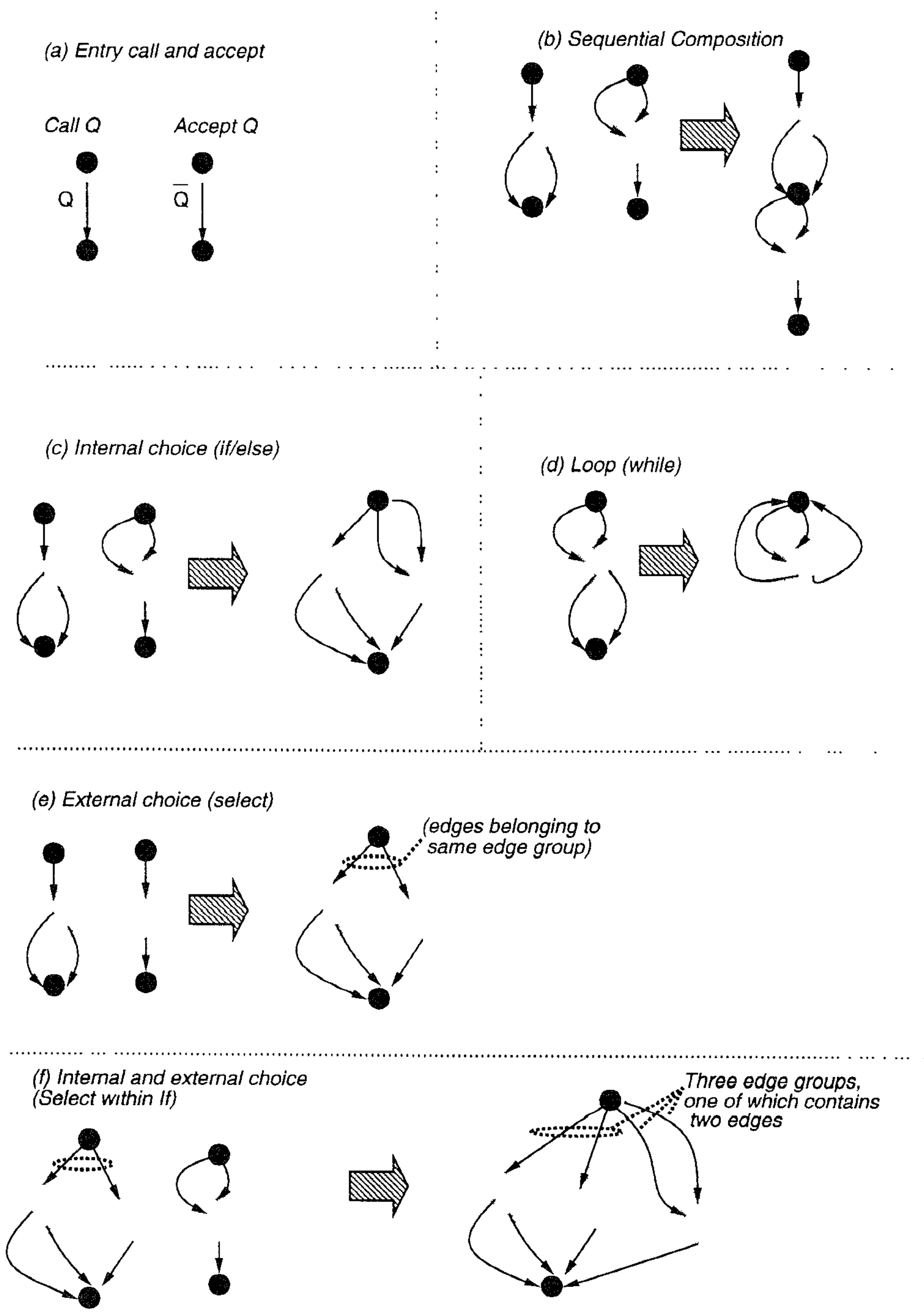

Fig. 4. Illustration of syntax-directed construction rules for TIGs.

The construction above does not introduce any invisible moves into the TIG. Implementations will typically obtain the same result but use a different approach: they begin with a control flow graph representation and treat every noncommunicating action as an invisible action. Invisible actions are removed in a step analogous to (but different from) the well-known subset construction for producing deterministic acceptors from nondeterministic acceptors.

We omitted in this construction the annotation of TIG nodes by "edge groups," which are necessary for deadlock checking. Edge groups are described below, after the transition rule for TIGs, and their use in deadlock checking is described in Section 3.3.1.

The construction above does not produce a different TIG node for each in-edge; it already incorporates one of the "optimizations" that we found critical to reducing the size of reachability graphs. If desired, nodes can be split to obtain the model defined by Long and Clarke (one copy for each in-edge, treating initiation also as an in-edge) in a postprocessing step. This is shown in Figure 5.

Transition Rule for TIGs. The rule for combining TIG representations of individual tasks into TICGs (reachability graphs) is simple and typical of reachability graph constructions; we match communicating actions and label reachability graph TICG nodes with tuples of TIG nodes.

Definition 3.2.4 (*Reachability Graph*). Given a domain N, a distinguished start node n_0, and a successor relation $\Gamma \subseteq N \times N \times L$, the reachability graph is the smallest labeled flowgraph $G = \langle M, E, L, n_0, T, l \rangle$ such that (1) $M \subseteq N$, (2) if $n_i \in M$ and $\langle n_i, n_j, a \rangle \in \Gamma$ then $n_j \in M$, $\langle n_i, n_j \rangle \in E$, and $\langle \langle n_i, n_j \rangle, a \rangle \in l$, (3) if $n_i \in M$ and $\nexists n_j, a$ such that $\langle n_i, n_j, a \rangle \in \Gamma$, then $n_i \in T$.

Informally, a domain of nodes is just a set of well-formed state descriptions. Condition (1) says that states in a reachability graph will be well formed; condition (2) forces it to contain all reachable states; and condition (3) identifies terminal states in the obvious way.

Definition 3.2.5 (*Task Interaction Concurrency Graph* (*TICG*)). Given a set of k TIGs, a TICG-node is a k-tuple of TIG-nodes, one for each TIG. A TICG is the reachability graph where the domain is TICG-nodes, the initial node $\langle s_1, \ldots, s_k \rangle$ where s_i is the initial node of TIG i, and with the following successor relation:

$$\langle m_1, \ldots, m_k \rangle \stackrel{[x]}{\rightarrow} \langle n_1, \ldots, n_k \rangle$$

iff there exists i and j such that

$$n_i \stackrel{x}{\rightarrow} m_i, \quad n_j \stackrel{\bar{x}}{\rightarrow} m_j, \quad \text{and for all } l \neq i, j, \quad m_l = n_l.$$

Informally, this says that a state transition consists of an individual pair of synchronizing tasks (i.e., this is an *interleaving model* of concurrency rather

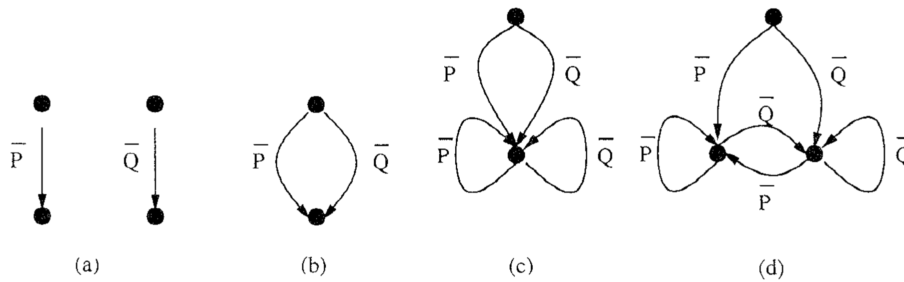

Fig. 5. Example TIG construction. Steps in the construction of a model of task $T1$ from Figure 2 are shown: (a) the representation of individual accept statements, (b) joined by a select statement, (c) enclosed in a loop (after unrolling one iteration to move the exit test to a while position). In step (d) nodes are split so that all in-edges to a single node represent the same interaction as required by the original TIG model as described in Long and Clarke [1989]. Omitting step (d) improves performance at a slight cost in bookkeeping (edges rather than nodes must be associated with source code regions) but no cost in modeling accuracy.

than a *true concurrency model* in which several independent actions might occur simultaneously).

The TICG corresponding to the Ada program of Figure 2 is shown in Figure 6). TICG-nodes are labeled with triples that describe the state of the three tasks that comprise the Ada program.

Edge Groups. Potential deadlock is among the most important program properties that reachability analysis techniques use for detection. One may intuitively expect that deadlock is manifested by a terminal node in a reachability graph, but this is not always so. In particular, the deadlocks that result when task $T1$ of Figure 2 executes its loop body exactly once are represented by nonterminal nodes in the reachability graph (TICG) constructed from the TIG model of the program (nodes 2-5-6 and 1-4-7 in Figure 6). The TICG folds together deadlocked states and nondeadlocked states, resulting in a smaller state space but a nontrivial check for possible deadlock at each reachability graph node. The difference is a direct consequence of the treatment of so-called "internal choice," i.e., arbitrary scheduling and control-flow choices.[4]

The task interaction graph model produces flowgraphs with no silent moves (η actions, in our flow-graph model). However, the distinction between select and if is essential and must be maintained (see Figure 7). In fact, the flowgraph model as described so far does not preserve enough information about the branching structure of programs to detect potentially deadlocked states. Additional information must be associated with flow graph nodes to preserve this information.

[4] Since we do not model the values of variables (with a few exceptions described in Section 4), we must consider data-dependent control-flow branching as arbitrary choice. Although scheduling decisions are in principle also arbitrary, issues of representing scheduling decisions do not arise until one considers hierarchical techniques in which some scheduling decisions are already represented in an incomplete model of the system under analysis

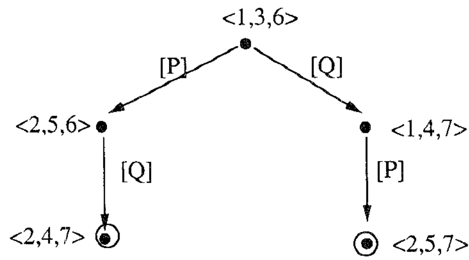

Fig. 6. The TICG corresponding to the Ada program of Figure 2. Numbers correspond to task regions (TIG nodes) in Figure 3.

