CONCURRENT RUNTIME MONITORING OF FORMALLY SPECIFIED PROGRAMS

Manas Mandal and Sriram Sankar

Technical Report No. CSL-TR-90-425

April 1990

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

Concurrent Runtime Monitoring of Formally Specified Programs

Manas Mandal* and **Sriram Sankar**

Program Analysis and Verification Group
Computer Systems Laboratory
Stanford University
Stanford, California 94305-4055

Computer Systems Laboratory Technical Report CSL-TR-90-425

Abstract

This paper describes an application of formal specifications after an executable program has been constructed. We describe how high level specifications can be utilized to monitor critical aspects of the behavior of a program ***continuously*** while it is executing. This methodology provides a capability to distribute the monitoring of specifications on multi-processor hardware platforms to meet practical time constraints.

Typically, runtime checking of formal specifications involves a significant time penalty which makes it impractical during normal production operation of a program. In previous research, runtime checking has been applied during testing and debugging of software, but not on a permanent basis.

Crucial to our current methodology is the use of ***multi-processor*** machines — hence runtime monitoring can be performed ***concurrently*** on different processors. We describe techniques for distributing checks onto different processors. To control the degree of concurrency, we introduce ***checkpoints***- a point in the program beyond which execution cannot proceed until the specified checks have been completed. Error reporting and recovery in a multi-processor environment is complicated and there are various techniques of handling this. We describe a few of these techniques in this paper.

An implementation of this methodology for the Anna specification language for Ada programs is described. Results of experiments conducted on this implementation using a 12 processor Sequent Symmetry demonstrate that permanent concurrent monitoring of programs based on formal specifications is indeed feasible.

Keywords-Ada, ***Anna, concurrent monitoring, consistency checking, debugging, instrumentation, multi-processors, program specification, program testing, self-checking programs.***

***from Department of Computer and Information Science, The Ohio State University, Columbus, Ohio -** 43210.

Computer Systems Laboratory
Stanford University
Copyright © 1990

1 Introduction

A substantial body of current formal methods research is aimed at utilizing formal specification languages in the software development process. Top-down methodologies such as VDM seek to utilize specifications based on mathematical formalisms (e.g., the Larch and Z specification languages) to develop correct programs (see [2,3,6,14]). Typically, standard theories of specification concepts are developed prior to an application, and then high-level specifications using the concepts of those theories are gradually transformed into executable code in some programming language. Other formal methods seek to prove mathematically that a high-level specification is consistent with a program.

These methodologies may be divided rather loosely into *soft* and ***hard*** formal methods. The main difference between the two is that soft methods do not require a rigorous proof of correctness of each step while hard methods do. The difference between these two philosophies has arisen probably because the hard methods were seen as not achieving short term applications on a wide scale.

This paper describes an application of a soft formal method. We describe how a high level specification can be utilized to monitor critical aspects of the behavior of a program ***continuously*** while it is executing. The executable program (the ***underlying program)*** is constructed using the formal specification prior to the application of our methodology.

Typically, runtime monitoring of formal specifications involves a significant time penalty which makes it impractical during normal production operation of a program. In previous research, runtime monitoring has been applied during testing and debugging of software, but not on a permanent basis [8,11,13]. In this paper, we describe the use of multi-processor machines to perform runtime monitoring concurrently with the execution of the underlying program.

Concurrent runtime monitoring is aimed particularly at dealing with security and safety-critical problems that may occur in systems even when they are produced by hard formal methods. Incorrect behavior in systems produced by formal methods can result from failures in underlying hardware or errors in compilers, operating systems, or other software upon which the target system depends [12].

This use of formal specifications complements both categories of formal methods mentioned above that apply earlier in the development process when a program is being constructed. Also, specifications that have already been constructed and analyzed by other formal methods earlier in the lifecycle can be utilized here.

1.1 Background

Instrumenting programs for the purpose of consistency checking has been around since the mid-70's [15,16]. Since then, many systems have been built where assertions have been compiled into runtime checking code. The recent work on the ***Anna Consistency Checking System*** [11] involves the transformation of various specification language constructs written in Anna [9] into checking code, which is then embedded into the underlying Ada [1] program. This checking code takes the form of ***checking functions*** which are called from the underlying program from locations where the specifications can potentially be violated.

A methodology whereby the checking functions are replaced by Ada tasks is discussed in [10]. These ***checking tasks*** accept program data through ***entry** calls* and perform consistency checking concurrently. A disadvantage of this approach is that the underlying program may run in a potentially unsafe state while the checking tasks are still performing earlier checks. However, this may not pose a problem in many situations. For example,

- It may be acceptable to allow a database application to continue running even after the database gets corrupted until some critical operations are performed by the application.
- Consistency checking of an abstract data type implementation may take place concurrently with the execution of the underlying program until the underlying program makes another call to an operation of this abstract data type.
- In more simplistic programs, the final program output may be the only important activity of the program. In such cases, consistency checking can take place concurrently until the program is ready to produce its output.

In this paper, we describe an implementation of concurrent runtime monitoring of Anna programs. The methodologies described here are a refinement and extension of the methodologies described in [10]. This implementation is bootstrapped on top of the sequential monitoring system [11]. We avoid going into details of the implementation that are not very interesting in the top-level view of the system. The concurrent monitoring system currently assumes that the underlying program is sequential. If the underlying program has multiple threads of control, an extra level of complexity is added. This is a subject for future research.

1.2 Possible Applications

Below is a list of possible applications of the technology for permanently monitoring a program against its formal specifications. These are typically large applications involving complex specifications.

Security of databases: Operations may sometimes want to permit non-secure output based upon secure data, for example, producing an unclassified report which averages some top secret data. Also, a database may want to check dynamically that certain properties of data are preserved. Such details can be specified as constraints which are then checked automatically at runtime. Denning [4] describes an intrusion-detection expert system capable of detecting break-ins, penetrations and other forms of computer abuse. This system is based on monitoring the system's audit records for abnormal patterns. These patterns can be specified as formal constraints and our technology can then be used to automatically generate this expert system. An overview of data security is presented in [5], and clearly indicates the usefulness of the technology.

Consistency of databases: Specifications can be used to describe what it means for a database to contain consistent information. For example, one may specify that the relation f `ather_of` is anti-symmetric. These specifications can be checked at runtime to ensure that the database does not become inconsistent. An example of such a database is

CYC [7]. CYC is a very large evolving knowledge-base of commonsense facts about the world whose consistency aspects are specified formally and checked at runtime.

Program maintenance: Permanent self-checking is also useful in the maintenance of large programs. It is quite common to make inconsistent modifications to a large program. Such mistakes can be detected automatically by a self-checking program. A subtle error may be detected only after many runs of the program on some very specific kinds of inputs.

1.3 Organization of the Paper

The organization of this paper is as follows: Section 2 gives an overview of Anna[1]. Section 3 gives an overview of the relevant aspects of the sequential checking methodology. Section 4 introduces checking tasks and describes how they are used. This section describes the core of the concurrent checking methodology. Section 5 goes into the details of checkpointing and the algorithms used to implement it. Section 6 discusses error reporting and recovery in general. Section 7 describes the experiments conducted on the concurrent checking system and also outlines other experiments that will be performed in the future. Section 8 concludes the paper. Appendix A shows a sample Anna program before and after transformation for concurrent checking.

2 An Overview of Anna

Anna (ANNotated Ada) is a language extension of Ada [1] to include facilities for formally specifying the intended behavior of Ada programs. Anna was designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. In this section we give a brief overview of Anna with special emphasis on subtype annotations. A complete definition of Anna is given in [9].

Anna is based on first-order logic and its syntax is a straightforward extension of the Ada syntax. Anna constructs appear as ***formal comments*** within the Ada source text (within the Ada comment framework). Anna defines two kinds of formal comments, which are introduced by special comment indicators in order to distinguish them from informal comments. These formal comments are ***virtual Ada text***, each line of which begins with the indicator `--:` , and ***annotations,*** each line of which begins with the indicator `--|`.

