

# A Query Language for Automated General Analysis of Concurrent Ada Programs

C. Black, S. M. Shatz  
Concurrent Software Systems Laboratory  
Department of Electrical Engineering and Computer Science  
University of Illinois at Chicago

S. Tu  
Department of Computer Science  
University of New Orleans

## Abstract<sup>1</sup>

It is generally accepted that design, implementation and analysis of concurrent-software systems are very difficult activities that need support by automated techniques and tools. This is especially true for analysis, which is typically based on use of some type of formal model of the program and associated analysis of this model. In this paper, we present and discuss an Ada tasking query language called TQL. TQL supports very general forms of analysis and defines a user interface that is independent of the particular formalism upon which the analysis framework is build (e.g., Petri nets). A software engineer uses a TQL-based analysis session to ask questions about the potential tasking behavior of a program, including behaviors related to entry calls, accepts, and rendezvous. A prototype implementation that uses two analysis methods is discussed. Also, an annotated transcript of a simple TQL analysis session is presented and explained.

## 1. Introduction

A concurrent program consists of a number of processes that communicate with one another to synchronize their activity and to exchange data. These processes can be executed by different processors and thus form a

---

This work was supported in part by grants from the Office of Naval Research (ONR) under grant number N00014-90-J-1446 and NSF under grant numbers CCR-9113569 and CCR-9321743.

An earlier version of this paper appears in Proceedings of the International Conference on Distributed Computing Systems, June 1992.

distributed program. The subtle and often unexpected interactions that occur in concurrent programs make understanding and analysis of this type of software very difficult.

In this paper, we are interested only in static analysis schemes. Most static analysis schemes derive some type of formal model of the program and generate a state space which tends to be very large and by necessity must be evaluated using computer tools. This approach has the desirable property of being able to identify program behaviors that are independent of particular execution environment factors like process scheduler characteristics and distributed network configurations. One of the earliest static analysis methods was Taylor's concurrency history algorithm [1]. Taylor's algorithm traces the control flow of a source program's tasks and generates a series of concurrency states (system snapshots) that define all possible syntactically specified communication and synchronization points in the program. This approach was refined and developed as a prototype system called CATS (Concurrency Analysis Tool Set) [2]. Another static analysis approach translates a program's design into a "constrained expression" [3]. This method, like the Petri net invariant method of [4], can often give very good performance for special-case analysis queries like deadlock detection. Shatz et al. defined and implemented a toolkit that supports general analysis of concurrent software using a Petri net framework [5]. This method is oriented toward Ada tasking and defines a system called the Tasking Oriented Toolkit for the Ada Language, TOTAL [6] (see Figure 1). One advantage (and intent) of the TOTAL system is that it can benefit from existing Petri net theory and tools. For example, we have examined Petri net reduction techniques for deadlock analysis [7].

One thing that is generally lacking in most concurrent software analysis research is consideration for a user interface that can support expression of general analysis queries in a way that is independent of the particular formalism (program model) supporting the analysis approach. Independence of analysis queries and the analysis formalism is desirable so that users need to understand neither details of the model used to represent a tasking program nor the means for interpreting tasking-oriented queries in terms of that model. Furthermore, expression of queries in a query language provides an opportunity to automate query processing by integrating various analysis strategies into an interpreter for the query language. In this paper, we introduce an Ada tasking query language called TQL that has been designed to support a very broad set of analysis issues. We have implemented a TQL interpreter that can selectively apply Petri net analysis techniques defined by the TOTAL system.

TQL's design carefully hides from the user the formalism upon which the analysis framework is built (e.g., Petri nets). TQL defines a language by which queries of Ada interactions themselves can be expressed. The only other tasking-type language that we are aware of is TSL (Task Sequencing Language) [8, 9], designed to specify sequences of tasking interactions that can be checked against actual tasking behavior by a run-time monitor. TSL syntax specifying a tasking interaction sequence can be interpreted as specifying a program state, as does TQL syntax, but TSL syntax is designed to express constraints used by a run-time monitor and is less suitable for interactive use. TQL provides support for the interactive user, who formulates new queries as the analysis session proceeds. A software engineer uses a TQL-based analysis session to ask questions about the potential tasking behavior of a program. This can entail checking that certain desirable states or events are possible (e.g., a rendezvous between a producer task and a buffer task) or that certain states or events are not possible (e.g., two writer tasks writing at the same time in a readers/writers problem).

Our major focus is identifying capabilities to express states that are of interest in understanding a tasking program's behavior and showing how these capabilities can be integrated into a concurrency analysis query language. The query language can be thought of as operating on a virtual state space, where a state defines the tasking-related actions of the program's constituent tasks at some point in time. This is basically equivalent to Taylor's notion of a concurrency state. We will discuss an implementation of TQL that is based on using a Petri net reachability graph as a representation of a program's state space. In principle, different analysis queries require different information from the program's state space. Since it is well recognized that a concurrent system's state space size grows exponentially as you add system components (i.e., the so-called state explosion problem), it is important to consider optimized (reduced) models. For example, an optimized model for a particular query should *only* contain the information necessary to answer that query. In this paper, we show that some model optimization can be integrating into the TQL interpreter. This illustrates a benefit of having a query language interface between the analysis schemes and the end user.

We assume that the reader is familiar with basic tasking concepts; these concepts are referred to freely in the paper. Our discussion is limited to the tasking concepts from the Ada-83 standard, including the use of entry calls, accepts, rendezvous, and selective-waits. TQL does not provide querying capabilities for delay statements or entry families, nor for any of the newer tasking-related constructs unique to the Ada-95 standard [10] (e.g., asynchronous

transfer of control (ATC), protected objects, and requeue statements). Extending TQL to support Ada-95 would require the definition of some new state expressions, such as states to capture the execution of an abortable part of code in an ATC statement, the execution of an entry in a protected object, and the requeueing of some entry call. Further evaluation of the appropriate Ada-95 constructs is needed to identify how they impact our current interpretation of rendezvous-related states as stated in Section 3.2. Also, on the implementation side, we need to define suitable Petri net models of the relevant Ada-95 constructs and show how the new TQL queries of concern can map to net analysis. As a first step, we have begun the work on defining Petri net models for the previously named Ada-95 constructs. In the remainder of this paper, references to Petri net concepts [11] are kept to a minimum and only become important in Section 4.

The rest of this paper is organized as follows. Section 2 provides a sample of some of TQL's basic capabilities to give the reader an intuitive feel for the language's intended usage. Section 3 provides a more formal description of TQL commands and operators. Section 4 discusses a prototype implementation of TQL, with some experimental results, and Section 5 provides an annotated simple session that illustrates how a hypothetical programmer might use TQL. Finally, Section 6 summarizes the work and suggests directions for future research.

## 2. A Sample of TQL Concepts and Capabilities

TQL is a prototype language, an effort to understand and demonstrate the capabilities needed for analyzing distributed behavior. Consequently, we have favored a syntax for TQL that is logical and functional rather than one that is specifically tuned from a human interface standpoint. Here we introduce some key concepts used by TQL and give some examples to establish an intuitive feel for the type of query language support we desire. Section 3 details the major underlying principles of TQL's syntax and design.

The key concept underlying TQL's design is that of a state of interest. Some typical states of interest are deadlock states; states in which some particular entry call in a task is blocked waiting for an accept; and states in which, say, two particular rendezvous are executing concurrently. States of interest are identified using state expressions. TQL provides a rich syntax for specifying state expressions. Basically, a state expression is evaluated to yield a set of states (the states of interest for that state expression). The "TQL>" symbol is the prompt generated by the TQL interpreter.