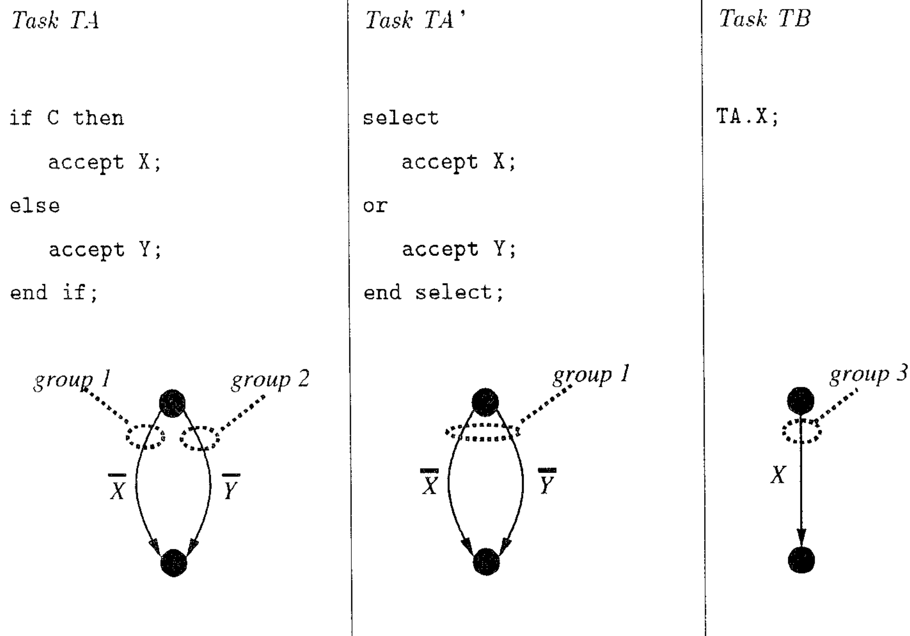

Fig. 7. Potential deadlock: if versus select. Tasks TA and TB can deadlock, but replacing TA by TA' removes the possibility of deadlock.

Nondeterministic choices in TIGs are modeled by *edge groups*, which carry the same information as *refusals* [Brooks et al. 1984] or *acceptance sets* [Hennessey 1988] in process algebras. Each out-edge from a TIG node belongs to a single group. (For purposes of deadlock checking we treat termination as an out-edge leading to a final region.) An edge that is not part of a select statement will form a singleton group. A select statement with no delay or else alternative will be represented by an edge group containing one TIG edge for each select alternative. Thus, the difference between the TIG representations of tasks TA and TA' in Figure 7 is that the two edges representing accept statements belong to different edge groups in the case of TA, but they belong to a single edge group in the case of TA'.

There are two kinds of edge groups, *blocking* and *nonblocking*.[5] Any task interaction that is not a select alternative is a blocking edge group. Edges corresponding to alternatives of a select statement without an else or delay alternative form a blocking edge group. In a select statement with a delay or else alternative, all select alternatives with the exception of the delay or else make up a nonblocking edge group. The else or delay alternative produces one or more blocking edge groups, depending on subsequent control flow.

Consider the following Ada fragment:

```
select
   accept A;
or
   accept B;
end select;
```

In the TIG representation of this fragment, the edges labeled by $\bar{A}$ and $\bar{B}$ belong to the same edge group, which is a blocking edge group. However:

```
select
   accept A;
else
   accept B;
end select;
```

In this fragment, else accept B; is semantically equivalent to or delay 0.0; accept B;. The else separates the two edges into separate groups. The edge labeled by $\bar{A}$ belongs to a nonblocking group, while the edge labeled by $\bar{B}$ belongs to a blocking group.

An infinite wait cannot occur in a program execution state in which a select delay or else alternative is possible. However, it is a consequence of the way TIGs are constructed from Ada programs that a TIG node cannot have *only* out-edges belonging to nonblocking edge groups. Only blocking edge groups need be considered for purposes of checking individual TICG nodes for potential deadlock, as described in Section 3.3.1. Since nonblocking edge groups are irrelevant for this check, in the rest of this article "edge group" or "group" will mean a blocking edge group.

3.3 CATS Architecture

The organization of the CATS system, and the tool components which comprise the system, fits the general model of Figure 1. A more-detailed diagram of information flow in the CATS system is presented in Figure 8. The system model is a set of TIGs [Long and Clarke 1989] derived from the source code of a concurrent Ada program or another design notation for rendezvous-style concurrent systems. A compiler front end produces a semantically analyzed graph representation of the program, and this representation is then translated by the TIGGER component into the task interaction group representa-

[5] In Long and Clarke [1989], blocking edge groups are called one-edge groups, and nonblocking edge groups are called zero-edge groups. We have modified the terminology to make the discussion clearer.

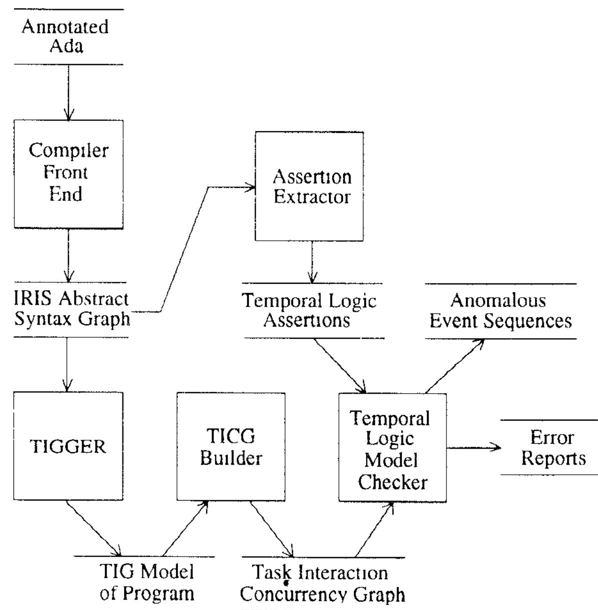

Fig. 8. Information flow among tool components in CATS.

tion described in Section 3.2. An enumeration of possible sequences of interactions among tasks is represented by a global state-transition graph, which we call a task interaction concurrency graph (TICG). Explicit specifications in a branching-time propositional temporal logic are checked using the decision procedure described by Clarke et al. [1986], and a separate procedure verifies freedom from deadlock. Behaviors which violate the specification formulas are reported directly to the user.

In general, the size of a TICG may be the product of the sizes of the individual task interaction graphs from which it is constructed. Exhaustive global analysis of large systems can never be practical, so parceling of analysis is an absolute requirement. The granularity to which systems must be divided for analysis depends on space efficiency, fast traversal, and locality of reference in the representation of TICGs. A reasonable goal is that the maximum practical granule size should be large enough so that, when a system is divided into modules in design, each natural module can be analyzed in whole. We have constructed a separate prototype to experiment with scalable, divide-and-conquer analysis (but which does not address tool integration or other critical constraints on CATS); the result of this effort is reported elsewhere [Yeh 1993; Yeh and Young 1991; 1994]. Our experience suggests that a modular analysis will be practical for large but well-structured systems if graphs representing a few thousands of states and events are constructed and analyzed in a few minutes.

Separation of concerns suggests an organization which clearly separates the generic aspects of reachability graphs from those aspects of TICGs

dependent on the language of the software under analysis (e.g., Ada). As it happens, a design that isolates language-dependent features serves also the efficiency needs of reachability analysis. For these reasons, task interaction concurrency graphs in CATS are divided into two parts: a language-independent attributable graph structure and a set of attributes.

The graph structure underlying a TICG contains no attributes except the connectivity of the reachability graph. The attributable graph structure is completely independent of whether the reachability graph is built from task interaction graphs, a Petri net, or some other model. States and events are represented as attributes of nodes and edges, respectively, of the basic attributable graph structure. Attributes are encapsulated separately from the graph structure.

For space and time efficiency, a parallel array structure is used for representing the graph structure and its attributes (Figure 9). This results in good locality of reference, since a typical reference pattern is to traverse the whole graph while accessing only one or two attributes. In particular, checking temporal logic specification formulas involves a traversal for each subformula; and in each traversal one or two boolean attributes is accessed, and another is produced. The efficiency and convenience of the parallel array representation comes at some cost in information hiding: the representation of state and edge identifiers as a dense set of keys is visible to several components.

An interesting example of reuse of the generic attributable graph structure underlying the TICGs is a special-purpose, internal representation of task interaction graphs. Just prior to constructing a TICG, an optimized internal representation of task interaction graphs is built (e.g., with all identifiers of tasks and entries replaced by small integers), and references to the original structure are treated as attributes of the graph. The original structure of the TIGs reflects a different set of design tradeoffs than reachability graphs: task interaction graphs of individual tasks are much smaller than TICGs, so compromising abstraction for performance would be inappropriate. They are instead represented by an attributed graph structure generated automatically by *P-Graphite* [Wileden et al. 1988]. *P-Graphite* manages persistence of TIGs, allowing them to exist beyond the lifetime of a single program without explicit input/output. The last-second translation of this representation into optimized form allows CATS to increase performance without losing all the benefits of the *P-Graphite*-based representation.

3.3.1 *Checking Sequencing Constraints.* CATS checks two kinds of sequencing conditions on the behavior of concurrent software as modeled by a task interaction concurrency graph. Freedom from deadlock is considered an implicit specification, and is checked using a special-purpose procedure. Additional constraints can be explicitly specified by the user, by embedding temporal logic assertions in the Ada source code. Temporal logic assertions are checked using an adaptation of the model-checking algorithm of Clarke et al. [1986]. When a sequencing error is detected, example violating behaviors (sequences of events) are produced and reported to the user.