2.1 Virtual Ada Text

Virtual Ada text is Ada text appearing as formal comments, but otherwise obeying all of the Ada language rules. Virtual text may refer to actual text, but is not allowed to affect the computation of the actual program. Actual text cannot refer to virtual text. The purpose of virtual Ada text is to define *concepts*[2] used in annotations. Often the formal specifications of

---

[1] It is assumed that the reader has a working knowledge of Ada.

[2] Functions used in annotations are called concepts.

a program will refer to concepts that are not explicitly implemented as part of the program. These concepts can be defined as virtual Ada text declarations. Virtual Ada text may also be used to ***compute*** values that are not computed by the actual program, but that are useful in specifying the behavior of the program.

2.2 Annotations

Annotations are constraints on the underlying Ada program. They are comprised of expressions that are boolean-valued. The location of an annotation in the Ada program together with its syntactic structure indicates the kind of constraints that the annotation imposes on the underlying program. Anna provides different kinds of annotations, each associated with a particular Ada construct. Some examples of annotations are subtype annotations, object annotations, statement annotations, subprogram annotations, exception propagation annotations and axiomatic annotations. The Anna expressions extend (i.e., are a superset of) the expressions in Ada.

The Anna language and the nature of the sequential monitoring system is such that concurrent checking needs to be explicitly implemented only for the following Anna constructs: subtype annotations, object annotations and axiomatic annotations. All other constructs are either transformed to one of the above constructs by the sequential monitoring system or else they are not amenable to concurrent checking. For reasons of brevity, this paper concentrates on subtype annotations only. The ideas described in this paper can easily be extended to other kinds of annotation constructs and other specification and programming languages. Subtype annotations are discussed in a little more detail below.

2.2.1 Subtype Annotations

A subtype annotation is a constraint on an Ada type. The constraint applies throughout the scope of the type definition. Subtype annotations are located immediately after the definition of the type they constrain, and are bound to the type definition by the keyword where.

Example of a subtype annotation:

```
type EVEN is new INTEGER;
--| <<EVEN-CONSTRAINT>>
--| where X:EVEN =>
--|     x mod 2 = o;
```

The above subtype annotation constrains all values of the type **EVEN** to be even numbers. The name of the annotation is **EVEN-CONSTRAINT.** This name can be used to refer to the annotation later. This constraint also applies to all derived types and subtypes of **EVEN.**

2.3 Anna Pragmas for Concurrent Checking

Many new Anna language-specific pragmas have been defined for the purpose of controlling the degree of concurrency, checkpointing and error reporting. These pragmas must appear within

virtual text and are described in the sections that follow.

3 Overview of the Sequential Transformation Methodology

Subtype annotations are transformed into ***checking functions.*** These functions take a parameter of the type in question and check this parameter for consistency with the subtype annotation. Calls to the checking functions are inserted at places where inconsistencies with respect to the annotation can arise. Examples of such places are assignment statements, procedure call statements and type conversions.

An example of such a transformation is shown in Figure 1. Here, a subtype annotation that constrains a type to have only even values is transformed to a checking function. Also, an assignment statement in the example is transformed to one that includes a call to the checking function. The subtype annotation and the new code generated as a result of its transformation are enclosed in boxes to indicate their equivalence. The naming conventions used in all examples in this paper are for ease of readability. A more thorough naming convention is used in the actual system.

BEFORE:

```
declare
    type EVEN is new INTEGER;
    --| <<EVEN_CONSTRAINT>>
    --| where X:EVEN =>
    --|       X mod 2 = 0;
    E:EVEN;
begin
    E := exp;
end;
```

AFTER:

```
declare
    type EVEN is new INTEGER;
    E:EVEN;
    function CHECK_EVEN(X : EVEN)
        return EVEN is
    begin
        if not (X mod 2 = 0) then
            report_error;
        end if;
        return X;
    end CHECK-EVEN;
begin
    E := CHECK_EVEN( exp );
end;
```

Figure 1: Example of a Checking Function

If a new type or subtype is defined based on the type EVEN, then the constraint on EVEN also applies to the new type or subtype. For example, the subtype POS_EVEN in the following example is constrained by both its own subtype annotation as well as the annotation on type EVEN.

```
subtype POSEVEN is EVEN;
--| <<POS_EVEN_CONSTRAINT>>
--I where X:POS_EVEN =>
--|      X > 0;
```

In such situations, the checking function of the newly defined type or subtype makes a call to the checking function of the type or subtype based on which it was defined. The checking function for **POS-EVEN** is shown below:

```
function CHECK_POS_EVEN(X:POS_EVEN) return POS-EVEN is
begin
   if not (X > 0) then
      report-error;
   end if;
   return CHECK-EVEN(X);
end CHECK-POS-EVEN;
```

4 Checking Tasks

Checking tasks are Ada tasks that perform consistency checks concurrently with the execution of the underlying program. The motivation behind generating checking tasks instead of checking functions is to improve the performance of the self-checking program in a multi-processor system. Checking tasks accept check requests from the underlying program through entry calls. The parameters of these entry calls are the same as the parameters of the checking functions in the sequential case.

The checking task generated for the type **EVEN** is shown below:

```
task CHECK-TASK-EVEN is
   entry ENQUEUE(E:EVEN);
end CHECK-TASK-EVEN;

task body CHECK-TASK-EVEN is
   X:EVEN;
begin
   loop
      accept ENQUEUE(E:EVEN) do
         X := E;
      end ENQUEUE;
      if not(X mod 2 = 0) then
         report-error;
      end if;
   end loop;
end CHECK-TASK-EVEN;
```

The functionality of ***report-error*** is described in the subsequent sections on checkpointing, and error reporting and recovery.

The above checking task for type **EVEN** is generated instead of the checking function shown in Figure 1. However, the checking function is not completely erased, rather the check within the checking function is replaced by the entry call to the checking task. The modified checking functions for the types **EVEN** and **POS-EVEN** are shown below:

```
function CHECK-EVEN(X:EVEN) return EVEN is
begin
   CHECK-TASK-EVEN.ENQUEUE(X);
   return X;
end CHECK-EVEN;


function CHECK_POS_EVEN(X:POS_EVEN) return POS-EVEN is
begin
   CHECK_TASK_POS_EVEN.ENQUEUE(X);
   return CHECK-EVEN(X);
end CHECK-POS-EVEN;
```

Notice that since each checking function interface is still the same as in the sequential case, no modifications are required for calls to these functions. This simplifies the transformation process because we can leverage on the already existing sequential system. In addition, we can also mix sequential and concurrent checking within the same program. For example, the constraint on type **EVEN** can be checked concurrently, while the constraint on type **POS-EVEN** is checked sequentially. Anna pragmas are defined later in this section to specify the kind of checking to be performed for each annotation.

If we did not maintain the checking function and instead enqueued checks directly to the checking task, we would save the time of making one function call. This problem can be solved by using the Ada pragma INLINE to specify that all checking functions that perform the actual enqueuing be expanded inline at their calls.