This first example shows a check to see if a program can reach a deadlock. The reserved set variable DEAD contains the set of states (identified by state numbers) in which the program deadlocks. So, the simple query

```
TQL> DEAD
```

displays the set of states in the reserved set DEAD. If the set is null, the program is deadlock-free. TQL supports two optional output formats: one to provide the cardinality of the state expression (indicated by a "-C" attribute at the end of a state expression query), and one to provide an interpretation in English of each state that matches the state expression (indicated by a "-I" attribute). This is illustrated by an example in Section 5.1.

TQL evaluates interactive queries involving Ada tasking behavior, which is manifested in rendezvous, call-blocked and accept-blocked states. Two tasks are in rendezvous if the calling task has made an entry call to the accepting task which has accepted the call and the body of the accept is executing. A task is said to be *call-blocked* when the task has made an entry call that has not yet been accepted, and a task is said to be *accept-blocked* when the task is blocked at an accept statement waiting for a corresponding entry call.

The following query identifies states in which task T1 is call-blocked on a call to entry E in task T2. The colon specifies that call-blocked is the type of Ada tasking behavior of interest; the left-hand side of the expression specifies the calling task and the right-hand side specifies the accepting entry point.

```
TQL> T1 : T2.E
```

Using TQL we can identify states in which two (or more) kinds of tasking events are occurring. For example, to identify states in which a rendezvous is in progress between tasks T1 and T2, and task T2 is call-blocked waiting for some other task to accept its entry call, we use the following query (the double colon operator indicates a rendezvous state, the comma denotes the *and* operator, and the \* serves as a task name wild card):

```
TQL> T1 :: T2, T2 : *
```

To identify states in which task T1 is call-blocked on task T2's entry E and in fact T1 is the only task waiting at that entry, we can use the following TQL query. The Q function returns the set of states in which the size of the specified entry queue satisfies some boolean relation; in this example, we are interested in states in which the size of the entry queue for entry E of task T2 is equal to 1.

TQL> T1 : T2.E , Q(T2.E) = 1

The Q function can also be used to find the states in which there are a maximum number of call-blocked tasks in an entry queue. This is done by comparing the queue function's result to a keyword *max*.

We now show some queries that use the TQL feature of a *named wild card*. As a first example, consider a query to display states containing a concurrent rendezvous in which tasks T1 and T2 are both in rendezvous with the same task. A named wild card must be used to identify a wild-carded name for a task that holds across the entire expression. Every instance of a particular named wild card in a state expression must represent the same task. Note in the following query that T1 and T2 are specific known task names (provided by the user when the query is formed), but \*A is a task-name symbol (a named wild card).

TQL> T1 :: \*A , T2 :: \*A

This query says to find those states in which task T1 (as the caller) is in rendezvous with some task, and task T2 (as the caller) is also in rendezvous with that same task.

Named wild cards must also be used to identify states in which a nested rendezvous is occurring. For example, we might want to identify states in which task T1 is in rendezvous with some other task (a wild-carded task \*A) and at the same time that the same task (denoted by \*A) is in rendezvous with yet another task. This can be expressed by a query that uses both a named wild card and the unnamed wild card. For example, to identify states in which a two-task circular deadlock is occurring, we can specify a query in which a wild-carded task \*A is call-blocked on a second wild-carded task \*B and at the same time task \*B is call-blocked on the first task \*A. This defines one way that deadlock can occur in an Ada program. The query is as follows:

TQL> \*A : \*B , \*B : \*A

The need for named wild cards becomes clear when we attempt to formulate a query, using only the unnamed wild card, that specifies states containing at least two rendezvous:

TQL> \* :: \* , \* :: \*

The problem here is that both of the conditions in the query are the same -- the query simply specifies states in which a rendezvous is occurring (between any two tasks) and a rendezvous is occurring (again, between any two

tasks). So this query really identifies states in which at least one rendezvous is occurring. The distinct wild card names A and B in the following query, on the other hand, serve to explicitly specify different calling tasks and thus two distinct rendezvous.

```
TQL> *A :: * , *B :: *
```

It is possible to identify states containing exactly two concurrent rendezvous by adding to the query a condition that specifies that a third distinct rendezvous is not allowed -- this identifies states containing proper two-concurrency (i.e., exactly two concurrent rendezvous). In TQL this is done by prefixing a state query term with the  $\sim$  symbol, which serves as the not operator. So, to identify states containing exact n-concurrency we can use the following query:

```
TQL> *A1 :: * , *A2 :: * , ... , *An :: * , ~ *I :: *
```

### 3. TQL Structure

In Section 2, we used examples to motivate TQL concepts and to introduce representative TQL queries. In this section we turn our attention to providing a formal definition of some key structures of the TQL language.

#### 3.1 State Expressions

The user (programmer) specifies the particular type of concerned activity using a state expression, which TQL evaluates to a set of program states. A state expression consists of set references, set descriptors, and operators which combine them. The operators, in order of precedence, are the unary operator not ( $\sim$ ) and the binary operators and ( $\cdot$ ) and or ( $\cup$ ). As usual, operators of like precedence are evaluated from left to right, while parentheses may be used to force a particular order of evaluation.

A set reference explicitly specifies a set of states. A set reference can be a reserved set variable, a user-defined set variable, or a set constant. TQL supports five reserved set variables: ALL is the set of all program states, INIT is the set containing the initial program state, TERM is the set of all terminal states (states with no successor states), END is the set of terminal states that represent normal program termination (when all tasks reach their end) and DEAD is the set of terminal states that represent program deadlock. A user-defined set variable is created using the

TQL set command, which evaluates a state expression and assigns the resulting set of states to the specified variable (the set command is discussed further in Section 3.4). The name of a user-defined set variable is a string consisting of letters, digits, and the underscore character. A set constant is a set of states of the form { #int, #int, ..., #int }; each state is identified by a unique state number.

A set descriptor specifies an activity of interest. Evaluation of a set descriptor yields the set of all states containing that activity. There are two types of set descriptors, tasking activity descriptors and program activity descriptors. These are discussed in Sections 3.2 and 3.3, respectively.

Where appropriate, a user can specify a wild card (\*) in a set descriptor in place of a task name. A wild card can match any task name. Furthermore, wild cards in set descriptors can be named to specify particular relationships between tasks. A named wild card has the form

\* wcardname

where wcardname is a string of characters specifying the wild card's name. TQL evaluates a state expression to a set of program states  $S$  such that, in each state in  $S$ , task names corresponding to the same wild card name are the same and task names corresponding to different wild card names are different.

### 3.2 Tasking Activity Set Descriptors

Tasking activity [12] occurs whenever one task interacts, or attempts to interact, with another. These interactions ultimately specify a distributed program's behavior and are the object of our study.

An executing task that contains entry calls can be in the following rendezvous-related states:

- Call-blocked. The task has issued an entry call that has not yet been accepted.
- Rendezvous. The task has issued an entry call that has been accepted, and it is waiting for the rendezvous to finish.

An executing task that contains accept statements can be in the following rendezvous-related states:

- Accept-blocked. The task is blocked at an accept statement waiting for an entry call.
- Rendezvous. The task has accepted an entry call and the body of the accept statement is executing.

Further, if the task contains a selective wait statement, it can be in the following state:

- Select-blocked. The task is blocked at a select statement waiting for an entry call for one of its accept alternatives.