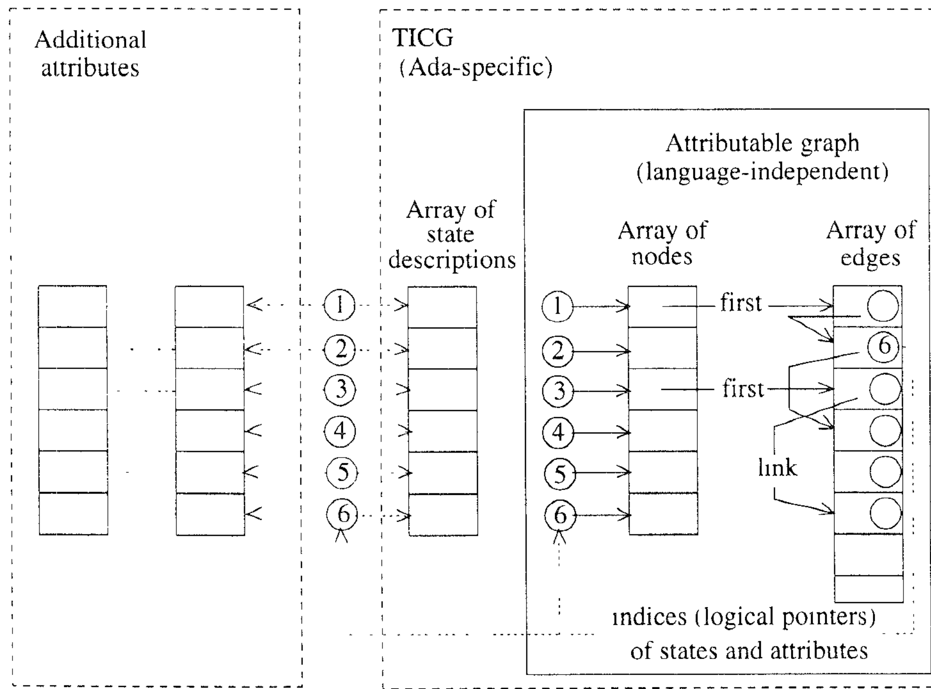

Fig. 9. Division of TICGs into a language-independent attributable graph structure and a set of attributes, using a parallel array structure for efficiency and to allow easy creation of new attributes. The core attributable graph structure and the facilities for adding new attributes (e.g., boolean value of a proposition evaluated in each state) belong to the language-independent portion of the toolset. (For simplicity, only attributes of nodes (states) are shown here; edges (events) are attributed in a similar manner.)

Checking for Deadlock. We have described in Section 3.2 why checking for deadlock is not a simple matter of recognizing TICG (reachability graph) nodes with no out-edges. This simple check would suffice for representations in which internal nondeterminism (arbitrary choice by a single task) is represented by independent progress of one task (e.g., τ actions in CCS). However, representing choice in this manner is costly. In the TIG representation, a single TIG node represents a region of sequential code which may involve both internal and external nondeterminism; the "edge groups" facility of the TIG representation is used to distinguish between the deadlock potential of the internally nondeterministic choice (e.g., Ada if) and external nondeterministic choice (Ada select).

It is easy to show (by reduction from graph k-colorability) that determining whether a single TICG node represents a potential deadlock is NP-complete in the number of tasks. However, the problem size is typically very small. Moreover, avoiding the more complex check at each node by using a modeling formalism in which internal nondeterminism is represented by state transitions, as in CCS or Petri nets, results in additional size in the reachability

graph equivalent to the worst case of extra checking at each TICG node and far more expensive in space. TICG nodes are checked for deadlock in CATS after calculating their out-edges. We take advantage of the earlier computation of edges by attempting to find a set of internal choices (assignment of edge groups to TIG nodes) inconsistent with all TICG edges.

For example, a TICG representation of tasks TA and TB in Figure 7 would contain a node with a single TICG edge, labeled $[X]$. This edge would be associated with an edge group in the TIG representation of task TA; another edge group associated with the same TIG node contains a single edge labeled $\bar{Y}$. There is no TICG edge associated with this second edge. By assigning this second edge group to the TIG node (i.e., by assuming the else branch of the if statement has been taken, and task TA waits only at entry Y), we obtain an assignment of edge groups inconsistent with all TIG edges, i.e., a potential deadlock. However, if task TA is replaced by TA', then the edges labeled $\bar{X}$ and $\bar{Y}$ belong to the same edge group, and every selection of edge groups (only one choice is possible) contains an edge labeled $\bar{X}$ and is therefore consistent with the TICG edge $[X]$. Thus there is no deadlock.

Given the NP-completeness of the check at each TICG node, we cannot do better in the worst case than exhaustive enumeration of possible edge-group combinations. However, we can perform this enumeration in a heuristic order based on the intended use of TICGs as well as their typical properties. A list of thousands of potential deadlocks would be useless (we would halt the check after some small number of potential deadlock reports), so the important case is a large TICG with few or no potential deadlocks. Possible assignments are therefore enumerated in an order designed (usually) to terminate early for states that do not represent deadlock. This approach has been generally satisfactory; experience reported in Section 4 varies from an insignificant portion of total analysis time for an example with no internal nondeterminism to roughly two thirds of total analysis cost in the case of a real-world example with a great deal of both internal and external nondeterminism.

In select statements with timeout alternatives (else or or delay branches), only the timeout branch is significant in deadlock checking. Other branches are considered as separate edge groups, and both these groups and any TICG edges involving them are eliminated from consideration before the search for an assignment of edge groups to TIG nodes. Informally, this is because a process can never be deadlocked (stuck forever) without falling through to the timeout branch.

In comparing the cost of edge groups to invisible moves, we have considered only explicit enumeration of reachable states. Implicit representations such as binary decision diagrams (BDDs) have the same worst-case complexity as explicit enumeration, but their actual expense depends on the regularity of the state space rather than its size. We do not know whether edge groups would be advantageous for symbolic representations, although it would be straightforward to incorporate edge groups in a BDD representation of a transition relation. The deadlock check at each TICG node is essentially a check for satisfiability of a propositional formula, which could be encoded in a transition rule leading to a dead state.

Temporal Logic Assertions. The temporal logic supported by CATS is a propositional, branching-time logic based on the computation tree logic (CTL) of Clarke et al. [1986]. But whereas the propositions in a CTL formula refer to states, it is often more convenient to describe sequences of events (e.g., rendezvous). Our assertion language is equivalent to CTL (with a different surface syntax) except for the addition of formulae describing events. Since CTL is not defined for graphs with terminal nodes, any node without other successors is given a self-loop (which has no practical consequence since we perform deadlock checking separately).

The atomic propositions describe states (TICG nodes) or events (which label TICG edges). An atomic state formula denotes a set of nodes in the TICG, and an atomic event formula denotes a set of edges; the interpretation of atomic formulae is the responsibility of a language-specific module of CATS (which refers to the IRIS abstract syntax graph representation underlying TIG nodes [Forester 1991]). The boolean connective and denotes set intersection; or denotes set union; and not denotes set complement, relative to the universal set of all states (nodes) or all events (edges) in a TICG. The boolean formula a → b is shorthand for (not a) or b. The temporal operators with states as operands have their standard meanings in computation tree logic (see Table I). The temporal operators involving events express similar notions.

The basic event formulas are not strictly necessary. In principle one could achieve the same effect using standard CTL by transforming the reachability graph (introducing a new node for each edge), but we found it straightforward to implement alternative versions of each basic temporal operator.

Model Checking. As in the design of the reachability graph representation, a main objective in designing the temporal logic checking component of the system was to keep all language-specific aspects isolated from the rest of the model-checking system. This is accomplished by separating the processing of atomic propositions, which describe individual events (edges) and states (nodes) and thus are inherently language dependent, from the logical and temporal connectives which are independent of the attributes associated with nodes and edges (Figure 10).

A single module, isolated from the rest of the model checker, is responsible for determining which nodes and edges match a description. The current atomic proposition evaluator matches descriptions of Ada task interactions, but the design is easily extensible to recognize other states and actions of interest. For example, to detect concurrent reads and writes of a shared variable, we would use one atomic proposition to indicate TIG nodes that perform read (but not write) accesses, and another propositional variable for write access by each task.[6] If the model-checking component of CATS were to be used for a different language or model (say, Petri nets), then a suitable

[6] Although we could also extend the TIG model to represent reads and writes as events, representing their interleavings is more expensive than annotating TIG and TICG nodes.

Table I. Computation Tree Logic with Event Propositions

Surface Syntax	CTL	Meaning
eventually a	**AF**a	True of a state s iff s satisfies a or every immediate successor state t satisfies eventually a.
eventually ev		True of a state s iff, for each edge e connecting s to a successor state t, either e matches ev, or t satisfies eventually ev.
always a	**AG**a	True of state s iff s satisfies a and every immediate successor state t satisfies always a.
always ev		True of state s iff, for each edge e connecting s to a successor state t, e matches ev, and also t satisfies always ev.
potentially a	**EF**a	True of state s iff s satisfies a or, for some immediate successor state t, t satisfies potentially a.
potentially ev		True of state s iff, for some edge e connecting s to a successor state t, e matches ev or t satisfies potentially ev.
a until b	**A**(a**U**b)	True of state s if s satisfies b, or if s satisfies a and, for each immediate successor state t, t satisfies a until b.
a until ev		True of state s if s satisfies a, and also, for each edge e connecting s to a successor state t, either e matches ev, or else t satisfies a until ev.