This scheme of using checking tasks has one problem — the checking task will not respond to a check request while it is performing a previously requested check. This will therefore cause the underlying program to block. To prevent this from happening, we generate a *buffer* task for each checking task. This task maintains a queue of check requests. The underlying program enqueues check requests to the buffer task and the checking task dequeues these requests from the buffer task and then performs the actual check. The buffer tasks also perform other activities specific to the annotation for which they have been generated. For this reason, these buffer tasks are referred to ***as secretary*** tasks. The other activities of the secretary tasks are described in the subsequent sections. A simplified version of the secretary task for **EVEN-CONSTRAINT** is shown below. This version does not include any of the secretary task's additional functionalities that will be described later[3]:

[3] In the actual implementation, secretary tasks are generic instantiations of a secretary task template.

```
task CHECK-SECY-EVEN is
   entry ENQUEUE(X:EVEN);
   entry DEQUEUE(X: out EVEN);
end CHECK_SECY_EVEN;

task body CHECK_SECY_EVEN is
   EMPTY:BOOLEAN := TRUE;
   TMP:EVEN;
   ...
begin
   loop
      select
         accept ENQUEUE(X:EVEN) do
            TMP := X;
         end ENQUEUE;
         enqueue the value of TMP and update EMPTY;
      or
         when not EMPTY =>
            accept DEQUEUE(X:out EVEN) do
               X := first element in the queue;
            end DEQUEUE;
            dequeue the first element and update EMPTY;
      end select;
   end loop;
end CHECK-SECY-EVEN;
```

The modified checking task and checking function for **EVEN-CONSTRAINT** are shown below:

```
task CHECK-TASK-EVEN;

task body CHECK-TASK-EVEN is
   X:EVEN;
begin
   loop
      CHECK_SECY_EVEN.DEQUEUE(X);
      if not (X mod 2 = 0) then
         report-error;
      end if;
   end loop;
end CHECK-TASK-EVEN;


function CHECK-EVEN(X:EVEN) return EVEN is
begin
   CHECK_SECY_EVEN.ENQUEUE(X);
   return X;
end CHECK-EVEN;
```

The interaction between the underlying program, secretary tasks and checking tasks is illustrated in Figure 2. Assume that the variable P in this figure is of type POS-EVEN.

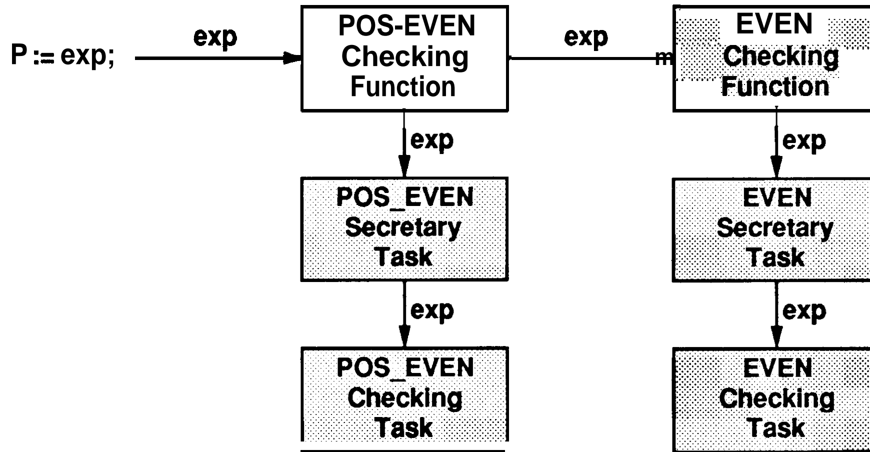

Figure 2: Checking Tasks and Secretary Tasks

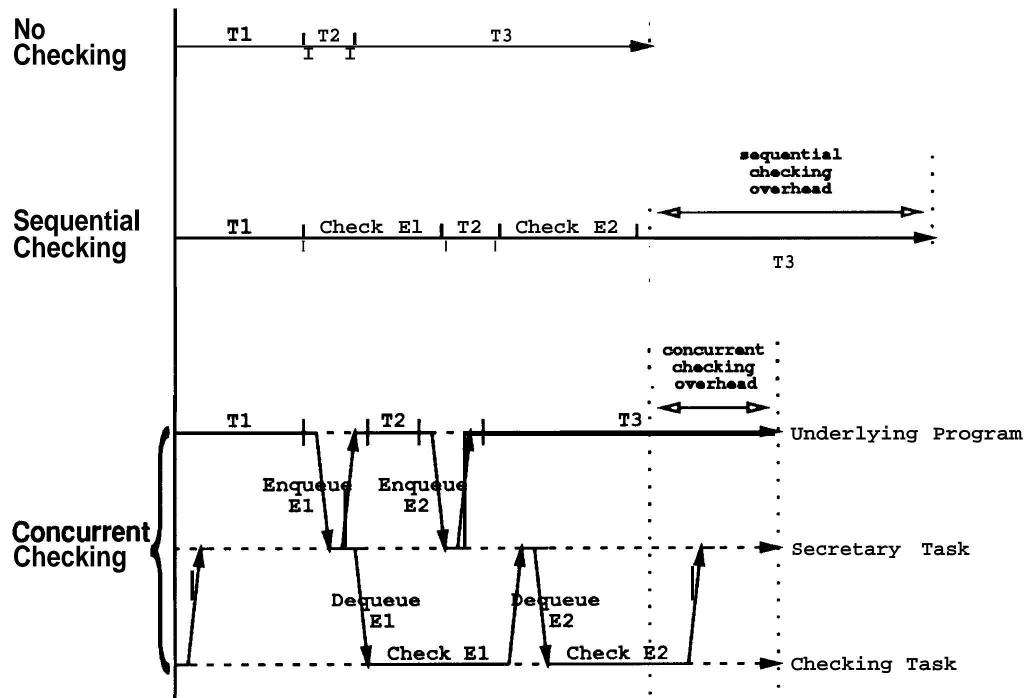

Figure 3: Timing Diagram of Check Requests for EVEN

A scenario where a program generates two check requests for variables El and E2 of type EVEN in quick succession is illustrated in Figure 3. This figure compares the performance of

the program in the three cases of (1) no runtime monitoring, (2) sequential runtime monitoring, and (3) concurrent runtime monitoring in a multi-processor machine. The dashed lines in this figure denotes that the corresponding processor is idle.

Ada is a block-structured language. In most situations, Ada requires that a thread of control cannot leave a scope in the Ada program until all local tasks created by this thread of control have terminated. Therefore, upon leaving scopes in the transformed Ada program, all local checking tasks and secretary tasks will have to be terminated. It is preferable to terminate these tasks after all requested checks have been completed. This is achieved using the checkpointing facility described in Section 5. The code generated at the end of the scope of **EVEN-CONSTRAINT** is shown below. We will elaborate more on this code later in the section on checkpointing.

```
wait until all check requests to CHECK-TASK-EVEN have been handled;
abort CHECK_TASK_EVEN,CHECK_SECY_EVEN;
```

The structure of the checking tasks and the secretary tasks is such that abort-ing these tasks terminates these tasks as quickly as using an Ada terminate alternative[4]. Such code to perform the handling of activities at the end of a scope will occur at (1) the physical end of the sequence of statements of the scope as well as of each of the exception handlers for that scope; and (2) return, exit and *goto* statements that transfer control outside of the scope. In addition, a dummy exception handler is introduced to intercept all exceptions that are not handled by the already existing exception handlers. The dummy exception handler executes the above code and then reraises the exception that it intercepted.

4.1 Anna Pragmas for Specification of Checking Methodology

A set of Anna pragmas are provided to the programmer to specify whether annotations are to be checked concurrently or sequentially. They are listed below. These pragmas have parameters to describe the mechanism for error recovery. These parameters are described in Section 6.

ANNA-PARALLEL-SCOPE: This pragma can appear anywhere in a program (but within Anna virtual code), and its effect is to generate code for concurrent checking of all annotations ***declared*** in the scope of the pragma, unless overridden by a more local pragma.

ANNASEQUENTIAL,SCOPE: Like the previous pragma, this pragma can appear anywhere in a program. Its effect is to generate code for sequential checking of all annotations declared in the scope of the pragma, unless overridden by a more local pragma.

ANNA-PARALLEL-ANNOTATION: This pragma must occur immediately after an annotation. Its effect is to generate code for concurrent checking of this annotation.

ANNASEQUENTIALANNOTATION: As in the case of pragma **ANNA-PARALLEL-ANNOTATION,** this pragma must occur immediately after an annotation. Its effect is to generate code for sequential checking of this annotation.

[4] Either **the abort or the terminate approach can be used. No study has been conducted as to which is more efficient.**

All checks pertaining to the same annotation have to be performed in the same manner — either all concurrently or all sequentially.