Note that a reference to a "blocked" task here refers only to a particular program state and does not necessarily imply that the task is permanently blocked.

TQL queries can be built from tasking activity set descriptors, which specify tasking activities. These descriptors correspond to the discussion of tasking activity above and can be any of the following: an execution descriptor (discussed in Section 3.2.1), which allows the user to determine if a task has finished executing; a rendezvous related descriptor (Section 3.2.2), which allows the user to identify the execution of entry call and accept statements; a select-blocked descriptor, which allows the user to identify the execution of select statements; or a queue descriptor (Section 3.2.3), which allows queries on an entry's queue. These descriptors are discussed in the following subsections.

### 3.2.1 Execution Descriptor

Evaluation of an execution descriptor yields the set of states in which a task has finished executing -- i.e., the task has reached its end. It has the form

taskname

where taskname is the name of the task of interest. If the *not* operator precedes the descriptor, its evaluation yields instead the set of states in which the specified task has not completed its execution.

### 3.2.2 Rendezvous Related Descriptors

All rendezvous related descriptors have the general form

call-specification <op> accept-specification

where call-specification identifies entry calls of interest, accept-specification identifies accept statements of interest, and <op> is the descriptor's operator. There are four rendezvous related operators: 1) the rendezvous operator, 2) the rendezvous event operator (which both use a double-colon operator symbol (::) to indicate that two tasks are committed to rendezvous), 3) the call-blocked operator, and 4) the accept-blocked operator (the call-blocked and accept-blocked operators both use a single-colon operator symbol (:)) to indicate that one task is waiting for rendezvous to occur). Further discussion on the semantics of these operators is given below.

TQL allows the degree of specificity for the source of the entry call (call-specification) to vary in every possible way: the asterisk wild card (\*) specifies an entry call made by any task, a task name (calltask) specifies an

entry call made by one particular task, and a group of line numbers ( $e_1, e_2, \dots, e_n$ ) specify entry calls made from particular lines in a task.

Similarly, TQL allows the degree of specificity for the entry call's acceptor (accept-specification) to vary in every possible way: the asterisk wild card (\*) denotes an accept statement in any task, a task name (accepttask) denotes an accept statement in one particular task, an entry name (accepttask.entry) denotes an accept statement for a certain entry in a particular task, and a group of line numbers ( $a_1, a_2, \dots, a_n$ ) denotes accept statements in particular lines in a task.

Table 1 summarizes the various possible forms of rendezvous related descriptors. Notice that when call-specification is a group of line numbers  $e_1, e_2, \dots, e_n$ , accept-specification does not redundantly specify a task name or entry name because the entry calls in lines  $e_1, e_2, \dots, e_n$  already specify that information. We now discuss each of the four rendezvous related descriptor operators.

Use of the rendezvous operator defines a rendezvous descriptor. Evaluation of a rendezvous descriptor yields a set of states in which one task has made an entry call that another task has accepted, and a rendezvous is in progress. The rendezvous descriptor has the general form

call-specification :: accept-specification

where call-specification and accept-specification together specify entry calls and accept statements of interest. Table 1, using operator ::, summarizes the possible forms of the rendezvous descriptor.

Table 1 Forms of the Rendezvous Related Descriptors

| call-spec <op> accept-spec   | call-specification<br>specifies an entry call                            | accept-specification<br>specifies an accept                              |
|--|--|--|
| * <op> *   | in any task  | in any other task  |
| * <op> accepttask  | in any task  | in task accepttask   |
| * <op> accepttask.entry  | in any task  | for entry entry in task accepttask                                       |
| * <op> (a <sub>1</sub> , a <sub>2</sub> , . . . , a <sub>n</sub> )   | in any task  | in one of lines a <sub>1</sub> , a <sub>2</sub> , . . . , a <sub>n</sub> |
| calltask <op> *  | in task calltask   | in any other task  |
| calltask <op> accepttask   | in task calltask   | in task accepttask   |
| calltask <op> accepttask.entry   | in task calltask   | for entry entry in task accepttask                                       |
| calltask <op> (a <sub>1</sub> , a <sub>2</sub> , . . . , a <sub>n</sub> )  | in task calltask   | in one of lines a <sub>1</sub> , a <sub>2</sub> , . . . , a <sub>n</sub> |
| (e <sub>1</sub> , e <sub>2</sub> , . . . , e <sub>n</sub> ) <op> *   | in one of lines e <sub>1</sub> , e <sub>2</sub> , . . . , e <sub>n</sub> | in any task  |
| (e <sub>1</sub> , e <sub>2</sub> , . . . , e <sub>n</sub> ) <op> (a <sub>1</sub> , a <sub>2</sub> , . . . , a <sub>n</sub> ) | in one of lines e <sub>1</sub> , e <sub>2</sub> , . . . , e <sub>n</sub> | in one of lines a <sub>1</sub> , a <sub>2</sub> , . . . , a <sub>n</sub> |

Use of a rendezvous event operator defines a rendezvous event descriptor. When a rendezvous occurs between a calling task and an accept statement that does not have a body, we consider the rendezvous to be an atomic event that moves the program directly from a state in which the rendezvous is ready to occur to a state in which it has occurred. As a result, there is no program state during which the rendezvous is in progress. Evaluation of a rendezvous event descriptor yields a set of states in which a specified rendezvous is ready to occur; this identifies rendezvous involving accept statements without bodies as well as those with bodies. The rendezvous event descriptor has the general form

call-specification :?: accept-specification

where call-specification and accept-specification together specify entry calls and accept statements of interest. Table 1, using operator :?, summarizes the possible forms of the rendezvous event descriptor.

Use of a call-blocked operator defines a call-blocked descriptor. Evaluation of a call-blocked descriptor yields a set of states in which a task has made an entry call that has not yet been accepted. The call-blocked descriptor has the general form

call-specification : accept-specification

where call-specification and accept-specification together specify entry calls and accept statements of interest. Table 1, using operator `:`, summarizes the possible forms of the call-blocked descriptor, with the following exception: since an entry call in Ada cannot specify which accept statement corresponding to that entry should accept the call, accept-specification cannot be constrained by specifying a group of line numbers  $a_1, a_2, \dots, a_n$ .

Use of an accept-blocked operator defines an accept-blocked descriptor. Evaluation of an accept-blocked descriptor yields a set of states in which a task is blocked at an accept statement waiting for an entry call on that entry. The accept-blocked descriptor has the general form

`: accept-specification`

where accept-specification specifies the accept statements of interest. Note that the accept-blocked descriptor does not specify call-specification. This results from the asymmetry of Ada's rendezvous mechanism--the sender explicitly names the receiver but the receiver cannot specify the sender. In effect, then, call-specification must be the wild card `*`, and referring to Table 1 we see that accept-specification can be `*`, `accepttask`, `accepttask.entry`, or  $(a_1, a_2, \dots, a_n)$ .

### 3.2.3 Queue Descriptor

Evaluation of a queue descriptor yields a set of states in which the queue for a particular entry has a specified size (i.e., contains the specified number of pending entry calls). The queue descriptor has the general form

`Q(taskname.entry) op size`

where `taskname.entry` is the entry queue of interest, `op` is a relational operator (`=`, `<`, `>`, `<=`, or `>=`), and `size` is `max` or an integer. If `size` is `max`, evaluation of the queue descriptor yields the set of states in which the queue for the specified entry has its maximum size with respect to all possible states. (Queue descriptors using the operators `<` and `<=` ignore states in which the specified queue is empty, since the user is probably interested only in states in which there are entry calls in the queue.) For example, the queue descriptor

`Q(MONITOR.START_WRITE) > 10`

specifies states in which the queue for entry `START_WRITE` in task `MONITOR` contains more than ten entry calls -  
- i.e., more than ten writers are waiting to begin writing.