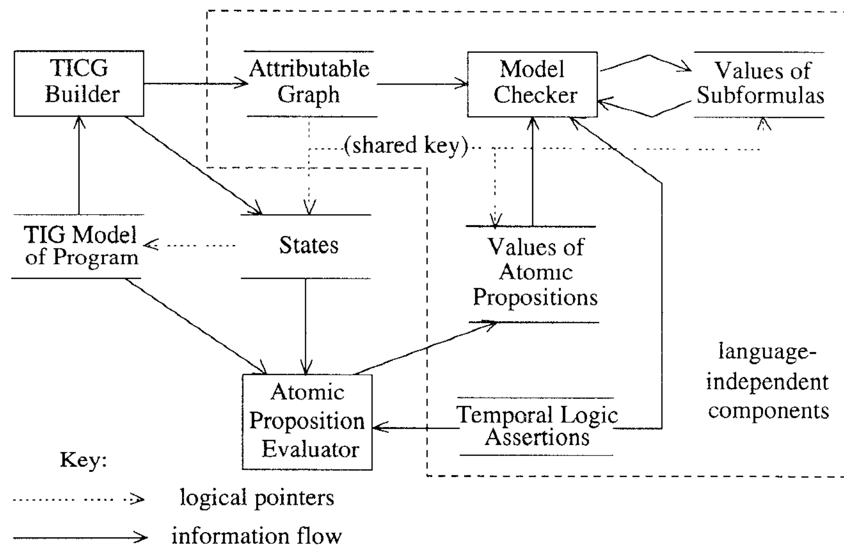

Fig 10. Language-dependent and language-independent components of the temporal logic model checker. Many components of CATS are independent of the particular system model In the initial version, the language-dependent parts (outside the dashed line) model concurrent Ada programs The model-independent portions can be reused for other reachability analysis capabilities, e.g., of executable design notations. The dotted lines represent the logical pointer relation illustrated in Figure 9. Values of atomic propositions and assertion subformulae are boolean attributes of the graph structure. Compare to the overall information flow described in Figure 8.

procedure for checking atomic propositions (e.g., markings of the net) would have to be written.

Temporal logic assertions are extracted from comments in Ada source. An assertion is checked in a bottom-up walk of its expression tree as described in Clarke et al. [1986], with each subexpression evaluating to a bit vector which is treated as a boolean attribute of the language-independent attributable graph structure; the evaluator invokes the language-dependent component only to evaluate atomic propositions.

The location of each assertion in a source program may be associated with many states in the TICG. An asserted formula is evaluated at all states, but the result is of interest only in states governed by the location of the assertion. We therefore interpret each asserted formula p as if it were an implication $\mathrm{at}(\alpha) \rightarrow p$, i.e., the proposition must be true whenever control is at location α, the location of the assertion; $\mathrm{at}(\alpha)$ is an atomic proposition evaluated by the language-dependent component.

4. EXPERIMENTS AND EXPERIENCE

We have applied CATS to a variety of canonical example problems in concurrency. Experience with one of these (dining philosophers) and with an actual, nontoy concurrent application are presented below. Three experiments were conducted on the dining philosophers example to illustrate the salient properties of CATS. First, the canonical, deadlocking version was analyzed to explore the capacity of the tool suite in terms of TICG size and to evaluate its execution speed. Second, an ordering was imposed on the resources (forks) in order to avoid deadlock. Temporal logic assertions are illustrated with this version. Third, a butler task was added to ensure freedom from deadlock and to highlight the problem of spurious error reports caused by data folding. A solution for this case, namely unrolling the loop in the butler, was analyzed. The practical potential of static analysis with CATS is demonstrated by its application to the run-time environment of Chiron [Keller et al. 1991], a highly concurrent user interface development system.

These examples exercised all components of CATS. The dining philosophers examples used the complete system, from source code translation through error reporting. The Chiron exercise required manual translation from source code to TIG models because of bugs in the compiler front-end components available to us. (In particular, we do not have a static semantics analyzer ("type checker and overload resolver") capable of handling all the features of Ada utilized by Chiron, and this analysis must be done before TIGGER can be used to build the TIGs.) Therefore, we constructed the TIG representations of Chiron tasks manually by inspection of the source code. This is a straightforward, but tedious, process for pure Ada code, as outlined in Long and Clarke [1989]. Manual extraction or annotation of synchronization structures continues to have a role for certain program features, e.g., to model interaction with the XView toolkit through Unix signals. While, in principle, translators like TIGGER can be developed for multilanguage programs and operating system facilities, in practice analysis is most likely to involve a mix of

automatically extracted and manually provided (and therefore suspect) tasks and features. In such cases it will be important to combine static concurrency analysis with run-time monitoring, so that the accuracy of manually constructed portions of models can be validated through testing.

4.1 Dining Philosophers

The dining philosophers problem is a well-known example of exposure to deadlock, and a simplified version of a significant class of problems for concurrent systems. (See Andrews [1991] for a good modern treatment.) While dining philosophers is a "toy" problem, its simplicity and regular structure are convenient for experimenting with variations on representation and analysis procedures and for illustrating aspects of the analysis tool. We assume the reader has some familiarity with the problem.

Simple Dining Philosophers. The first version of the dining philosophers considered is the classic (non-)solution, in which all n philosophers remain seated at the table, and each picks up the left fork before the right. In this version deadlock occurs when each philosopher holds one fork, and analysis is trivial, once the TICG is built. The interesting questions concern the impact of certain details of the TIG model on capacity and performance of tool components.

The dining philosophers system consists of $2n$ processes (Ada tasks), one for each philosopher and fork. The performance of the TICG builder and deadlock checker is shown in Table II in terms of the sizes of the TICGs that could be handled, and the CPU times[7] required to build them and check for deadlock.

For the *unoptimized* cases each communication (fork up, or fork down) is represented by *two* distinct interactions (beginning and end of Ada rendezvous). In general modeling a rendezvous as two synchronizations is necessary (e.g., when entry calls or shared-variable references are nested within accept statements), but it is straightforward to recognize the special case of "synchronization-only" rendezvous which can be represented by a single interaction; this is given in the table as *optimized* cases. In this example, it is also clear that the order in which tasks are initiated is irrelevant; the optimized version does not explicitly model the initiation step of each task.

The TIG model as originally described by Long and Clarke [1989] requires that all edges entering a TIG node correspond to a single region of source code; thus the entrance to a loop is often represented twice, distinguishing the first iteration from subsequent iterations. While this restriction makes some bookkeeping chores simpler, it has a cost in performance. The column

[7] These CPU times were obtained from the Unix time command (user + system), on a 64 MB Sun 4/670. They were not obtained under controlled conditions, but have been sufficiently consistent over several runs that (given the shape of the curve) more precision was not deemed useful. The CATS tools were compiled with SunAda 1.0(d). Times in tables are always for building the TICG and checking for deadlock, and do not include checking for violations of temporal logic formulas because, in every case, the cost of temporal logic model checking was dominated by the cost of building the TICG.

Table II. Dining Philosopher TICG Sizes and Times for TICG Construction and Deadlock Checking

Philosophers/ (Tasks)	*Unoptimized*			*Optimized*			*Modified TIG Model*		
	States	Edges	Time, sec	States	Edges	Time, sec.	States	Edges	Time, sec.
i. Classical deadlocking version									
2 (4)	40	56	2.2	19	28	2.4	8	10	1.4
3 (6)	268	576	2.7	84	186	2.6	26	51	1.6
4 (8)	1,792	5,168	4.2	375	1,112	3.1	80	212	2.1
5 (10)	11,744	42,440	15.5	1,653	6,130	4.7	242	805	2.2
6 (12)	76,720	332,928	246.1	7,282	32,412	220.2	728	2,910	2.9
7 (14)				32,063	166,502	5575.3	2,186	10,199	5.2
8 (16)				141,167	837,808	35773.7	6,560	34,984	26.0
9 (18)							19,682	118,089	336.8
10 (20)							59,048	393,650	8090.1
ii. With butler task									
2 (5)	157	286	2.6	58	110	2.5	28	50	1.4
3 (7)	1,951	4,968	4.7	489	1,374	3.5	154	411	2.0
4 (9)	22,221	72,356	75.8	3,794	14,204	19.7	832	2,964	2.7
5 (11)				28,686	134,045	307.5	4,474	19,925	6.8
6 (13)							24,040	128,478	55.8
iii. With unrolled butler task									
2 (5)	56	59	2.6	28	31	2.6	11	11	1.5
3 (7)	1,025	2,135	3.9	317	737	3.4	79	161	1.8
4 (9)	14,444	42,467	141.8	2,862	9,783	7.7	511	1,543	2.5
5 (11)				23,767	105,769	62.6	3,111	12,389	5.3
6 (13)				188,942	1,029,185	5410.1	18,263	90,155	47.0
7 (15)							104,679	625,873	1099.3

modified TIG model relaxes this restriction, producing more compact TIGs. These *modified* TIG models were constructed manually because we have not yet modified TIGGER or implemented a separate optimizer to produce TIGs in this form. These modifications applied to a single fork task are illustrated in Figure 11.

For the four-philosopher problem, the deadlock checker produced the report below for the single deadlocked state. It has two portions: the first is an example sequence of task interactions leading to the deadlocked state, and the second lists the interactions that each task is ready to perform. In this example, each task is only ready to perform one interaction, though in general it may be waiting on any number, e.g., all open alternatives of an Ada select statement.

```
Deadlock Violation Detected:
  Engage phil1, fork1.up
  Finish phil1, fork1.up
  Engage phil2, fork2.up
  Finish phil2, fork2.up
  Engage phil3, fork3.up
  Finish phil3, fork3.up
  Engage phil4, fork4.up
  Finish phil4, fork4.up
```

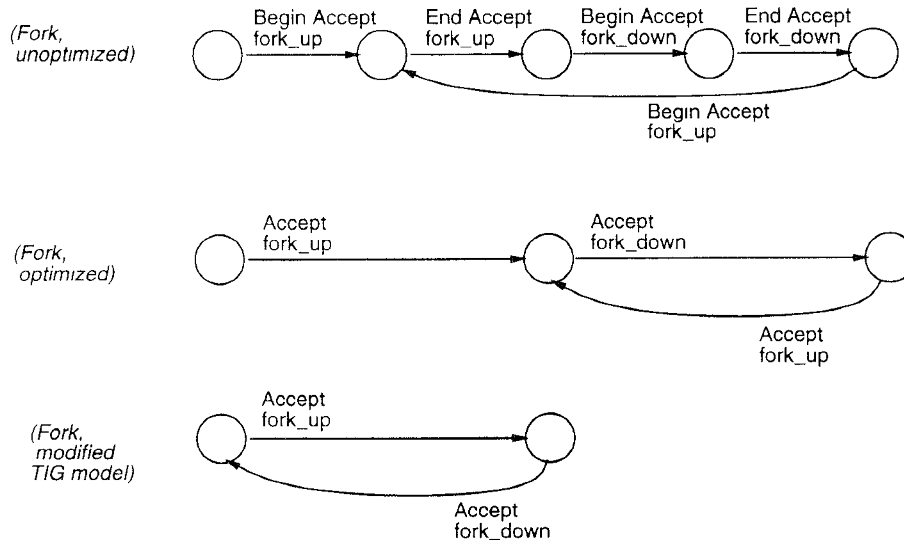

Fig. 11. TIG representations of a single fork task in the dining philosophers example. In the general case, each rendezvous must be modeled as two separate interactions as in the *unoptimized* version. Synchronization-only rendezvous can be modeled as single interactions as in the *optimized* version. TIG nodes are annotated with corresponding regions of source code (not shown). Relaxing the requirement that each node correspond to a region with a unique beginning allows the third version *modified TIG model* The impact of these representation changes is apparent in Table II.

```
phil1  attempting to engage fork4.up
phil2  attempting to engage fork1.up
phil3  attempting to engage fork2.up
phil4  attempting to engage fork3.up
fork1  attempting to accept fork1.down
fork2  attempting to accept fork2.down
fork3  attempting to accept fork3.down
fork4  attempting to accept fork4.down
```

Resource Ordering. The dining philosophers experiment was repeated, with the modification that (1) an ordering is imposed on forks and (2) each philosopher picks up its lower-numbered fork first. This is a well-known approach to deadlock avoidance and had the expected effect: the deadlocked state which is detected in the first version of the dining philosophers is absent in the version with resource ordering. The size of the TICG, time to generate it, and time to check each state for deadlock are approximately the same as for the first version of the dining philosophers problem.