4.2 Multiple Checking Tasks per Annotation

It is quite easy to generalize our model to permit multiple checking tasks for each annotation. There will still be only one secretary task which queues up all the check requests. Whenever any of the checking tasks have completed a previous check, they can dequeue the next available check request. Having multiple checking tasks per annotation is very useful in situations where there are many check requests made for each annotation.

This generalization has not yet been implemented. We intend to implement it at a later time, and provide Anna pragmas to specify the number of checking tasks for each annotation.

5 Checkpoint ing

When a check is performed concurrently by a checking task, the underlying program continues to execute beyond the point where the check request was made. If the checking task detects an inconsistency, the underlying program could potentially be running in an unsafe state. If the programmer has chosen to perform a particular check concurrently, it means that the programmer felt it does not matter for the program to continue executing in such situations. However, the programmer might want to prevent the underlying program from executing beyond a certain point until certain checking tasks have processed all their pending check requests.

The ***checkpointing*** facility is used to specify such points in the program beyond which it is unsafe to continue execution until all checks have been performed. It is specified using the Anna pragma **ANNA-CHECK-POINT.** This pragma takes as its parameter a list of annotation names. The underlying program is suspended whenever it reaches an **ANNA-CHECK-POINT** pragma until ***all*** check requests pertaining to each of the annotations in the list have been processed. If the list of annotation names is empty, then the underlying program is suspended until all check requests for every annotation has been processed. By assuming that the underlying program is sequential, we have avoided many complications in defining the semantics of a checkpoint. For example, what does it mean for one task to checkpoint on an annotation, while another task continuously makes check requests on the same annotation?

Checkpoints are implemented by maintaining an *enqueue* ***counter*** and a ***dequeue counter*** for each annotation being checked concurrently. The enqueue counter is incremented every time a check request is sent to the corresponding secretary task, while the dequeue counter is incremented every time a check request is processed[5] by the corresponding checking task. The underlying program is allowed to proceed from a checkpoint when all the relevant dequeue counter values catch up with their corresponding enqueue counter values. The actual implementation of this algorithm involves more details. For example, if we are working on a multi-processing system without any shared memory, comparing counters will require some amount of message passing.

[5]The **dequeue counter maintains a count of all the checks requests that have been *processed* by the checking task, rather than a count of all the check requests that have been** *dequeued* **from the secretary task. These two numbers will differ by 1 while the checking task is processing a check request.**

Our approach is to use a dedicated ***checkpoint controller,*** which is a separate process. Every time a checking task-secretary task combination starts running, the secretary task registers itself with the checkpoint controller. Similarly, whenever these tasks are aborted at the end of their scopes, the checkpoint controller is notified.

We assume the presence of shared memory. It so happens that only the `enqueue` counters need to be placed in shared memory in our algorithm. Furthermore, the `enqueue` counters are used ***only*** by the underlying program when it is not waiting at a checkpoint; and used ***only*** by the checkpoint controller when the underlying program is waiting at a checkpoint. Hence, if we do not have shared memory, we need to exchange the values of the relevant `enqueue` counters only at checkpoints.

The `enqueue` counters are incremented by the corresponding checking functions every time a check request is made. The dequeue counters are maintained by the checkpoint controller. The secretary task periodically sends the checkpoint controller the number of completed checks since it last communicated with the checkpoint controller. The checkpoint controller then adds this number to its appropriate dequeue counter. The secretary task attempts to send these messages as infrequently as possible at the same time ensuring that the underlying program is not made to wait too long at a checkpoint. Hence, the secretary task sends such a message every time its queue of check requests is empty and the checking task is ready for more checks. Chances are that at this point the secretary task is anyway going to be idle and so it might as well use this time to send the checkpoint controller an update of its status.

Every time a checkpoint is reached, the underlying program makes an entry call to the checkpoint controller, giving the checkpoint controller the list of annotations to checkpoint on. Since the underlying program is sequential, the `enqueue` counters cannot be incremented while the underlying program is waiting at a checkpoint — the `enqueue` counters are constant for the duration of the checkpoint. The checkpoint controller therefore waits for update messages to the corresponding dequeue counters from their respective secretary tasks until all the dequeue counters becomes equal to their corresponding `enqueue` counters. The underlying program is then released from the checkpoint. The secretary tasks are not allowed to manipulate the dequeue counters directly because there could be times when the checkpoint controller and the secretary task attempt to access the dequeue counter simultaneously.

The secretary task and checking function for **CHECK-EVEN** are shown below after modification to handle checkpointing. Note the use of the local counter **CHECKS-PROCESSED.** This counter maintains a count of all the processed check requests since the last time the secretary task communicated with the checkpoint controller. Actually, this counter is incremented as soon as a check request has been dequeued, i.e., earlier than it should have been incremented. However, by the time the secretary task processes the next dequeue request from the checking task, this counter contains the correct value.

```
task CHECK_SECY_EVEN is
   entry ENQUEUE(X:EVEN);
   entry DEQUEUE(X: out EVEN);
end CHECK_SECY_EVEN;
```

```
task body CHECK_SECY_EVEN is
    EMPTY:BOOLEAN := TRUE;
    TMP:EVEN;
    CHECKS-PROCESSED:NATURAL := 0;
    ...
begin
    register EVEN-CONSTRAINT with checkpoint controller;
    loop
        select
            accept ENQUEUE(X:EVEN) do
                TMP := X;
            end ENQUEUE;
            enqueue the value of TMP and update EMPTY;
        or
            accept DEQUEUE(X:out EVEN) do
                if EMPTY then
                    if CHECKS-PROCESSED /= 0 then
                        send checkpoint controller the value of
                                CHECKS-PROCESSED;
                        CHECKS-PROCESSED := 0;
                    end if;
                    accept ENQUEUE(Y:EVEN) do
                        TMP := Y;
                    end ENQUEUE;
                    enqueue the value of TMP and update EMPTY;
                end if;
                X := first element in the queue;
            end DEQUEUE;
            dequeue the first element and update EMPTY;
            CHECKS-PROCESSED := CHECKS-PROCESSED+1;
        end select;
    end loop;
end CHECK_SECY_EVEN;


function CHECK-EVEN(X:EVEN) return EVEN is
begin
    CHECK_SECY_EVEN.ENQUEUE(X);
    ENQUEUE_COUNTER(EVEN_CONSTRAINT) :=
            ENQUEUE_COUNTER(EVEN_CONSTRAINT) + 1;
    return X;
end CHECK-EVEN;
```

The code to handle a checkpoint on **EVEN-CONSTRAINT** is shown below:

```
CHECKPOINT_CONTROLLER.CHECKPOINT(EVEN_CONSTRAINT);
```

The relevant portion of the checkpoint controller is shown below. Note that the enqueue and dequeue counters are zeroed to prevent the counters from overflowing:

```
accept CHECKPOINT(ANNOTATION: ANNOTATION_TYPE) do
   while DEQUEUE_COUNTER(ANNOTATION) /=
         ENQUEUE_COUNTER(ANNOTATION) loop
      wait for corresponding secretary task to send CHECKS-PROCESSED;
      DEQUEUE_COUNTER(ANNOTATION) :=
         DEQUEUE_COUNTER(ANNOTATION) + CHECKS_PROCESSED;
   end loop;
   ENQUEUE_COUNTER(ANNOTATION) := 0;
end CHECKPOINT;
DEQUEUE_COUNTER(ANNOTATION) := 0;
...
```

The above is a simplified version of the checkpoint controller. In general, the checkpoint controller will have to handle multiple annotations at the same time. The timing diagram in Figure 4 illustrates the checkpointing algorithm. The scenario is similar to the one depicted in Figure 3.