### 3.3 Program Activity Set Descriptors

In addition to defining descriptors that refer to tasking activities, TQL also defines a descriptor that can be used to explicitly identify general (and programmer defined) program activities. For example, consider the implementation of a simple version of the readers/writers problem shown in Figure 2. Task `R1` asks task `MONITOR` for permission to read by initiating a rendezvous with that task (through entry `START_READ`), and later uses another rendezvous to inform the monitor that reading activities have ended (entry `END_READ`). Between these two rendezvous no tasking activity is taking place. Thus, the reading activity is not itself a tasking activity, but rather an activity that is coordinated by tasking activity. A similar observation may be made regarding the writing task `W1`.

This use of tasking activity to coordinate an activity of interest, rather than being itself the activity of interest, motivates an additional level of abstraction in TQL. The programmer may explicitly identify and name various activities in the program using an activity statement in the form of the special Ada comment `--*`. For example, to support analysis using TQL, the body of the loop in task `R1` can be annotated as follows:

```
MONITOR.START_READ;
--* reading
MONITOR.END_READ;
```

The activity name `reading` allows the `TOTAL` toolkit to associate the reading activity by name with specific states of the program's execution.

We refer to a named activity using a program activity set descriptor, which has the general form

```
task-specification ( activity-name )
```

where `task-specification` is the asterisk wild card (`*`) or an explicit task name. Evaluation of such a descriptor identifies states in which activity `activity-name` is taking place in a task specified by `task-specification`. For example, the descriptor

```
R1 (reading)
```

identifies states in which task R1 is engaged in a reading activity (i.e., an activity named "reading"). Further, the state expression

**R1 (reading), W1 (writing)**

finds the set of states in which task R1 is reading and, at the same time, task W1 is writing; one would presumably expect this set to be empty for a correct version of a readers/writers program. Extending Figure 2 to allow multiple readers and writers, the state expression

**\*RDR1 (reading) , \*RDR2 (reading)**

identifies states in which two distinct readers are simultaneously reading, useful for ensuring that the program allows simultaneous reads. Similarly we can identify states (undesirable, of course) in which multiple writers are writing using the expression

**\*WTR1 (writing) , \*WTR2 (writing)**

Finally, we can find states in which any reader is reading and any writer is writing using the query

**\*(reading) , \*(writing)**

### 3.4 Commands

Since evaluating a state expression may take a long time, if a set of states will be used more than once in an analysis session, it is much faster to evaluate the expression once and assign the resulting set to a set variable. The set variable can then be used in subsequent queries. Also, since a state expression may be lengthy, consisting of a number of elements, some expressions may be hard to enter all at one time without making a typographical error. Even if entered correctly, a long state expression may be hard to read, particularly when an analysis session is reviewed later. So, a better way to enter a long state expression is to assign each element in the state expression to a meaningfully-named set variable, then combine the set variables to form the final state expression.

TQL supports such use of set variables through a set command. This command assigns the value of the state expression on the right of the equals sign to the set variable on the left of the equals sign.

The set command has the general form

`var = state-expression`

where `var` is a user-defined set variable (see Section 3.1) and `state-expression` is a state expression. Since evaluation of a state expression always yields a set of states, a set variable always represents a set of states. As discussed in the beginning of Section 3, a set variable is a semantically valid state expression, so it can be used anywhere a state expression is allowed. An example of the use of a set command is given in the sample session of Section 5.

A state expression used by itself corresponds to a state query, or a state command. The state query simply evaluates the state expression and displays the resulting set of states. It has the general form

`state-expression [-C][-I]`

By default, TQL displays the set of states as a set of state numbers. Option C causes TQL to display only the cardinality of the set of states, while option I forces an interpretation of each state and causes TQL to display an English description of each state in the set of states. The description of a state indicates the actions of each task at that program state.

To identify and examine sequences of states that precede, follow, or connect states of interest, requires a path query. Some examples of path queries are given in the sample session of Section 5.

## 4. Automated Analysis in a Prototype Implementation of TQL

### 4.1 Supporting Methods and Logic

In this section, we discuss the integration of a set of Petri net analysis techniques into a prototype interpreter of TQL. It is the TQL interpreter that makes automation of analysis possible.

In our implementation, TQL serves as a user interface language/tool for the TOTAL analysis system [6], a prototype system being developed to evaluate the application of Petri nets for supporting automated tasking analysis. Petri nets, a formal (graph-based) model of concurrency, are well suited for modeling systems characterized by asynchronous concurrency with synchronization and nondeterminism. The state of a Petri net model is defined by certain nodes (called place nodes, or just places) that are distinguished in that state--the places are said to be marked.

Details on Petri net terminology, definitions, and behavior can be found in [11]. Since TQL defines queries over the program's state space in the context of the Ada language, while the underlying analysis techniques are rooted in the domain of Petri nets, an analysis session is carried out in three steps: (1) translation of the source program into a Petri net model; (2) interpretation of the TQL queries into internal queries expressed in the context of the net model; and (3) evaluation of the internal queries and presentation of the query results. Tools for translating the program into an appropriate Petri net model, which we call an Ada net, are discussed elsewhere [5, 6]; here, we focus on interpreting and evaluating TQL queries.

In keeping with our desire to use available Petri net theory and tools, we have built our system around the TOTAL analysis system which embraces a number of analysis techniques based on Petri net theory and methods. Among these techniques are reachability graph analysis, net reduction, and linear algebra techniques. For reachability analysis, an existing tool set called P-NUT [13] is adopted. The two most important P-NUT tools for our use are a reachability graph builder, called RGB, and a reachability graph analyzer, called RGA. We use RGB to build the reachability graph, and RGA to perform the appropriate searches and evaluations of that graph. RGA defines a versatile interactive language capable of performing analysis of Petri net reachability graphs. However, two obstacles exist in the direct use of P-NUT tools. First, because RGA is a Petri net tool, it can be effectively used only by someone intimately familiar with the construction of the Petri net being analyzed. As we have stated, though, the goal of TQL is to allow formulation of analysis queries without requiring knowledge of Petri net concepts on the part of the user. Our implementation of TQL basically consists of translating TQL queries into appropriate RGA queries, and translating RGA responses back into user-consumable query replies. The second obstacle is that the state space explosion problem severely limits the practical capability of reachability analysis. One way that has shown promise of being effective is to reduce the Ada net by net reduction before constructing the reachability graph with RGB. Another way is to avoid enumeration of an entire reachability graph by using linear algebra techniques. We will explain how this works shortly.

We have implemented TQL in C using Unix's lex and yacc tools to recognize the user's TQL queries. As shown in the previous section, one TQL query may correspond to a number of program states. The TQL interpreter enumerates these program states and recognizes the logical relationship (such as *and* or *or*) between these states according to the semantics of the TQL query. The correspondence between program states and Ada net markings is

established by taking advantage of the informative label of each place in an Ada net. When an Ada net is constructed, each place is labeled with a string telling the type and line number of its corresponding statement and the role of the place [5]. When one or more statements explicitly define a program state, it is straightforward to recognize their corresponding places in the Ada net. We call these places *key places*. For a completely specified program state (in which every task's status is specified by the user), a Petri net reachability problem is defined, i.e., to search for a marking in which only a set of key places are marked, corresponding to the specified program state. For a partially specified program state (in which the user specifies only the status of certain tasks, but does not care about other tasks), a Petri net coverability problem is defined, i.e., to search for one of the markings in which at least a set of key places are marked. In either case, the analysis procedures are automated by the TQL interpreter. For some queries, such as deadlock detection (DEAD in TQL), no key place is singled out. The target markings are implicitly specified and TQL invokes some special analysis routines [7, 14].