Temporal Logic Checking. After freedom from deadlock was verified in the four-philosopher system with resource ordering, the following temporal logic assertion was checked:

```
(always (eventually accept fork1.up) and
        (eventually accept fork2.up) and
        (eventually accept fork3.up) and
        (eventually accept fork4.up))
```

Together with the FIFO acceptance of queued entry calls guaranteed by Ada, this property should guarantee that philosophers never starve. If the task scheduler is unfair, though, the property may not hold. A sequence of events violating the constraint is reported to the user:

```
    Engage phil1, fork1.up
    Finish phil1, fork1.up
    Engage phil1, fork4.up
 <<LOOP>>
      Finish phil1, fork4.up
      Engage phil1, fork1.down
      Finish phil1, fork1.down
      Engage phil1, fork4.down
      Finish phil1, fork4.down
      Engage phil1, fork1.up
      Finish phil1, fork1.up
      Engage phil1, fork4.up
```

If one knows that a particular implementation provides a fair scheduler, it may be desirable to suppress such reports by making a fairness assumption. Algorithms for incorporating various notions of fairness in temporal logic model checking are well known [Clarke et al. 1986] but have not yet been incorporated in the CATS model checker.

Adding a Butler. Another well-known solution to the dining philosophers problem is the addition of a "butler" task, which ensures that the number of philosophers at the table is one fewer than the number of forks. The butler task is as follows:

```
task body BUTLER is
  ROOM_OCCUPANTS : natural := 0;
  ROOM_CAPACITY : constant := PROBLEM_SIZE;
begin
  loop
    select
      when ROOM_OCCUPANTS < ROOM_CAPACITY =>
        accept ENTER;
      ROOM_OCCUPANTS := ROOM_OCCUPANTS + 1;
    or
      accept LEAVE;
      ROOM_OCCUPANTS := ROOM_OCCUPANTS - 1;
    end select;
  end loop;
end BUTLER;
```

Each philosopher is modified to enter the room before picking up either fork and to leave the room after eating. When the number of philosophers in the room reaches the limit (one less than the number of forks), the butler refuses to let another philosopher enter until some philosopher leaves, thus avoiding the deadlock situation. Addition of the butler task, combined with additional complexity in each philosopher, adds considerably to the size of the generated TICG, as shown in part ii of Table II.

The dining philosophers problem with a butler illustrates one of the problems of static concurrency analysis, namely, that it abstracts away variable values even when those values are critical to the synchronization structure of a system. In the present case, abstracting away the variable room_occupants causes an inaccurate representation of the behavior of the butler—it fails to prevent all the philosophers from entering the room simultaneously. When the TIG model of dining philosophers with a butler was analyzed, it reported a deadlock that cannot actually occur.

The role of abstraction in analysis is somewhat different in analyzing implementations (source code) than in reachability analysis of specifications and high-level designs. In the latter, abstraction can be largely left to the user; variable values and other details actually represented in the artifact to be analyzed may be presumed to be significant for the properties being analyzed. In source code of concurrent programs, the vast majority of variables should be irrelevant to synchronization structure (or else the program is certainly incomprehensible); an analysis tool may presumptively ignore them, but must accommodate selective modeling of a few critical variables.

Unrolling the Butler. One partial solution to the problem of spurious error reports is to combine static concurrency analysis with symbolic execution. Elsewhere [Young and Taylor 1988], we have described an approach in which candidate errors exposed by static concurrency analysis are used to guide a symbolic executor. An alternative approach is to use symbolic execution for partial evaluation *before* concurrency analysis. For instance, we can use symbolic execution to "unroll" the butler, that is, to trade the complexity of a counter for that of control flow. Symbolic execution to perform such a translation is similar to loop unrolling in an optimizing compiler. We have constructed a separate prototype processor for the Ada-like design language PAL that performs this transformation automatically [Yeh and Young 1991], but since the tool described here does not yet include a symbolic evaluator, we performed the unrolling manually. For a fixed value of room_limit, the butler task is unrolled into the code shown in Figure 12.[8]

The count of philosphers in the room is replaced by nesting copies of the loop body. At each step in the unrolling, the guard predicate room_occupants < room_limit evaluates either to *True* or *False*, and can be discarded. When it evaluates to *False*, the unrolling process terminates (the innermost copy of the select clause contains an accept leave immediately after the accept enter). The eventual termination of the unrolling process guarantees that the value of room_occupants will never exceed a fixed maximum value, but information in the butler task alone is not sufficient to verify that it cannot be decremented after reaching zero. For this reason, the outermost accept leave alternative is associated with an error state in the TIG representation; absence of this state in modeled executions can be checked automatically.

Unfolding in this manner must be applied very selectively; completely unfolding a variable with a large value domain can be disastrous. Since

[8] We thank Sol Shatz for pointing out an error in an earlier version of this code.

```
loop
  select                          -- ROOM_OCCUPANTS = 0;
     accept ENTER;
     loop
       select                     -- ROOM_OCCUPANTS = 1;
          accept ENTER;
          loop
            select                -- ROOM_OCCUPANTS = 2;
               accept ENTER;
          -- etc. until ROOM_OCCUPANTS = ROOM_LIMIT;
               accept LEAVE;
            or
               accept LEAVE;
               exit;
            end select;
          end loop;
       or
          accept LEAVE;
          exit;
       end select;
     end loop;
  or
     accept LEAVE;    -- error: ROOM_OCCUPANTS = -1;
     exit;
  end select;
end loop;
```

Fig. 12. "Unrolled" butler task. In practice it is more convenient to unroll the graph model rather than the source code, but the effect is the same.

human judgment seems necessary in this modeling decision, we envision (but have not yet implemented) a facility by which the user indicates which variables should be unfolded, and to what extent. We have implemented variable unfolding in a separate prototype tool [Yeh 1993; Yeh and Young 1993; 1994] and see no technical obstacle to supporting better user control.

A TIG representation of the unrolled version of the butler was constructed, and the analysis was repeated; see Table II part *iii*. Note that even though the unrolled version of butler is larger than the original version, it does not cause the TICG to become larger. This will always be the case when an eliminated variable was used to keep track of the states of other tasks, since the value of the variable is purely a function of those states. This is clearly the best case for selective unrolling with partial evaluation.

Discussion. Realistic problems (like the Chiron example next) are more relevant than artificial models like dining philosophers to assessing overall performance, but variations on the dining philosophers problem illustrates the performance impact of certain modeling decisions. First, it is clear that exploiting special cases (like synchronization-only rendezvous) and otherwise avoiding unnecessary complexity in modeling task structures (the optimized TIG model) are essential; they make a significant difference in the size of the system that can be analyzed in a reasonable time, or the size of parcels that larger systems must be broken into before analysis will be practical. Our experience with other examples, real and artificial, bears this out. Second, the "unrolled" version shows that, in some cases, increasing the accuracy of analysis through symbolic execution can actually improve performance.

4.2 Chiron 1.2 Run-Time

The Chiron user interface development system (UIDS) [Keller et al. 1991; Taylor and Johnson 1993] represents a modern, moderate-size system. Chiron 1.2 comprises roughly 10^5 source lines of code, designed and implemented over a two-year period with the equivalent of five individuals working at any one time. The run-time environment includes a reasonably complex task structure and was not "designed to be analyzed" (it is not a toy example).

To aid in understanding the significance of the CATS analysis of Chiron, we provide a brief overview of Chiron's purpose and structure.

The Chiron UIDS was built to address concerns of cost, maintainability, and sensitivity to changes in the development and maintenance of user interfaces for large applications. Chiron provides a series of interface layers that limit the propagation of effects from changes within the various layers. To separate application code from user interface code, user interface agents called *artists* are attached to selected abstract data types belonging to the application. Each artist implements a separate user interface, specifying logical appearance and behavior. Operations on the abstract data types within the application trigger user interface activities within the artist by means of *dispatchers* which are transparently (to the application) wrapped around the ADTs by a preprocessor at compile-time. At run-time, the dispatcher intercepts all operation calls on the ADT and notifies each of the artists associated with that ADT of the operation. Artists may also respond directly to events initiated by user(s) and inform the application of necessary changes through the dispatchers.

Chiron also provides insulation between the user interface layer and the underlying system; artist code is written in terms of abstract depiction

libraries that insulate the code from the specifics of particular windowing systems and toolkits. Calls by the artists to create/update depictions are transmitted over an interprocess communication link to a Chiron server. The server implements, in C++, an inheritance hierarchy of abstract depiction classes. Calls from artists thus create and manipulate instances of these classes; the server maintains an "abstract depiction" of each artist's user interface. (Thus while most of the code of interest in the concurrency analysis is Ada, calls from Ada tasks through the abstract depiction hierarchy must be followed to determine all possible interactions.) The server also oversees rendering of the abstract depiction to a concrete depiction, using a windowing system and toolkit.