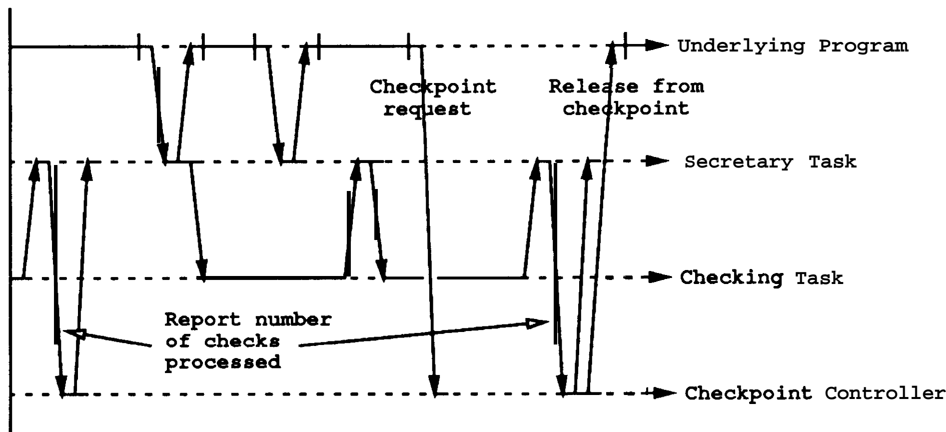

Figure 4: Timing Diagram of a Checkpoint

In Section 4, we discussed the need to abort all local checking and secretary tasks when control reaches the end of their scopes. The code presented there is refined below:

```
CHECKPOINT_CONTROLLER.CHECKPOINT(EVEN_CONSTRAINT);
tell checkpoint controller that EVEN-CONSTRAINT no longer exists;
abort CHECK-TASK-EVENCHECK-SECY-EVEN;
```

5.1 An Alternate Checkpointing Algorithm

We now describe another checkpointing strategy. No counters are required in this case, but we still make the assumption that the underlying program is sequential. This strategy is outlined

below.

Every time the queue of the secretary task is empty and the corresponding checking task is ready for more check requests (i.e., the checking task is idle), the secretary task communicates this information to the checkpoint controller. The secretary task also communicates with the checkpoint controller as soon as its queue becomes non-empty. The queue becomes non-empty when an enqueue is performed by the underlying program. In this algorithm it is essential for the secretary task to report to the checkpoint controller before releasing the underlying program from the ENQUEUE entry call. Hence the checkpoint controller is aware of when each checking task is idle and when it is not. The underlying program performs an entry call to the checkpoint controller at the checkpoint and communicates the list of annotations on which to checkpoint.

The checkpoint controller releases the underlying program from the checkpoint as soon as the checkpoint controller knows that all relevant checking tasks are idle. This method will work because once the underlying program is at a checkpoint, it cannot enqueue any further checks. Also, all earlier check requests have already been enqueued with the secretary tasks, and if necessary the checkpoint controller has been informed about it. Hence, the checkpoint controller just has to wait until all relevant checking tasks are idle.

Though this method is conceptually much simpler, it has some disadvantages:

- In this method, all communication has to be synchronous. This is because we rely on the fact that when the underlying program is waiting at a checkpoint, all previous enqueue requests have reached the corresponding secretary tasks and the checkpoint controller has also been notified of this.

  In the case of the counter implementation, the enqueue operation between the underlying program and the secretary task can be performed asynchronously (possibly using some meta-Ada construct). This is because the counter values at checkpoints are independent of whether or not previous check requests have reached the corresponding secretary task.

- In a situation where there are a large number of annotations to be checked, but very few check requests per annotation, checking tasks will become idle frequently. This could swamp the checkpoint controller with a large amount of communication from secretary tasks. Since a lot of this communication takes places while the underlying program is waiting to be released from an ENQUEUE entry call, this will slow down the underlying program.

5.2 Multiple Checking Tasks per Annotation

If we have one checking task per annotation, we know that the value of **CHECKS-PROCESSED** is correct whenever the checking task performs a dequeue operation.

When there are multiple checking tasks for each annotation, this is not enough. All other checking tasks also have to become idle before we have a correct value in **CHECKS-PROCESSED.** Designing a scheme to handle this is a topic for future research.

6 Error Reporting and Recovery

The Anna reference manual requires that the exception **ANNA-ERROR** be raised whenever an inconsistency with respect to an annotation is detected. Furthermore, this exception must be raised at the location of the inconsistency. In the sequential checking methodology, this is easy to implement. In fact, in the sequential case, control can also be transferred to a specialized Anna Debugger [8], through which further information about the inconsistency can be obtained.

When the checks are performed concurrently, it may be too restrictive to adhere to the Anna semantics of raising the exception **ANNA-ERROR** at the location of the inconsistency. To implement this, the program execution has to be backtracked to the location from where the failed check request was made. We have not implemented the backtracking scheme due to the complexities involved. However, there are many other schemes of error reporting and recovery which are easier to implement and at the same time quite useful. We have implemented the three schemes listed below. The programmer can choose a particular scheme of error reporting and recovery for each annotation.

- ***Inconsistencies are ignored.***
  In this scheme, the checking task makes a log of the inconsistency, but takes no other action. This scheme is useful in non-critical situations, where the log of the inconsistency can be used for routine maintenance of the underlying program. Checkpoints with respect to an annotation being checked using this scheme are ignored.

- ***Inconsistencies are reported to the underlying program by raising the exception*** ANNA_ **ERROR** ***at the next checkpoint.***
  In this scheme, the secretary task reports any inconsistencies to the checkpoint controller every time the secretary task sends an update on the number of processed checks. The checkpoint controller then causes the exception **ANNA-ERROR** to be raised in the underlying program when it releases it from the next checkpoint. This scheme is useful when the underlying program wishes to recover from an inconsistency.

- ***The first detection*** of ***an inconsistency causes the entire program to be aborted as soon as possible.***
  On detecting an inconsistency, the checking task initiates the process of aborting the underlying program. The main complication in achieving this is due to the Ada visibility rules. The underlying program is not necessarily visible to all the checking tasks (or secretary tasks). Our solution to this problem is discussed later.

In all three schemes, the inconsistencies are logged into a file. The default is the current (standard) output, but the programmer can specify an alternate file using the Anna pragma **ANNA-PARALLEL-LOG-FILE.** This pragma takes a filename as its argument.

To specify the error reporting and recovery scheme, the programmer may use the Anna pragma **ANNA-REPORT-MODE.** This takes as argument one of **IGNORING, REPORTING** or **ABORTING'.** The effect of this pragma is to set the default scheme for error reporting and

[6]The "-ING" **at the end of each of these possible options is to work around the fact that "ABORT" is an Ada reserved word.**

recovery throughout its scope except where it is overridden by a more local pragma that specifies an error reporting and recovery strategy.

These error reporting and recovery options may also be specified as parameters in the four Anna pragmas described in Section 4.1. This will have the same effect as a pragma **ANNA-REPORT-MODE** at the same location with the same parameter.

The following paragraphs go into details of the implementation of the **REPORTING** and **ABORTING** schemes.

In the case of the **REPORTING** scheme, if the checkpoint controller determines that ANNA_ **ERROR** must be raised in the underlying program the following Ada trick is used: The checkpoint controller raises the exception **ANNA-ERROR** inside the accept statement that is handling the checkpoint. This exception is then allowed to propagate outside the accept statement. This causes the exception to be reraised at the location of the checkpoint in the underlying program. The exception is also reraised within the checkpoint controller outside the accept statement, but the checkpoint controller "kills" this second propagation using an exception handler. An outline of the checkpoint controller code that achieves this is shown below:

```
...
begin
   accept CHECKPOINT(ANNOTATION:ANNOTATION_TYPE) do
      ...
      if MODE = REPORTING then
         raise ANNA-ERROR;
      end if;
   end CHECKPOINT;
exception
   when ANNA-ERROR =>
      null;
end;
...
```

We can achieve the effect of raising **ANNA-ERROR** in the underlying program through other means also. For example, the entry **CHECKPOINT** can return a parameter based on which the underlying program can determine whether or not to raise the exception. However, the method we have adopted is more general in that the checkpoint controller could potentially raise an exception of its choice and not just **ANNA-ERROR.**

Before we go into the details of how the **ABORTING** scheme is implemented, we discuss the different ways in which the underlying program can terminate. In general there are three ways:

1. ***Normal termination.***
   The underlying program terminates normally after all consistency checks have been completed successfully.