As we mentioned, reducing the state space is helpful for practical analysis. Basically, net reduction simplifies a Petri net by removing nodes while preserving a set of concerned properties such as liveness or safeness of the net. However, we must avoid over reduction of a net, meaning the removal of useful nodes such as some key places needed for a specific query. Since the TQL interpreter recognizes the key places for each concerned program state, it is always possible to tag each of these key places and then apply net reduction while avoiding removal of any tagged places.

Having reduced the net model, the actual analysis can be conducted. In addition to reachability analysis supported by RGA, linear algebra techniques can also be applied, attempting to avoid explicit enumeration of the net's reachability graph. For example, a reachable marking,  $M$ , can be represented by a set of linear equations (the state equation of a Petri net [11]),

$$A^T x + M_0 = M \quad (1)$$

where  $A$  is the incidence matrix of the Petri net model,  $M_0$  is the initial marking and  $x$  is a firing counting vector variable. (Each element of  $x$ ,  $x_i$ , represents the number of times that transition  $t_i$  fires in moving from the marking  $M_0$  to marking  $M$ .) If there is no solution to Equation (1), then  $M$  is not reachable. Often, solving a linear programming (LP) problem subject to a set of constraints similar to Equation (1) can greatly help in determining

reachability of program states [15]. As a typical example, the following LP problem can be used to check the reachability of a target marking  $M$ :

$$\begin{aligned} \min_x \{ & \sum_{i=1}^m x_i \} \\ \text{subject to } & A^T x + M_0 = M \end{aligned} \quad (2)$$

where  $m$  is the number of transitions in the net. The objective function  $\min_x \{ \sum_{i=1}^m x_i \}$  reflects our desire to find the target program state reachable in fewest steps.

Similarly, a coverable marking,  $M$ , can be represented by using a set of linear inequalities as the constraints of the LP problem,

$$A^T x + M_0 \leq M \quad (3)$$

As a well studied and commonly used mathematical method, many powerful off-the-shelf LP software packages are available. In use of the LP method, however, the difficulty lies in automated formulation of the appropriate LP problems for various Ada tasking related queries [16]. Now, given a formal query language, we can let the TQL interpreter construct the LP problem according to the concerned program state and the semantics of the query.

When invoked, TQL creates two processes. One is to run RGB and RGA, and the other is to run a linear programming (LP) solver. Both processes connect to the TQL interpreter through Unix sockets. Figure 3 illustrates the basic architecture of our implementation. After a program is translated to its Ada net form, a reduced net, called  $RN$ , for deadlock detection is generated using a net reduction tool called NRT [7]. Then the reachability graph of  $RN$  is constructed by RGB. When a query is parsed in TQL, the TQL interpreter will define some target program states and create corresponding queries in the Ada net context. To do this, a set of key places are tagged if any such places are defined for the query at hand. For example, a rendezvous query would define some key places that indicate rendezvous states, whereas a deadlock query would not define any such key places. If every key place is contained in the current reduced net  $RN$ , the TQL interpreter will define a set of targeted markings and create an RGA query and pass the query to the RGA tool. An example of this will be given shortly. The TQL interpreter also formulates an LP problem corresponding to the query and sends this problem to the LP solver. So, the RGA analyzer and the LP solver can work in parallel. As soon as one of these two processes obtains a definitive result, it sends the result back to the TQL interpreter, which will in turn terminate the other process. This returned information can consist of

a set of states, a set of places, or a set of state sequences. This result is either displayed to the user in an appropriate form based on the output option specified in the TQL command, or saved by TQL and used for further processing.

Of course it is possible that some key places for a given query are not actually preserved in the reduced Ada net, RN. In this case, the TQL interpreter will have to create a new reduced net for use in the analysis -- one that keeps every currently tagged place. The remaining jobs for the query processing will be the same as already discussed.

## 4.2 Creating Reachability Queries

We enhanced the original RGA tool to perform socket communication and to support some string pattern matching capabilities. The modified RGA allows us to find states by providing appropriate place label patterns. The labeling scheme for the nodes of the Ada net is defined by our program-to-net translator tool. For example, to find states in which a task has made an entry call to the task Buf and is waiting for an acknowledgment, we can use the place label pattern `ack_entry_Buf_*`, where `*` indicates a match of any characters. Other types of pattern matching are also used for other types of TQL queries.

When TQL first initiates RGA, several RGA functions are created. The `state_search` function has an input parameter called `places_set`, which is a set of place labels. The function returns the set of states named TMP. TMP is obtained by searching in the reachability graph for states in which any marked place is in `places_set`. The actual RGA code for the `state_search` function is given below, where `STATE_SET`, `s`, and `p` are local variables; `S` is RGA's predefined set of all reachable states; and the function `p(s)` returns 1 if the place labeled `p` is marked in the state `s`. Details of RGA are given in [13].

```
state_search(places_set)[STATE_SET,s,p] ::=
  STATE_SET := {};
  forall s in S [forall p in places_set
    [if p(s)=1 then
      STATE_SET := union(STATE_SET,{s })
    fi;
    true]];
  STATE_SET
```

To process the query "TQL> \*::\*", which identifies states in which any two tasks are in rendezvous, TQL constructs an RGA query to find all the places in the Ada net that can be marked when a rendezvous occurs. These

places are labeled `entry_ex_i_j`, where  $i$  is the line number of the accept statement and  $j$  is the line number of the entry call statement. TQL thus derives the place label pattern `entry_ex_*`, and forms the following RGA query that will generate a set of place labels matching the pattern ( $P$  is RGA's predefined set of all place labels for the net being processed):

```
entry_exs := { entry_ex_* in P };
```

Next, TQL uses the `state_search` function discussed above to form an RGA query that will return the states of interest:

```
TMP_0 := state_search(entry_exs);
```

After TQL passes the above RGA queries to RGA through a Unix socket, TQL waits for RGA to return the results through the socket. In this case, RGA will send `TMP_0`, a set of state numbers, back to TQL.

If the *I* output format option is also specified in the TQL query, TQL passes a second set of RGA queries to RGA using RGA's `showstate` function. For each state passed in the query, the `showstate` function returns to TQL the labels of all places that are marked in that state. These place labels are used to create output that gives an interpretation of what is happening in each state. For example, assume that a TQL query identifies a state of interest that corresponds to a net state (i.e., a reachable marking in the Petri net) in which the following three Petri net place nodes are marked with a token: `entry__Operator_19`, `accept_29`, and `select_40`. In the case of a query that calls for an interpretation of this state, each place label is used to create a statement about a particular task. For example, the label `entry_Operator_19` indicates a place node that corresponds to line number 19 of the Ada source program. By consulting a "line number range" table, TQL can identify the name of the task. For the sake of illustration, assume that the task name is `Customer`. So, the output statement generated as an interpretation of this place label is "Task Customer at line 19 is ready to make an entry call to Operator.Prepay." TQL knows that the entry being called at line 19 is named `Prepay` by a simple lookup in another table called an "entry call table." The output statements generated as an interpretation of the other two place labels are "Task Pump is accept-blocked at accept Activate in line 29" and "Task Operator at line 40 is select-blocked at a select for entries: Prepay, Charge." As a group, the three

output statements form a TQL interpretation of the state of interest. The reader can see this same example used in the context of an actual program in the sample session given in Section 5.