Concurrency is pervasive in the Chiron architecture. Inside an application there can be multiple execution threads; there is no requirement for a user interface listening/dispatching routine to have exclusive control. Multiple artists can be attached to a single application abstract data type, providing alternative forms of access by a single user or coordinated access and manipulation by multiple users.

The key run-time components of Chiron are thus (1) the Chiron client, which consists of the application plus all artists, and (2) the Chiron server, which is responsible for maintaining and rendering the logical depictions created by the artists.

The task and package structure of a generic Chiron run-time environment is shown in Figure 13 using a simplified variation of Buhr diagrams [Buhr 1984]. Rectangles depict packages, nested as drawn. Parallelograms represent tasks, with arrows from callers to entries. Dots beside arrowheads represent guarded entry calls. Shadowed parallelograms with dashed lines indicate optional multiple task instances. While this figure is intended to highlight task interactions, the ADT is not a task, and the dashed arrows leading to it are simply procedure calls. Not shown in Figure 13 is the capability for multiple ADTs, which involves multiple instantiations of the Dispatcher package and interactions among dispatchers, the application, and artists.

All ADT operations are performed by either the main application task, labeled Start_App, or by artists in response to user requests (events). An artist has a separate task entry for each operation. Because there is no distinction among these various operations from a task interaction viewpoint, they are lumped together as a single "artist actions" entry in our model. Access to the ADT by Start_App and the artists is serialized by a concurrent read, exclusive write lock mechanism, implemented as the Control task in the Dispatcher_Controller package. A two-level package interface to the Control task handles the rather intricate dispatcher details such as notifying all other artists of a change to an ADT by any one artist or the main task.

As noted above, due to limitations in the front-end language processing tools available to us, the Chiron task interaction graphs were necessarily constructed by hand. Chiron also contains two examples of task interactions that are typically incorporated manually: Unix signals and interlanguage procedure and task entry calls. The Unix signals are tied to task entries via

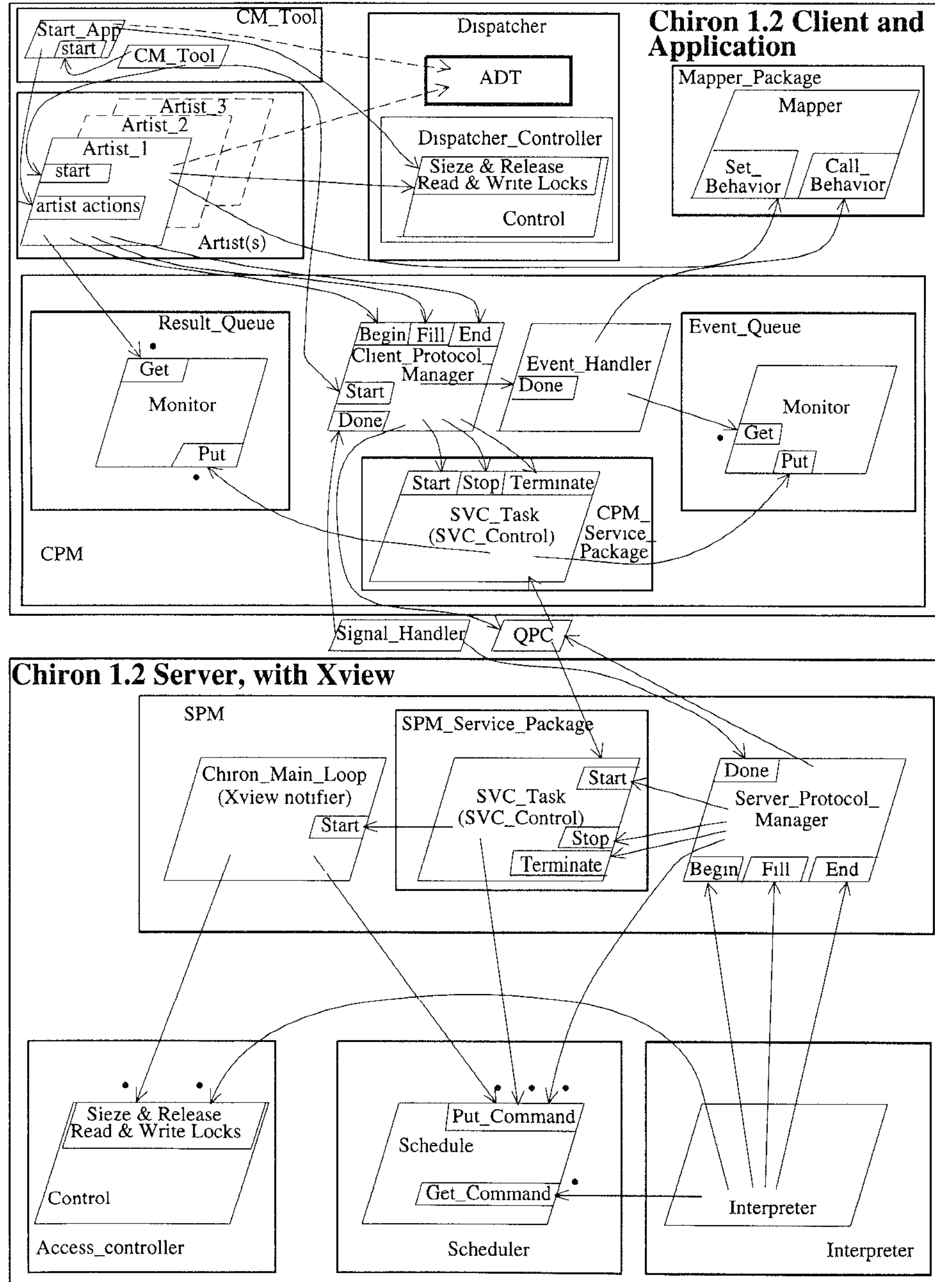

Fig. 13. Chiron 1.2 task and package structure.

representation clauses and therefore are modeled as task entry calls. The several Ada task entry calls in the C++ hierarchy were readily identified because the interlanguage interfaces are explicit in the code.

Another form of analyst intervention is parceling, or partitioning into components that are amenable to analysis. Parceling removes interleavings of unrelated tasks and consequently is more effective with decreasing component interaction (loose coupling). Chiron provides a good example of the applicability of this approach: the design separates it into two operating system processes, the server and the client/application. Therefore, a natural parcel boundary is along this split. The two "in-between" tasks shown in Figure 13 provide a higher-level interface for handling the operating system signals and all interprocess communication. The relevant portions of these tasks were included in the analysis on both sides of the split.

TICG Sizes. As shown in Table III, the concurrency state graphs are of a very manageable size. The single-artist client/application module, with 12 tasks, is analyzed in less than half a minute. Adding a second artist increases the TICG size greatly, since each artist interacts nearly independently with the main Start_App, Result_Queue_Monitor, Mapper, and Dispatcher. An analysis with two artists takes over 40 minutes of CPU time.

Roughly one-third of the execution time was spent building the TICGs and two-thirds checking for deadlock. This ratio varies with the particular TICGs analyzed; as explained in Section 3.3.1, the check of each concurrency state may at worst require time exponential in the number of edge groups per TIG node. In contrast to the dining philosophers problem, Chiron interfaces involve a mix of internal and external nondeterminism which must be analyzed in the deadlock-checking step.

Deadlock States. The Chiron 1.2 design includes modifications to the original Chiron 1.0 design to avoid deadlock. The following discussion results from our analysis of Chiron 1.0, which illustrates better the information provided by the deadlock checker.

The deadlock check of Chiron 1.0 reported 24 potential deadlocks on the server side. Further investigation revealed that none were in fact feasible. The server had a shutdown mechanism which was initiated by the Signal_Handler task after catching a user-originated kill signal from the operating system. Of the potential deadlocks, 22 were after the Signal_Handler had started the shutdown. This was expected because we did not model terminate and because some tasks waited on that alternative. To verify the shutdown hypothesis, the Signal_Handler was disabled by disconnecting the edge representing the shutdown rendezvous from its initial TIG node. The 22 previously reported shutdown deadlocks vanished, confirming their origin. The two remaining deadlock reports were manually inspected and were determined to be infeasible. Both involved variables in guards to select statement alternatives in which all the alternatives were guarded. Inspection of the program logic revealed that at least one alternative must always be open.

Table III. Chiron 1.2 TICG Sizes and Construction/Deadlock Check Times

Component	Tasks	States	Edges	Time (sec.)
server	8	9,494	35,612	17 1
client/application, one artist	12	8,055	36,003	19.2
client/application, two artists	13	110,674	547,846	2426.8

On the client/application side (with one artist task), 17 potential deadlocks were originally reported.[9] Fifteen were due to the same shutdown mechanism as on the server side. The remaining two represented actual deadlock states, and had been previously observed in the operation of Chiron. These were caused by a race condition on a variable that tracked the number of tasks (artists and application) processing dispatcher calls. The dispatcher was redesigned to eliminate this anomaly; there are now 3 reported deadlocks for the single-artist client and 18 for the two-artist client, all of which are artifacts of not modeling the shutdown procedure.

Other Tasking Anomalies. While constructing the TIGs for each of the tasks, we noted another synchronization anomaly, which was not previously known to the Chiron developers. It involved a race condition on an unprotected variable in the Dispatcher_Controller, a variable writable both by the Control task and by others outside the package through access procedures. Access to the global variable was serialized through the Control task, and analysis was carried forward on the modified system.

5. DISCUSSION

Reachability analysis capabilities like those provided in CATS are limited in several ways. They are applicable only to analysis of synchronization structure, and not functional correctness, performance, robustness, etc. Moreover, even analysis of synchronization structure has limitations. While the gross size of a program to be analyzed (in lines of code or some other measure) is not an issue, analysis is typically limited to small collections of tasks on the order of a dozen. (Hierarchical composition of these small collections is possible; Yeh and Young [1991] describe encouraging progress in this direction.) Accuracy of the analysis is limited by elision of details, including control flow determined by variables whose values are not completely modeled.