2. ***A bnormal termination.***
   An exception was raised within the underlying program during its execution and was not handled. This could either be an Ada exception, or the exception **ANNA-ERROR** raised at a checkpoint due to an annotation being checked using the **REPORTING** scheme.

3. ***Termination due to inconsistency.***
   This happens when an inconsistency is detected with respect to an annotation being checked using the **ABORTING** scheme.

To handle these three situations, the original main program of the underlying program is transformed into a task. This task is then placed inside a procedure that becomes the new main program. This new main program waits for one of the above three situations to arise and takes the necessary action to terminate the underlying program and the monitoring tasks.

In the runtime system, we place a task with the two entry calls, **SIGNAL-END** and **WAIT-END.** These entries have one parameter each. This parameter is used to specify the kind of program termination. All this task does is wait for an entry call to **SIGNAL-END** and copies the parameter over; it then waits for an entry call to **WAIT-END** and returns this copied parameter to the task that issued the **WAIT-END** entry call. This task is shown below:

```
type TERMINATION is (NORMAL, ABNORMAL, INCONSISTENCY);

task PGM_CONTROLLER is
   entry SIGNAL-END(SIGNAL : in TERMINATION);
   entry WAIT-END(SIGNAL : out TERMINATION);
end PGM_CONTROLLER;

task body PGM_CONTROLLER is
   END-SIGNAL : TERMINATION;
begin
   accept SIGNAL-END(SIGNAL : in TERMINATION) do
      END-SIGNAL := SIGNAL;
   end SIGNAL-END;
   accept WAIT_END(SIGNAL : out TERMINATION) do
      SIGNAL := END-SIGNAL;
   end WAIT-END;
end PGM_CONTROLLER;
```

The new main program makes an entry call to **WAIT-END** and waits until this entry call is processed by the task PGM_CONTROLLER. Hence, if any information needs to be communicated to this main program, all one has to do is to make an entry call to SIGNAL-END with the appropriate parameter. This works around the Ada visibility rules restriction due to which direct access to the main program is not permitted.

The way in which the new main program handles each of the different possible termination schemes is described below:

NORMAL: **SIGNAL-WAIT** is called by the underlying program when it reaches a normal termination point. The new main program simply terminates.

ABNORMAL: An exception handler is inserted to catch all exceptions that are not handled within the underlying program. Within this exception handler, SIGNAL-WAIT is called. Following this, the previously mentioned trick (to propagate an exception across tasks) is

used to propagate the exception to the new main program. The new main program can then mimic the abnormal termination of the underlying program.

INCONSISTENCY: As soon as a checking task detects an inconsistency with respect to an annotation being checked using the **ABORTING** scheme, the checking task calls **SIGNAL-END.** The new main program then aborts the underlying program and then terminates itself by raising the exception **ANNA-ERROR.**

As part of the termination process, the new main program also terminates the checkpoint controller. The mainline of the underlying program before and after transformation is shown below:

Before:

```
procedure MAIN is
   ...
end MAIN;
```

After:

```
procedure MAIN is
   -- This is the new main program.
   END-SIGNAL : TERMINATION;

   task MAIN-PGM is
      entry PROPAGATE-ADA-ERROR;
   end MAIN_PGM;

   task body MAIN-PGM is
      procedure MAIN is
         -- The original main program transformed to include checking
         -- tasks, secretary tasks, entry calls to these tasks, etc.
         ...
      end MAIN;
   begin
      MAIN;
      -- Normal termination. New main program should terminate nor-
      -- mally.
      PGM-CONTROLLER. SIGNAL-END(NORMAL);
   exception
      -- Abnormal termination due to an exception raised within the
      -- underlying program. New main program should also terminate
      -- abnormally by raising the same exception.
      when others =>
         PGM-CONTROLLER. SIGNAL-END(ABNORMAL);
```

```
            -- The same trick that was used in implementing the RE-
            -- PORTING scheme is used here. Note that in this case,
            -- we do not know the exception being propagated.
            begin
               accept PROPAGATE-ADA-ERROR do
                  raise;
               end PROPAGATE-ADA-ERROR;
            exception
               when others =>
                  null;
            end;
         end MAIN_PGM;

      begin
         PGM_CONTROLLER.WAIT_END(END_SIGNAL);
         terminate checkpoint controller;
         if END-SIGNAL = NORMAL then
            -- Normal termination.
            null;
         elsif END-SIGNAL = ABNORMAL then
            -- Find out and raise the exception raised in the original main
            -- program.
            MAIN_PGM.PROPAGATE_ADA_ERROR;
         elsif END-SIGNAL = INCONSISTENCY then
            -- A checking task has detected a violation and the error reporting
            -- and recovery scheme is ABORTING.
            abort MAIN_PGM;
            raise ANNA-ERROR;
         end if;
      end MAIN;
```

This concludes our description of the translation schemes to generate a concurrent monitoring system from a formally specified program in Anna. Figure 5 illustrates all the processes and their interactions in this monitoring system. The box labeled **UNDERLYING PROGRAM** is also a separate process since it is invoked from the task **MAIN-PGM.**

7 Experimental Results

Many small test examples have been transformed and executed successfully using the concurrent monitoring system. These tests were designed to test the function&ties of the system thoroughly. Some examples of experiments conducted were (1) mixing sequential and concurrent checking, (2) mixing different kinds of error reporting and recovery techniques, and (3) checkpointing under various different circumstances. We have found that the monitoring system is quite sturdy and usable.

The other realm of experiments were concerned with performance issues. We were interested in comparing the performance of the underlying program in the three cases of running without any checking; with sequential checking; and with concurrent checking. We performed these

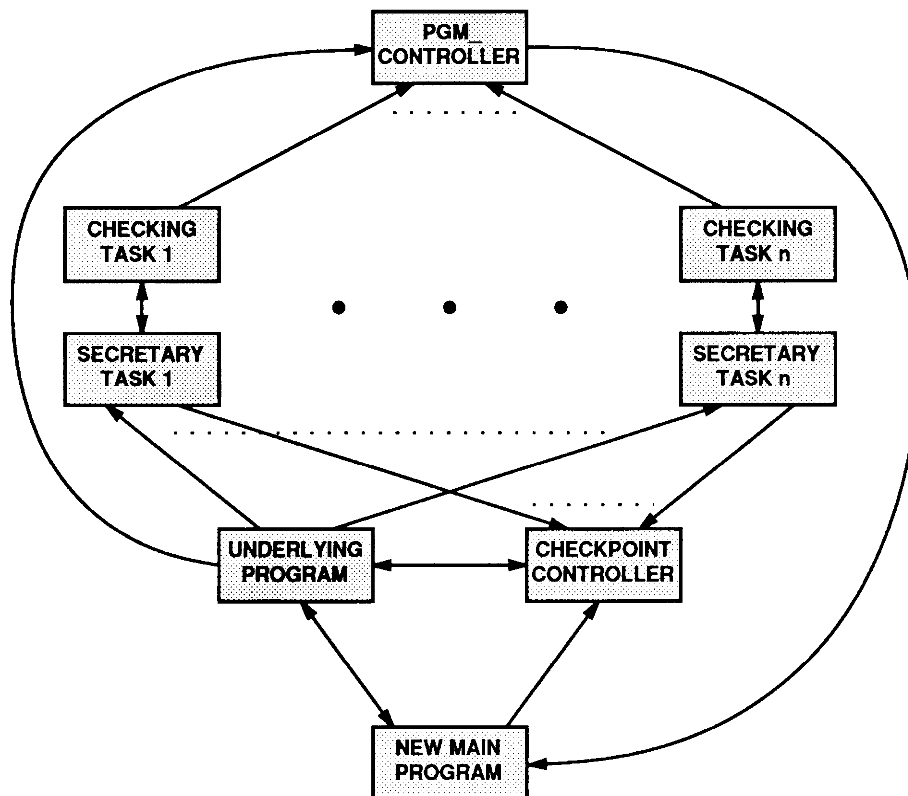

Figure 5: The Concurrent Monitoring System

experiments on a Sequent Symmetry machine with 12 processors. Though we did get the anticipated results — ***no checking*** < ***concurrent checking*** < ***sequential*** *checking*[7] — the lack of good instrumentation tools have hampered our attempts at obtaining precise timing information from our experiments.