In Section 3.3 we discussed the concept of a program activity descriptor. Implementation of the program activity descriptor requires associating a named activity with a place in the net. The most straightforward way to do this, of course, is to add a properly labeled place to the net for each named activity. However, each additional place increases the size of the net's state space. We have also studied the possibility of associating the activity with an existing place [17]. This approach can complicate proper net optimization, but this issue is outside the scope of our discussion.

Because of TQL's design, it is possible for TQL to be implemented in a distributed environment with TQL doing the query translation on one hardware platform and RGA performing the reachability graph analysis on a second (and probably more powerful) hardware platform.

TQL completely shields the underlying model, the Petri net, from the user. However, TQL also supports a number of "debugging" modes that can be interactively turned on and off by the user--these debugging modes allow the user to see the underlying RGA queries that are constructed by the TQL interpreter. This information is only useful to those familiar with RGA, since it is analogous to the assembly-language code generated by a compiler. TQL also allows users to input RGA queries directly (by prefixing the query with the '!' symbol) -- the query is passed directly to RGA and the RGA results are displayed. This is analogous to allowing the incorporation of assembly-language code into a high-level language program. Some examples of these features are shown in the next section.

## 5. A Simple Analysis Session and Experiments

### 5.1 A Simple Analysis Session

This section provides a transcript of a simple program analysis session using our TQL prototype. We have added some comments, but the rest of the information is reproduced exactly from the TQL session. User input is shown in bold face.

The example we use is a simple version of the popular gas station program [18] for which TOTAL has generated a full state space. Note that the program contains a number of the constructs which interest us--tasks with

entry calls and accepts, accepts with and without bodies, accepts with entry calls in the body, selective-waits with accept alternatives that have entry calls in the body, and activities coordinated by rendezvous. Figure 4 contains the gas station program, with line numbers added for reference.

First we can check for the any deadlock states and request for the display of an English interpretation of what is occurring in each such deadlocked state.

```
TQL> DEAD -I
In State #14
Task Operator at line 46 is call-blocked on entry Customer.Change
Task Customer at line 21 is in rendezvous with Pump.Finish at line 31
Task Pump at line 32 is in rendezvous with Operator.Charge at line 45
```

From this query result we can observe that there is only one deadlock state. In this state the Operator is trying to call the Customer when the Customer is in rendezvous with the Pump. Meanwhile, the Pump (as the caller) is in rendezvous with the Operator. This is a classic case of circular-waiting deadlock. Since we did not expect this to occur, we next check to see if the tasks can successfully make an iteration or if the discovered deadlock must occur during the progress of any iteration.

In the following, the first path query checks for state sequence paths of length one starting from the initial state (denoted by the keyword *INIT*). The second path query checks for state sequence paths that go from a particular source state (state #1 in this example) to a particular destination state (the initial program state in this example).

```
TQL> path +1 INIT
{<<#0,#1>>}

TQL> path {#1} INIT
{}
```

Since this query returns the null set, indicating the program never returns to an initial state, we know that the tasks in the gas station simulation program can never complete a full iteration.

Next, we verify key behavioral properties of the Pump. First we find out if it's possible for the Customer to be pumping even though the Pump hasn't started yet.

```
TQL> Customer(pumping) , ~ Pump(started) -C
0
```

As expected, no program states contain this behavior. Taking the matter one step further, we note that if the Customer is pumping, task Pump should be accept-blocked at entry Finish. We verify that this property is always true by checking that the inverse never occurs:

```
TQL> Customer(pumping) , ~ : Pump.Finish -C
0
```

The following example illustrates the use of a user-defined set variable. Suppose we want to display the number of states in which the Operator task is select-blocked, i.e., blocked at a select statement. Furthermore, we want to save those states in a set variable for later use. We use TQL's select-blocked descriptor (mentioned in Section 3.2), defined by the "\$" operator. The matching states are saved as the user-defined set variable Op\_SB (meaning Operator is Select Blocked).

```
TQL> Op_SB = $ Operator -C
10
```

Now we display interpretations of the states that match the following description: the Operator is select-blocked, the Pump is not accept-blocked, and the state is not the initial state.

```
TQL> Op_SB, ~ : Pump, ~INIT -I
In State #11
Task Customer at line 21 is in rendezvous with Pump.Finish at line 31
Task Pump at line 32 is ready to make an entry call to Operator.Charge
Task Operator at line 40 is select-blocked at a select for entries: Prepay, Charge
In State #12
Task Customer at line 21 is in rendezvous with Pump.Finish at line 31
Task Pump at line 32 is call-blocked on entry Operator.Charge
Task Operator at line 40 is select-blocked at a select for entries: Prepay, Charge
```

Note that the two states identified by this query are only different in that in one state (#11) the Pump is ready to make an entry call, but has not yet made the entry call, and in the other state (#12) the Pump has made the entry call and is call-blocked. Since a task's ability to make an entry call is not in any way dependent on the actions of other tasks, it is probably not necessary to generate states like state #11. State #11 shows up in this example since we did not apply net reduction (as mentioned in Section 4.1) to the Ada net model of this program.

How many states are there in which a rendezvous is ready to occur (i.e., a rendezvous event) between any task as the caller and the Customer as an acceptor? The answer will be zero because the program deadlocks before the Customer can make an accept on the Change entry:

```
TQL> * :?: Customer -C
0
```

We now turn on debug mode to force TQL to display the RGA query that it constructs for a TQL query that finds states in which any task is in rendezvous with the Operator. As discussed in Section 4, TQL forms an RGA query that matches a place-label pattern. In the information following the query we can see TQL attempt to match place labels that are prefixed by the keyword `entry_ex_`, followed by a line number in the range 38 to 50 (the range of the accepting task, which in this case is the Operator task), followed by any line number (since the TQL query specifies that any task can make the entry call). Other details, such as the range delimiters '@' and '#' and the other debug data, are not important to this discussion.

```
TQL> debug on

TQL DEBUG> * :: Operator
Sent: entry_exs := {entry_ex_@38..50#_* in P };
Sent: TMP_0 := state_search(entry_exs);
Sent with rcv: TMP_0
{#2..#4, #13, #14}

TQL DEBUG> debug off

TQL> quit
Exiting TQL and closing rga connection
```

## 5.2 Experiments

We have illustrated the features of our prototype work on TQL in the previous section. In this section, we report some experiments to give an indication of TQL's performance on some queries. Our purpose is to show the effectiveness of combining net reduction and the LP-based analysis technique with the TQL interpreter. The experiments consisted of automated generation of the Ada net, followed by query-directed net reduction and LP problem generation and solution. When the LP solver reports no solution, a conclusive answer has been provided. When a solution is obtained, the feasibility of the firing sequence corresponding to this solution must be confirmed

by constructing a (small) portion of the reduced Ada net's reachability graph (see [11] for details). In every case of the experiments reported below, the time for feasibility checking was negligible. All experiments were performed on a dual processor Sun Sparc 20/612 workstation with 64 Mb of memory.

The reader/writer program in Fig. 2 was used in Section 3 to demonstrate some of TQL's functionality. We indicated some mutual exclusion between readers and writers can be checked by the following query:

R1(reading), W1(writing)

Here, we consider some expanded versions of the basic program in Fig. 2: 5-reader-1-writer, 10-reader-1-writer, and 10-reader-5-writer programs. As desired, TQL returned a negative answer to the above query for all versions. The time to obtain the answer was less than 2 seconds in all cases. Table 2 provides the basic data for the experiments.