We stress, however, that CATS is not intended to be used as a standalone analysis tool. Reachability analysis is most useful as one among several

[9] This format of deadlock reports is described in Section 4.1.

analysis and testing techniques in a software development environment. Major design decisions in CATS were shaped by a combination of performance, applicability, and (especially) modularity constraints in this context.

Integrated Application Strategy. Our long-term goal is to support an extensive collection of analysis and testing techniques, integrated to capitalize on the strengths of the individual techniques and compensate for their weaknesses. We aim for a level of automation in coordinating analysis activities at least equal to the level of automation provided for compilation in contemporary environments.

To achieve this level of integration and automation, analysis at each point in development must be composed of several steps, viz., (1) examine the current structure of the system being analyzed and a repository of asserted properties which have been established or alleged for portions of the system; (2) determine which analysis techniques are currently applicable; and (3) attempt to apply them. "Results" of an analysis may include indications of why particular techniques could not be applied and suggestions for guiding reanalysis or restructuring the system. Both results and dependencies among results must be stored for use in later reanalyses.

CATS is suited to such an integrated strategy, and the tool components are organized to facilitate it. Several simple checks of a system and its asserted properties can be used to determine applicability of CATS. Static concurrency analysis in CATS can both make use of the results of other techniques and produce results useful to other testing and analysis tools.

Static concurrency analysis is applicable to systems with static task structures (without dynamic creation of unbounded numbers of tasks and without dynamic identification, e.g., arrays of tasks). When these conditions are met, CATS components can be combined in an appropriate configuration depending on whether only deadlock checking, only safety properties, or both safety and liveness properties should be checked; this can be determined simply by inspecting assertions in the code. In the absence of detailed user guidance in partitioning a system into modules small enough for analysis, scope structure together with worst-case estimates of TICG size may be sufficient to determine whether tasks within a program unit are likely to be successfully analyzed.

Results of flow analyses would be particularly useful in static concurrency analysis. For example, if CATS is to check for potential data races, variables that can be accessed by more than one task should be identified in a prior interprocedural data flow analysis. Similar simple flow analyses may eliminate or transform some tasks before analysis, e.g., a simple server task acting as a monitor [Hoare 1974] to protect a data structure can usually be elided. The required interprocedural analysis is similar to that used in optimizing Ada compilers for recognizing monitor clusters [Hilfinger 1982] and can be extended to elide hierarchies of server tasks iteratively.

TIG and TICG structures can be stored for further use, both for incremental composition with TIG or TICG representations of other parts of a system and for use in other analysis techniques. Incremental composition is dis-

cussed elsewhere [Yeh and Young 1991]; here we briefly consider some other uses of TICGs.

Formal Verification. Formal verification can in principle provide a high level of assurance for a wide variety of program properties, but for the foreseeable future the role of formal verification will be limited to algorithms and high-level designs, or to particularly critical properties of very small systems or components. If formal verification is to be used at all, it must coexist with other analysis and testing techniques. Static concurrency analysis may be a useful adjunct to formal verification, e.g., checking for conformance between verified high-level designs and actual code at the granularity of a few tasks at a time. A simple, but potentially useful, application of static concurrency analysis is to mechanize trivial proofs of noninterference or to alert the verifier when possible process interleavings make a nontrivial cooperation proof necessary. A TICG structure, together with a record of variables potentially accessed in each basic block, can provide the required information.

Testing. A basic premise of most approaches to software testing is that programs are characterized by input-output pairs, and that output is completely determined by input. In concurrent software, output is usually not completely determined by input (because of apparent nondeterminism in process schedulers), and output values may not even be the most important property of software. Effective testing strategies for concurrent software must include oracles for accepting or rejecting individual test runs, stopping rules (adequacy criteria) for deciding when concurrent software has been tested "enough," and methods for creating test cases. The TICG structure produced by CATS is potentially useful for adequacy criteria.

In particular, Taylor et al. [1992] describe a method for using concurrency graph models of programs in defining structural test coverage criteria. While they used a flowgraph model for their definition of a test case coverage hierarchy, the results can be transferred to a TICG model. Additionally, Weiss [1988] has developed a formal theory for reasoning about test coverage based on representing concurrent programs as a set of simulating sequential programs termed *serializations*, which are essentially similar to paths in a concurrency graph.

The structural coverage criteria defined to date are likely to be impractical, given the difficulty of determining which paths (or serializations) are actually feasible, but one can devise pragmatic variants. For instance, one might explore only the portion of the state space corresponding to observed executions, in order to recognize points where a nondeterministic choice could have led to a different execution history for the same test data. A configuration of CATS providing a suitable, directed, exploration component could be constructed by replacing a single small module responsible for search strategy. Similar directed exploration techniques have been proposed as aids to debugging [Taylor 1984] and incorporated in a debugging tool that combines static

and dynamic analysis [Raither et al. 1990]. Techniques for forcing a concurrent or distributed program through a chosen sequence of nondeterministic choices have been described by LeBlanc and Mellor-Crummey [1987] and by Tai et al. [1991].

6. RELATED WORK

Variations on the basic reachability analysis technique used in CATS have a long history, with most early work in the field of communication protocol analysis [Sunshine 1981]. Many concurrency analysis tools have been developed for high-level designs, algorithms, and protocols (e.g., Fernandez et al. [1992] and Holzmann [1991]), but few have been developed for programs (excepting code generated from protocol descriptions). While similar techniques are applicable at both levels, the pragmatic considerations are different. In particular, details irrelevant to synchronization structure are unavoidably present in source code, and the relevant portions of the program must be identified and extracted.

The most common approach to integrating concurrency analysis tools in a development environment is to translate software artifacts into a fixed modeling or analysis formalism. Examples include translation of Ada to Petri nets [Shatz et al. 1990], translation of Lotos to Petri nets [Fernandez et al. 1992], and translation of CCS process graphs (with labeled edges and unlabeled nodes) into Kripke structures (with unlabeled edges and labeled nodes) to interface with a temporal logic model checker [Cleaveland et al. 1990]. This is attractive when a small translation effort permits reuse of existing analysis tools, and the performance of those tools may conceivably make up for their lack of specialization.

Translation is tool combination at a coarse level; CATS is designed to support combination and reconfiguration at a finer grain, e.g., coverage testing with the directed exploration techniques discussed in Section 5. We have argued that static concurrency analysis can be integrated with other analysis and testing techniques and tools, and we have described some possible combinations. An important thread of future work will be to produce additional working examples of integration, particularly those involving dynamic analysis of time-sensitive parallel systems.

Additionally, we intend to exploit the modular structure of CATS and generalize it further to analyze systems in a mix of design and implementation notations.

Computational expense is a continuing concern in reachability analysis techniques for concurrent software. Several approaches exploit regularity in state spaces to speed enumeration and reduce storage. McDowell [1989] has implemented a static analyzer that merges a set of related states. Helmbold and McDowell [1991] present a generalization called "entity folding." Several approaches exploit regularity in *sequences* of states, noting that many interleavings of independent events are redundant. Godefroid's *sleep set* avoids exploring all interleavings of independent events; all states are still enumer-

ated, but not all of them must be stored [Godefroid et al. 1993]. Valmari [1990] recognizes a particular kind of independence between events (called "stubbornness," or sometimes "persistence") and elides both redundant event sequences and states. Our aim in CATS has been to obtain adequate performance in combination with other important constraints, and our current implementation is already adequate for analysis of systems well beyond the comprehension of an unaided programmer. As it happens, though, the modular structure of CATS should make incorporating some of these performance enhancements reasonably straightforward.

Although exponential lower bounds apply to all possible algorithms for deciding exposure to deadlock and other interesting properties of concurrent programs, symbolic techniques may better exploit regularity in some cases. The constrained expression approach represents possible interactions between tasks by a set of inequalities relating counts of event occurrences [Avrunin et al. 1986; Dillon et al. 1988]. Binary decision diagrams (BDDs) have been used to represent a transition relation by a characteristic predicate which can be combined with a BDD representation of properties to be verified [Burch et al. 1990]. In place of the combinatorial explosion in states in enumerative approaches, symbolic approaches may suffer an explosion in the number or size of symbolic descriptions of a system; in the case of BDDs, both worst-case and average-case sizes are exponential, and most BDD-based tools rely on careful tuning for each individual problem. The intuitive expectation that symbolic and enumerative approaches will each be capable of tackling some problems that the other cannot is backed by the experience of Ratel et al. [1991], who implemented both approaches to analyzing safety properties of *LUSTRE* programs.

An alternative direction is to sacrifice accuracy (further) for performance. Masticola and Ryder [Masticola 1993; Masticola and Ryder 1991] have developed a polynomial-time deadlock detection algorithm based on data flow analysis. In order to achieve this complexity, reachability is safely overestimated at the expense of precision: spurious deadlocks may be reported. Initial experimental results with some nontoy programs are encouraging, though the tasking behavior of these programs is simple. It is unlikely that weaker techniques can replace static concurrency analysis, but as we have discussed earlier they may be quite useful as a filter applied before more-accurate and expensive analysis techniques.

Ultimately no global analysis, whether symbolic or enumerative, will be useful for detecting faults in large-scale software systems. Performance enhancements postpone the inevitable. Practical approaches must parcel analysis into tractable subproblems, and preferably into subproblems that mirror the modular structure of the system under test. Layering and projection [Lam and Shankar 1984] are often used for decomposing models of communication protocols along functional lines. CATS does not currently provide automated support for dividing a system into modules for analysis or composing analysis results, although a system that uses program scope structure and communication topology to parcel an analysis automatically is reported in Yeh [1993]

and Yeh and Young [1991; 1993]; this approach will be incorporated in a future version of CATS.

7. CONCLUSION

The challenge in designing CATS was to meet and balance multiple and sometimes conflicting objectives. Any one of these dimensions could have been improved by sacrificing others. CATS analyzes Ada source code for exposure to deadlock as well as user-specified properties in the form of temporal logic assertions and reports potential errors in terms of source-level sequences of events. Nonetheless, Ada-specific processing and data structures are cleanly separated from language-independent aspects throughout the system, and major parts of both are reusable. CATS is composed of many small components for flexibility and to support integration with other tools and techniques. Its performance and capacity are sufficient that we do not feel compelled to sacrifice this organization.