Some example programs which have been run on this system include a ***Prime Number Generation*** program, and a ***Sorted Array*** program. The first program had a constraint on a data type that all values of the type be prime. When larger numbers were fed to the prime number checking algorithm, the concurrently monitored program completed more quickly than the sequentially monitored program.

The second program had a constraint on an array type that restricted all components to be sorted in a non-decreasing order. For small arrays, the sequentially monitored program ran more quickly, but once the array size became sufficiently large, the concurrently monitored program completed quicker than the sequentially monitored program.

[7]These **results are for large programs with a large overhead in annotation checking.**

7.1 A Performance Degradation Model

In this section, we develop a model using which we can quantitatively analyze the various performance degradation parameters of a runtime monitoring system.

For any monitoring system ***m***, let us define the following performance degradation parameters:

- t_m^o
  This is the time overhead due to the initialization and termination of the monitoring system. Even if there were no annotations to be checked, the transformed program will take this much longer to execute. In the system described in this paper, this includes the time take to create and terminate tasks like the checkpoint controller and the new main program, initializing the task within which the underlying program will run and the time taken to communicate with the new main program through the **SIGNAL-END** and **WAIT-END** entry calls.

- $t_m^{o_a}$
  Here, ***a*** is an annotation. This is the time overhead due to the presence of ***a*** in the program. In a program where ***a*** is the only annotation, but no checks are performed, the transformed program will take t_m^o + $t_m^{o_a}$ longer than the original program to execute. In the system described in this paper, $t_m^{o_a}$ includes the time taken to create and terminate the secretary task and the checking task of ***a***.

- $t_m^{c_a}$
  This is the amount of time by which the underlying program is slowed down every time a check is performed with respect to ***a***. In the system described in this paper, this is the time required to perform the enqueue operation and then to increment the enqueue counter. In the sequential checking system presented in Section 3, this time includes that taken to make a function call and then to perform the actual check.

  In general, $t_m^{c_a}$ will depend on the value being checked. For example, the annotation **IS-PRIME(X)** can take more time to check for larger values of *X*. To avoid complicating our model, we redefine $t_m^{c_a}$ to be the ***average*** amount of time by which the underlying program is slowed down every time a check is performed with respect to ***a***.

- n^a
  For any program, this is the number of checks with respect to ***a*** that needs to be performed. This number is independent of the monitoring system ***m***, but will vary depending on the particular program run. Here again, n^a can be considered the average number of checks with respect to ***a*** over the program runs which are of interest.

If the program contains annotations ***al***,. . . , a_k, then the performance degradation due to a monitoring system m is:

$$t_m^o + \sum_{i=1}^{k}\left(t_m^{o_{a_i}} + n^{a_i} \times t_m^{c_{a_i}}\right) \tag{1}$$

If m_{con} is the concurrent monitoring system and m_{seq} is the sequential monitoring system, then it make sense to use the concurrent monitoring system over the sequential monitoring system when:

$$(t^{o}_{m_{con}} - t^{o}_{m_{seq}}) + \sum_{i=1}^{k}\Big((t^{oa_i}_{m_{con}} - t^{oa_i}_{m_{seq}}) + n^{a_i} \times (t^{ca_i}_{m_{con}} - t^{ca_i}_{m_{seq}})\Big) \leq \mathbf{0} \qquad (2)$$

The above inequality is obtained from (1) by substituting m_{con} and m_{seq} respectively for ***m*** and then comparing the two. This inequality can be simplified by making a few assumptions. We can assume that t^{oa}_{m} is independent of the annotation ***a.*** Therefore, we rewrite t^{oa}_{m} as t^{ao}_{m}[8]. We also split t^{ca}_{m} into two components — the constant overhead due to setting up to perform the check, which we denote by t^{coa}_{m} and the overhead due to actually performing the check, which we denote by t^{cha}_{m}. We can assume that t^{coa}_{m} is independent of the annotation ***a*** (hereafter, this is referred to as t^{co}_{m}), and that $t^{cha}_{m_{con}} = 0$ [9]. Then (2) simplifies to:

$$(t^{o}_{m_{con}} - t^{o}_{m_{seq}}) + (t^{ao}_{m_{con}} - t^{ao}_{m_{seq}}) \times k + (t^{co}_{m_{con}} - t^{co}_{m_{seq}}) \times \sum_{i=1}^{k}\left(n^{a_i}\right) \leq \sum_{i=1}^{k}\left(n^{a_i} \times t^{cha_i}_{m_{seq}}\right) \qquad (3)$$

This inequality states that if the total time required to perform the ***actual*** checks is greater than the extra overhead of the concurrent system over the sequential system, the concurrent system should be used. It should be clear that this inequality assumes that the monitoring system does not interfere with the underlying program except when it holds up the underlying program during the enqueueing of check requests — i.e., all annotations are checked in **IGNORING** mode and the underlying program is never held up at a checkpoint.

We have attempted to determine values for the above performance degradation parameters for both the sequential and concurrent monitoring systems using a variety of typical annotations. Due to lack of good instrumentation tools, the values we obtained are not precise enough to be very meaningful in the scope of this paper.

We propose to refine our experiments to come up with more meaningful results. We hope to characterize for each annotation from a representative set when the conditions under which concurrent monitoring improves the performance over sequential monitoring.

8 Conclusions and Future Work

Our efforts in developing a permanent runtime monitoring system have been quite fruitful. We demonstrated the feasibility of using multi-processor machines to perform concurrent monitoring of critical aspects of a program's behavior continuously while it is running. We have an actual implementation of such a system for Anna subtype annotations. Since this system

[8] Here we also assume that the total number of tasks generated does not cause the Ada runtime scheduler to degrade. Otherwise, t^{oa}_{m} would be a function of how many checking tasks and secretary tasks have already been created in the program.

[9] We are making a simplifying assumption here that the underlying program never has to wait for a check to be completed. For example, we assume that if the underlying program reaches a checkpoint, all relevant checks have already been performed and the underlying program is not held up.

bootstraps on top of the already existing and well-developed sequential monitoring system, the concurrent monitoring system is able to handle most Ada constructs.

Notwithstanding the success of our project, we have identified several areas where further work is required.

- ***Extension of the concurrent monitoring system to handle more Anna constructs.***
  Extending the concurrent monitoring system to handle Anna object annotations requires the implementation of a similar methodology to that we have already implemented. However, there are other aspects that we have to spend more time on to design efficient methodologies. For example, an annotation on a complex data-structure may require saving a copy of this data-structure every time a check request is made. This might introduce an unacceptable overhead. Techniques need to be developed to save only those portions of the data structure that the check is based on. Alternatively, portions of the data structure are saved only when they are modified while there is a pending check request based on the old value of the data structure.

- ***Extension of the concurrent monitoring system to handle underlying programs with multiple threads of control.***
  When the underlying program is not sequential, many of our algorithms need to be modified before they can work properly. In addition, many of the features we have developed will also have to be refined. For example, what happens when one thread of control in the underlying program requests a checkpoint on a particular annotation, while another thread of control is enqueuing check requests repeatedly on the same annotation? We might need to permit checkpointing based on check requests from a particular thread of control, or from a particular subprogram in the underlying program. Similarly, our error recovery strategies may also require modification. For example, it may not make sense to report an inconsistency based on a check request from one thread of control to some other thread of control.