---

Table 2 Mutual Exclusion Checking in Readers/Writers Program

---

| (Readers, Writers) | # of Tasks | Response Time (Sec) |
|--------------------|------------|---------------------|
| (5, 1)             | 7          | 0.88                |
| (10,1)             | 12         | 1.50                |
| (10, 5)            | 16         | 1.61                |

The dining philosophers problem has long been used as a benchmark for concurrent programming and concurrency analysis. Almost all published experimental works dealing with this problem consider versions with less than 200 philosopher tasks. We used the TQL concepts to analyze relatively large versions of the dining philosophers program for some problems such as potential parallelism among events. For example, it is well known that no pair of adjacent philosophers should be able to eat at the same time. But is it possible that two such philosophers can put down forks simultaneously? In our coding of the problem for  $n$  philosophers, a philosopher  $i$  iterates over the following actions:

```

Pick_up fork (i+ 1, mod n)    -- the right fork
Pick_up fork i                -- the left fork
Eat
Put_down fork (i+ 1, mod n)  -- the right fork
Put_down fork i              -- the left fork

```

As an example of the above query, we can check to see if philosopher #200's left fork can be put down simultaneously with philosopher #201's left fork, by the following TQL query. The query uses the rendezvous event operator (:?:) defined in Section 3.2.2.

```

Philo_200 :?: Fork_200.Down, Philo_201 :?: Fork_201.Down

```

TQL gave a positive answer for programs consisting of 300, 400, and 500 philosopher tasks. As can be seen in Table 3, the time to obtain the answer was less than 5 minutes in all cases.

---

Table 3 Parallelism Check in Dining Philosophers Program

---

| # of Philosophers | # of Tasks | Response Time (sec) |
|-------------------|------------|---------------------|
| 300               | 600        | 59                  |
| 400               | 800        | 135                 |
| 500               | 1,000      | 246                 |

Our experiments indicate that the integration of various analysis methods into a TQL-type interpreter can lead to more useful fully automated analysis procedures based on formal methods.

## 6. Conclusion

Since it is neither possible nor even desirable to predict the specific program behaviors that an analyst will be interested in, it seems useful to instead provide an analysis toolkit that supports an interactive query language. We have defined and illustrated a tasking query language called TQL that supports the analysis of concurrent programs. TQL not only serves as a query user interface that allows expression of general analysis queries, it also integrates

various analysis techniques into query processing. TQL does so in a way that is independent of the particular formalism supporting the analysis approach, which is in our case the Petri net model.

While we have provided in TQL a complete set of queries based on program state attributes, we expect that this set will not be sufficient for certain analysis situations, such as those requiring knowledge of a state's context. A classical example of this is starvation analysis [19]. Also, as we gain confidence in TQL's effectiveness, we will work to optimize some of the key state-search and set combination algorithms, and seek to take advantage of additional analysis capabilities, such as those based on temporal logic. Finally, as was mentioned in Section 1, we need to consider the impact of Ada-95 tasking constructs on our foundation of analysis queries.

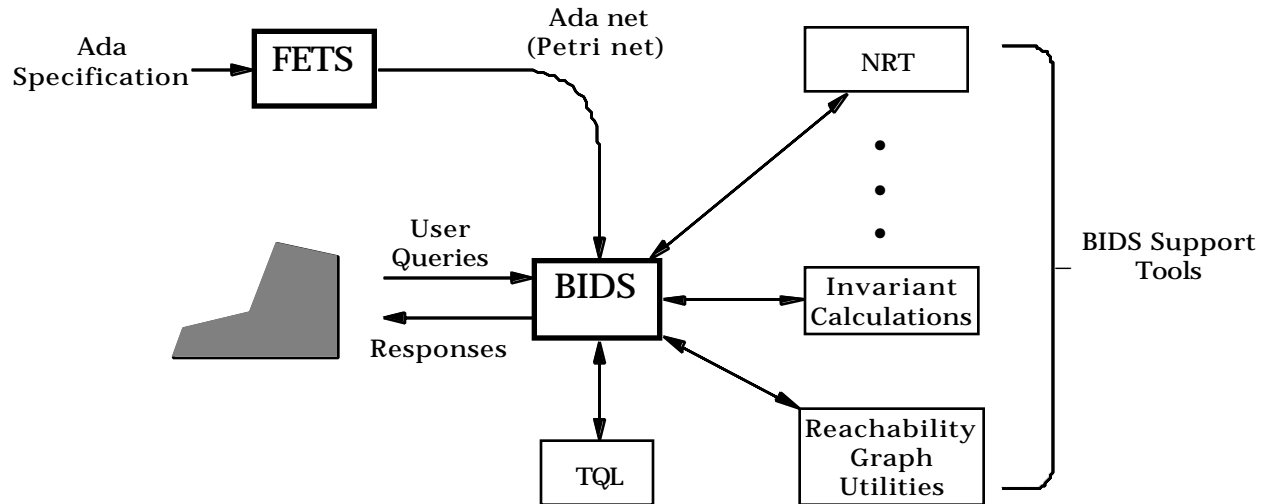
## Acknowledgment

The authors wish to acknowledge S. Upp for implementing most of the TQL query parsing and RGA query generation and S. Henrie for implementing the concept of activity set queries. We also thank Yewton Wang for carrying out the experiments reported.

## References

- [1] R. N. Taylor, "A General-purpose Algorithm For Analyzing Concurrent Programs," *Communications of the ACM*, May 1983, pp. 362-376.
- [2] M. Young, R. Taylor, D. Levine, K. Nies, and D. Brodbeck, "A Concurrency Analysis Tools Suite for Ada Programs: Rationale, Design, and Preliminary Experience," *ACM Trans. on Software Engineering Methodology*, Vol. 4, No. 1, Jan. 1995, pp. 65-106.
- [3] J. Wileden and G. Avrunin, "Toward Automating Analysis Support For Developers of Distributed Software," *Proc. of 8th Int. Conf. on Distributed Computing Systems*, June 1988, pp. 350-357.
- [4] T. Murata, B. Shenker and S.M. Shatz, "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Trans. on Software Engineering*, March 1989, pp. 314-326.
- [5] S. M. Shatz and W. K. Cheng, "A Petri Net Framework For Automated Static Analysis of Ada Tasking Behavior," *Journal of Systems and Software*, Dec. 1988, pp. 343-359.
- [6] S. M. Shatz, K. Mai, C. Black, and S. Tu, "Design and Implementation of a Petri Net Based Toolkit for Ada Tasking Analysis," *IEEE Transactions on Parallel and Distributed Systems*, October 1990, pp. 424-441.
- [7] S. M. Shatz, S. Tu, T. Murata, and S. Duri, "An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis," Accepted to *IEEE Transactions on Parallel and Distributed Systems*, 1996.