Experience with CATS, though preliminary, makes us optimistic about the role static concurrency analysis can play in combination with other techniques for analyzing concurrent software. While the well-known state explosion problem makes global analysis intractable, task interactions in subsystems well beyond the comprehension of unaided programmers can be exhaustively analyzed in a few minutes, and the hierarchical techniques reported in Yeh [1993] and Yeh and Young [1991; 1993] will permit future versions of CATS to analyze considerably larger collections of tasks. Our experience applying CATS to an application built using the Chiron user interface development system suggests that analysis of natural modules of real programs can be both practical and useful.

REFERENCES

ANDREWS, G. R. 1991. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, Calif.

APT, K. R. 1983. A static analysis of CSP programs. In *Proceedings of the Workshop on Program Logic* (Pittsburgh, Pa.).

AVRUNIN, G. S., DILLON, L. K., WILEDEN, J. C., AND RIDDLE, W. E. 1986. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Softw. Eng. SE-12*, 2 (Feb.), 278–292.

BAETEN, J. C M. AND VAN GLABEEK, R. J. 1987. Another look at abstraction in process algebra. In *Proceedings of the 14th International Colloquium on Automata, Languages, and Programming (ICALP)* (Karlsruhe, Germany, July). Lecture Notes in Computer Science, vol. 267. Springer-Verlag, Berlin, 84–94.

BERGSTRA, J. A. AND KLOP, J. W. 1984. Process algebra for synchronous communication. *Inf. Control 60*, 109–137.

BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. 1984. A theory of communicating sequential processes *J. ACM 31*, 3 (July), 560–599.

BUHR, R. J. A. 1984. *System Design with Ada*. Prentice-Hall, Englewood Cliffs, N.J.

BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 428–439.

CLARKE, E. M., BROWNE, M. C., EMERSON, E. A., AND SISTLA, A. P. 1985. Using temporal logic for automatic verification of finite state systems. In *Logics and Models of Concurrent Systems*, K R. Apt, Ed Springer-Verlag, Berlin, 3–26.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Trans. Program. Lang. Syst. 8*, 2 (Apr.), 244–263.

CLEAVELAND, R., PARROW, J., AND STEFFEN, B. 1990. A semantics-based verification tool for finite-state systems. In *Protocol Specification, Testing, and Verification*. Vol. 9. North-Holland, Amsterdam, 287–302.

DILLON, L. K., AVRUNIN, G. S., AND WILEDEN, J. C. 1988. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Trans. Program. Lang. Syst. 10*, 3 (July), 374–402.

FERNANDEZ, J.-C., GARAVEL, H., MOURNIER, L., RASSE, A., RODRIGUEZ, C., AND SIFAKIS, J. 1992. A toolbox for the verification of LOTOS programs. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia). IEEE Computer Society Press, Los Alamitos, Calif., 246–259.

FORESTER, K. 1991. IRIS-Ada reference manual. Arcadia Tech. Rep. UM-90-07, Univ. of Massachusetts, Amherst, Mass.

GODEFROID, P., HOLZMANN, G. J., AND PIROTTIN, D. 1993. State space caching revisited. In *Proceedings of the 4th Workshop on Computer-Aided Verification* (Montreal, Quebec, July). Lecture Notes in Computer Science, vol. 663. Springer-Verlag, Berlin, 175–186.

HELMBOLD, D. P. AND MCDOWELL, C. E. 1991. Computing reachable states of parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging. ACM SIGPLAN Not. 26*, 12, 76–84.

HENNESSY, M. 1988. *Algebraic Theory of Processes*. MIT Press Series in the Foundations of Computing. The MIT Press, Cambridge, Mass.

HILFINGER, P. N. 1982. Implementation strategies for Ada tasking idioms In *Proceedings of the AdaTEC Conference on Ada*. ACM, New York, 26–30.

HOARE, C. A. R. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct.).

HOLZMANN, G. J. 1991. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J.

KANELLAKIS, P. C. AND SMOLKA, S A. 1990. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput. 86*, 43–68.

KELLER, R. K., CAMERON, M., TAYLOR, R. N., AND TROUP, D. B. 1991. User interface development and software environments: The Chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering* (Austin, Tex., May). IEEE Computer Science Press, Los Alamitos, Calif., 208–218.

LADNER, R. E. 1979. The complexity of problems in systems of communicating sequential processes. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing* (Atlanta, Ga., April). ACM, New York, 214–223.

LAM, S. S. AND SHANKAR, A. U. 1984. Protocol verification via projections. *IEEE Trans. Softw. Eng. SE-10*, 4 (July), 325–342.

LEBLANC, T. J. AND MELLOR-CRUMMERY, J. M. 1987. Debugging parallel programs with instant replay. *IEEE Trans. Comput. C-36*, 4 (Apr.), 471–482.

LONG, D. L. AND CLARKE, L. A. 1989. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering* (Pittsburgh, Pa., May). IEEE Computer Society Press, Los Alamitos, Calif., 44–52.

LONG, D. AND CLARKE, L. A. 1991. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)* (Victoria, British Columbia, October). ACM Press, New York, 21–35.

MANDRIOLI, D., ZICARI, R., GHEZZI, C., AND TISATO, F. 1985. Modeling the Ada task system by Petri nets. *Comput. Lang. 10*, 1, 43–61.

MASTICOLA, S. P 1993. Static detection of deadlocks in polynomial time. Ph.D. thesis, Rutgers Univ., New Brunswick, N.J.

MASTICOLA, S. P. AND RYDER, B. G. 1991. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*. *ACM SIGPLAN Not. 26*, 12, 97–107.

MCDOWELL, C. E. 1989. A practical algorithm for static analysis of parallel programs. *J. Parallel Distrib. Comput. 6*, 515–536.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall, London.

MORGAN, E. T. AND RAZOUK, R. R. 1987. Interactive state-space analysis of concurrent systems. *IEEE Trans. Softw. Eng. SE-13*, 10 (Oct.), 1080–1091.

PETERSON, J. 1981. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, N.J.

RAITHER, B., CAILLET, J.-F., AND DE SEZE, P. 1990. IDEFIX: A tool for debugging Ada tasks in a real-time environment. *Technique et Science Informatiques 9*, 2, 150–156.

RATEL, C., HALBWACHS, N., AND RAYMOND, P. 1991. Programming and verifying critical systems by means of the synchronous data-flow language LUSTRE. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems* (New Orleans, La., Dec.). ACM, New York, 112–119.

RAZOUK, R. R. 1987. A guided tour of P-NUT. Tech. Rep. 86-25, Univ. of California, Irvine, Calif.

SHATZ, S. M., MAI, K., BLACK, C., AND TU, S. 1990. Design and implementation of a Petri net based toolkit for Ada tasking analysis. *IEEE Trans. Parallel Distrib. Syst. 1*, 4 (Oct.), 424–441.

SMOLKA, S. A. 1984. Analysis of communicating finite state processes. Ph.D. thesis, Tech. Rep. CS-84-05, Dept. of Computer Science, Brown Univ., Providence, R.I.

SUNSHINE, C. A., (ED.). 1981. *Communication Protocol Modeling*. Artech House, Dedham, Mass.

TAI, K. C., CARVER, R. H., AND OBAID, E. E. 1991. Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. Softw. Eng. 17*, 1 (Jan.), 45–63.

TAYLOR, R. N. 1983a. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica 19*, 57–84.

TAYLOR, R. N. 1983b. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM 26*, 5 (May), 362–376.

TAYLOR, R. N. 1984. Debugging real-time software in a host-target environment. *Technique et Science Informatiques 3*, 4, 281–288.

TAYLOR, R. N. AND JOHNSON, G. F. 1993. Separations of concerns in the Chiron-1 user interface development and management system. In *Proceedings of the Conference on Human Factors in Computing Systems* (Amsterdam, Apr.). ACM, New York, 367–374.

TAYLOR, R. N., LEVINE, D. L., AND KELLY, C. D. 1992. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng. 18*, 3 (Mar.), 206–215.

VALMARI, A. 1990. A stubborn attack on state explosion. In *Computer-Aided Verification, 2nd International Conference Proceedings*. Lecture Notes in Computer Science, vol. 531. Springer-Verlag, Berlin, 156–165.

WAMPLER, G. K. 1985. Static concurrency analysis of Ada programs. Master's thesis, Univ. of California, Irvine, Calif.

WEISS, S. N. 1988. A formal framework for the study of concurrent program testing. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, Canada, July). ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

WILEDEN, J. C., WOLF, A. L., FISHER, C. D., AND TARR, P. L. 1988. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of ACM SIGSOFT '88: 3rd Symposium on Software Development Environments* (Boston, Nov.). ACM, New York, 130–142.

YEH, W. J. 1993. Controlling state explosion in reachability analysis. Ph.D. thesis, Dept. of Computer Sciences, Purdue Univ., West Lafayette, Ind.

YEH, W. J. AND YOUNG, M. 1991. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)* (Victoria, British Columbia, Oct.). ACM Press, New York, 49–59.

YEH, W. J. AND YOUNG, M. 1993. Compositional reachability analysis of Ada programs using process algebra. Tech. Rep., Software Engineering Research Center, Dept. of Computer Sciences, Purdue Univ., West Lafayette, Ind.

YEH, W. J. AND YOUNG, M. 1994. Redesigning tasking structures of Ada programs for analysis: A case study. *Softw. Testing Verif. Reliab. 4*, 223–253.

YOUNG, M. 1988. How to leave out details: Error-preserving abstractions of state-space models. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, Canada, July). 63–70.

YOUNG, M. AND TAYLOR, R. N. 1988. Combining static concurrency analysis with symbolic execution. *IEEE Trans. Softw. Eng. 14*, 10 (Oct.), 1499–1511.

YOUNG, M., TAYLOR, R. N., FORESTER, K., AND BRODBECK, D. A. 1989. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)* (Key West, Fla., Dec.). *ACM SIGSOFT Softw. Eng. Not. 14*, 8, 200–209.

Received December 1993; accepted August 1994