- ***Dynamic modification of checking and error reporting/recovery schemes.***
  In many complex systems, it may be useful to have the ability to dynamically change the method of checking and the method of error reporting and recovery. For example, the checking of an annotation could be suppressed completely, or changed from sequential checking to concurrent checking. Also, inconsistencies with respect to the same annotation may be critical in some situations (in which case, the program may have to be immediately aborted), and not so critical in other situations (in which case the annotation violation is just logged to a file).

- ***Design of a new specification monitoring language.***
  Our studies have shown the necessity for designing a new language in which one can not only specify the program behavior, but also the degree of concurrency with which checking can take place and the method of error reporting and recovery. Our Anna pragmas have extended Anna in a very rudimentary manner to specify these other aspects of runtime monitoring. This new language need not be based on any particular programming language like Ada, but can provide a general paradigm for specification monitoring of programs written in any language.

- ***Characterization and implementation of real-life examples where runtime monitoring is useful.***
  In this paper, we listed a few possible applications of our monitoring system. These applications and others will have to be implemented for runtime monitoring to test the feasibility of concurrent monitoring on real-life examples.

- ***Better performance metrics and measurements.***
  Especially with the design of a new and more sophisticated specification monitoring language, our simple performance degradation parameters will have to be extended. We also have to create the necessary framework in which we can measure these parameters precisely.

9 Acknowledgements

We are grateful to Prof. David C. Luckham, David S. Rosenblum and other members of the Program Analysis and Verification Group at Stanford for the ideas generated through our many interactions with them. Special thanks are due to Geoff Mendal for his extensive proof-reading of the paper.

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

References

[1] ***The Ada Proqramming Language Reference Manual.*** US Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

[2] Dines Bjørner and Cliff B. Jones. ***Formal Specification and Software Development.*** Prentice/Hall International, 1982.

[3] R. Bloomfield, L. Marshall, and R. Jones, editors. ***VDM'88, VDM — The Way Ahead.*** Volume 328 of ***Lecture Notes in Computer Science,*** Springer-Verlag, 1988.

[4] D. E. Denning. An intrusion-detection model. In ***Symposium on Security and Privacy,*** IEEE Computer Society, 1986.

[5] D. E. Denning and P. J. Denning. Data security. ***ACM Computing Surveys,*** 11(3):227–249, September 1979.

[6] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. ***IEEE Software,*** 2(5):24–36, September 1985.

[7] D. Lenat and R. V. Guha. ***Building Large Knowledge Based Systems.*** Addison Wesley, 1989.

[8] D. C. Luckham, S. Sankar, and S. Takahashi. ***Two Dimensional Pinpointing: An Application of Formal Specification to Debugging Packages.*** Technical Report CSL-TR-89-379, Stanford University, April 1989. To appear in IEEE Software.

[9] D. C. Luckham, F. W. von Henke, B. Krieg-Brückner, and *0. Owe. **Anna — A Language for Annotating Ada Programs.*** Springer-Verlag — Lecture Notes in Computer Science No. ***260,*** July 1987.

[10] D. S. Rosenblum, S. Sankar, and D. C. Luckham. Concurrent runtime checking of annotated Ada programs. In ***Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science,*** pages 10-35, Springer-Verlag — Lecture Notes in Computer Science No. 241, December 1986. (Also Stanford University Computer Systems Laboratory Technical Report No. 86-312).

[11] S. Sankar. ***Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.*** PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN-CS-89-1282, and Computer Systems Laboratory Technical Report No. CSL-TR-89-391.

[12] S. Sankar. A note on the detection of an Ada compiler bug while debugging an Anna program. ***ACM SIGPLAN,*** 24(6):23–31, 1989.

[13] S. Sankar, D. S. Rosenblum, and R. B. Neff. An implementation of Anna. In ***Ada in Use: Proceedings of the Ada International Conference, Paris,*** pages 285-296, Cambridge University Press, May 1985.

[14] J. M. Spivey. ***Understanding Z, A Specification Language and its Formal Semantics.*** Cambridge Unversity Press, 1988. Tracts in Theorectical Computer Science, Volume 3.

[15] L. G. Stucki and G. L. Foshee. New assertion concepts for self-metric software validation. In ***Proceedings of the International Conference on Reliable Software,*** pages 59-65, April 1975.

[16] S. S. Yau and R. C. Cheung. Design of self-checking software. In ***Proceedings of the International Conference on Reliable Software***, pages 450-457, April 1975.

A A Sample Transformation

The following Ada program includes many constructs that have been used in the examples of this paper. The complete transformation of this program is shown later.

```
--: pragma ANNA-REPORT-MODE(REPORTING);
procedure P is
   type EVEN is new INTEGER;
   --| <<EVEN-CONSTRAINT>>
   --| where X:EVEN =>
   --|      X mod 2 = 0;
   subtype POS_EVEN is EVEN;
   --| <<POS_EVEN_CONSTRAINT>>
   --| where X:POS_EVEN =>
   --|      X > 0;
```

```
    I:EVEN := 2;
    J:POS_EVEN := 6;
begin
    I := 4;
    --: pragma ANNA-CHECK-POINT(EVEN-CONSTRAINT);
    J .- .- 2;
end P;
```

The Anna program after transformation is shown below. Note the use of the **END-OF-SCOPE** procedure to encapsulate everything that has to be done at the end of the scope of procedure **P**, and also note how all exit points (including abnormal exit points) are handled. The details of the checking tasks, checking secretaries, etc. have been omitted to save space. The actual code generated by our tool differs in minor details, but these details are beyond the scope of this paper.

```
procedure P is
    END-SIGNAL:TERMINATION;
    task MAIN-PGM is
        entry PROPAGATE-ADA-ERROR;
    end MAIN_PGM;
    task body MAIN-PGM is
        procedure P is
            type EVEN is new INTEGER;
            subtype POS_EVEN is EVEN;
            I:EVEN := 2;
            J:POS_EVEN := 6;
            task CHECK_SECY_EVEN is ...
            task body CHECK_SECY_EVEN is ...
            task CHECK-TASK-EVEN;
            task body CHECK-TASK-EVEN is . . .
            function CHECK-EVEN(X:EVEN) return EVEN is . . .
            task CHECK_SECY_POS_EVEN is ...
            task body CHECK_SECY_POS_EVEN is ...
            task CHECK-TASK-POS-EVEN;
            task body CHECK-TASK-POS-EVEN is . . .
            function CHECK_POS_EVEN(X:POS_EVEN) return POS_EVEN is . . .
            procedure END-OF-SCOPE is
            begin
                CHECK_POINT_CONTROLLER.CHECK_POINT(EVEN_CONSTRAINT,
                                                   POS-EVEN-CONSTRAINT);
                tell checkpoint controller that EVEN-CONSTRAINT and
                POS-EVEN-CONSTRAINT no longer exist;
                abort CHECK_TASK_EVEN,CHECK_SECY_EVEN,
                      CHECK_TASK_POS_EVEN,CHECK_SECY_POS_EVEN;
            end END-OF-SCOPE;
        begin
            I := CHECK-EVEN(4);
            CHECK_POINT_CONTROLLER.CHECK_POINT(EVEN_CONSTRAINT);
```

```
            J := CHECK-POS-EVEN(2);
            END-OF-SCOPE;
        exception
            when others =>
                END-OF-SCOPE;
                raise;
        end P;
    begin
        P;
        PGM_CONTROLLER.SIGNAL_END(NORMAL);
    exception
        when others =>
            PGM_CONTROLLER.SIGNAL_END(ABNORMAL);
            begin
                accept PROPAGATE-ADA-ERROR do
                    raise;
                end PROPAGATE-ADA-ERROR;
            exception
                when others =>
                    null;
            end;
    end MAIN_PGM;
begin
    PGM_CONTROLLER.WAIT_END(END_SIGNAL);
    terminate checkpoint controller;
    if END-SIGNAL = NORMAL then
        null;
    elsif END-SIGNAL = ABNORMAL then
        MAIN_PGM.PROPAGATE_ADA_ERROR;
    elsif END-SIGNAL = INCONSISTENCY then
        abort MAIN-PGM;
        raise ANNA-ERROR;
    end if;
end P;
```