- [8] D. Helmbold and D. Luckham, "TSL: Task Sequencing Language," *Proceedings of the Ada International Conference*, Paris, France, 1985, pp. 255-274.
- [9] D. Rosenblum, "An Overview of TSL, a Language for Specifying and Debugging Current Programs," *IEEE Software*, May 1991, pp. 52-61.
- [10] A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge-University Press, 1996.
- [11] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, April 1989, pp. 541-580.
- [12] A. Burns, A. M. Lister, and A. J. Wellings, *A Review of Ada Tasking*, Berlin: Springer-Verlag, 1987.
- [13] E. T. Morgan and R. Razouk, "Interactive State-Space Analysis of Concurrent Systems," *IEEE Trans. on Software Engineering*, Oct. 1987, pp. 1080-1091.
- [14] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, "Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada," *ACM Transactions on Software Engineering Methodology*, Vol. 3, No. 4, Oct. 1994, pp. 340-380.
- [15] G. Avrunin, U. Buy, J. Corbett, L. Dillon, and J. Wileden, "Automated Analysis of Concurrent Systems with the Constrained Expressions Toolkit," *IEEE Transactions on Software Engineering*, Vol. 17, No. 11, Nov. 1991, pp. 1204-1222.
- [16] S. Tu, Y. Wang, and M. E. Mathews, "Query-Driven Petri Net Reduction for Analysis in Ada Tasking," *Proceedings of the 1995 IEEE Phoenix Conference on Computers and Communications*, Phoenix, AZ, March 1995, pp. 334-340.
- [17] S. Henrie, "Adding Task Activity Queries to the TQL System," Master of Science Project, University of Illinois at Chicago, Department of EECS, 1991.
- [18] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, March 1985, pp. 47-57.
- [19] G.M. Karam and R.J.A. Buhr, "Starvation and Critical Race Analyzers for Ada," *IEEE Trans. on Software Engineering*, August 1990, pp. 829-843.



FETS - Front End Translator Subsystem

BIDS - Back-End Information Display Subsystem

Fig. 1 The TOTAL System Architecture

---

Figure 2  
Implementation of the Readers/Writers Problem

---

```
task body R1 is
begin
loop
    MONITOR.START_READ;
    -- reading activity
    MONITOR.END_READ;
end loop;
end R1;

task body W1 is
begin
loop
    MONITOR.START_WRITE;
    -- writing activity
    MONITOR.END_WRITE;
end loop;
end W1;

task body MONITOR is
    READER_PRESENT : BOOLEAN := FALSE;
    WRITER_PRESENT : BOOLEAN := FALSE;
begin
loop
select
    when not WRITER_PRESENT =>
    accept START_READ do
        READER_PRESENT := TRUE;
    end START_READ;
    or
    accept END_READ do
        READER_PRESENT := FALSE;
    end END_READ;
    or
    when not READER_PRESENT =>
    accept START_WRITE do
        WRITER_PRESENT := TRUE;
    end START_WRITE;
    or
    accept END_WRITE do
        WRITER_PRESENT := FALSE;
    end END_WRITE;
end select;
end loop;
end MONITOR;
```

---

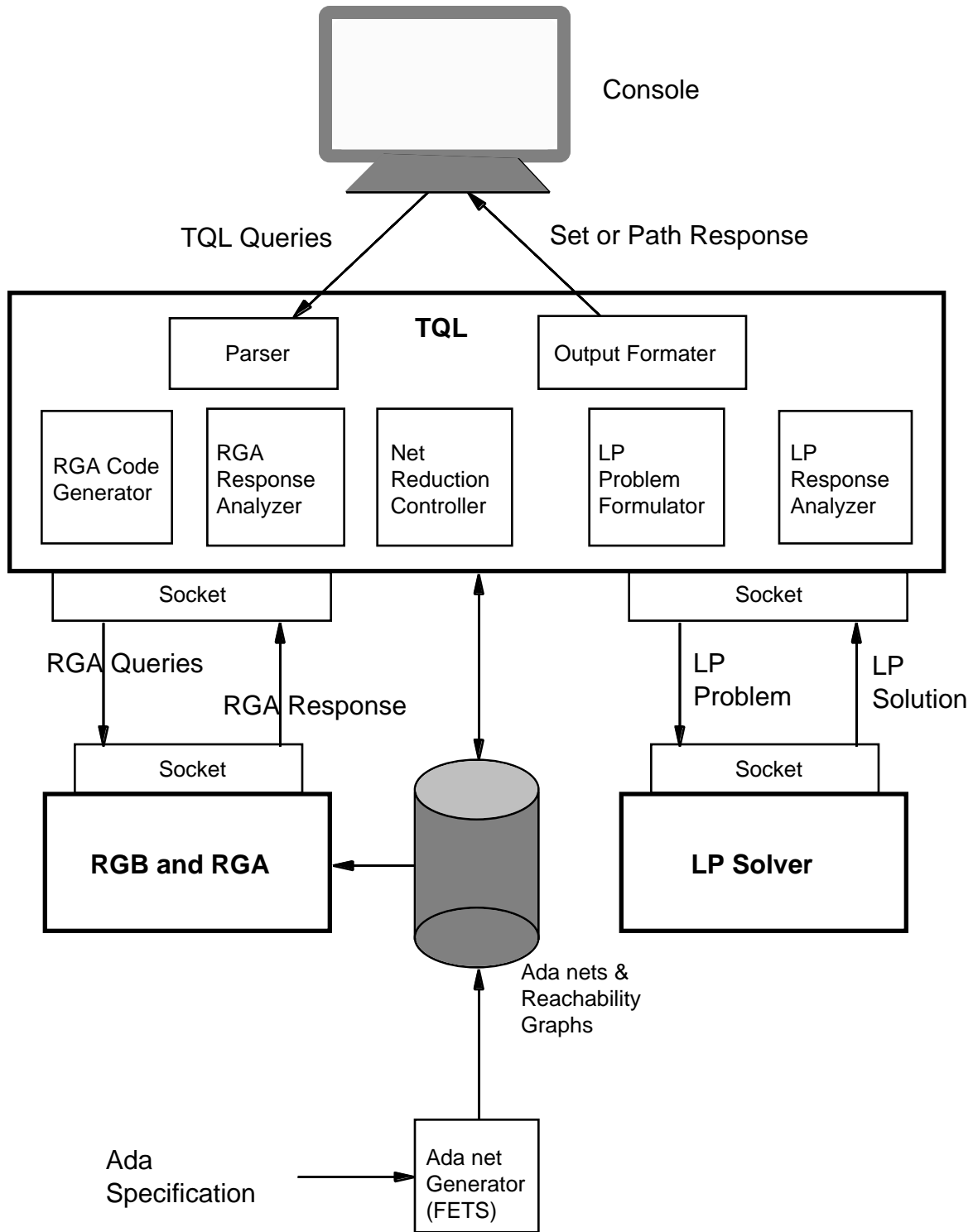


Fig. 3 TQL's Implementation Architecture

Figure 4. Gas Station Program

---

```
1 task Customer is
2   entry Change;
3 end Customer;
4
5 task Pump is
6   entry Activate;
7   entry Start;
8   entry Finish;
9 end Pump;
10
11 task Operator is
12   entry Prepay;
13   entry Charge;
14 end Operator;
15
16 task body Customer is
17 begin
18 loop
19   Operator.Prepay;
20   Pump.Start;
20   --* pumping
21   Pump.Finish;
22   accept Change;
23 end loop;
24 end Customer;
25
26 task body Pump is
27 begin
28 loop
29   accept Activate;
30   accept Start;
30   --* started
31   accept Finish do
32     Operator.Charge;
33   end Finish;
34 end loop;
35 end Pump;
36
37 task body Operator is
38 begin
39 loop
40   select
41     accept Prepay do
42       Pump.Activate;
43     end Prepay;
44   or
45     accept Charge do
46       Customer.Change;
47     end Charge;
48   end select;
49 end loop;
50 end Operator;
```

---