Implementation of the Sentry System

Sarah E. Chodrow Mohamed G. Gouda

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{sal,gouda}@cs.utexas.edu

10 May 1994

Abstract

The sentry of a concurrent program P is a program that observes the execution of P, and issues a warning if P does not behave correctly with respect to a given set of logical properties (due to a programming error or a failure). The synchronization between the program and sentry is such that the program never waits for the sentry, the shared storage between them is very small (in fact linear in the number of program variables being observed), and the snapshots read by the sentry are consistent. To satisfy these three requirements, some snapshots may be overwritten by the program before being read by the sentry. We develop a family of algorithms that preserve these requirements for properties involving scalar variables, then extend the algorithms to permit the observation of large data structures without additional overhead. We describe in detail the annotation language with which the properties can be expressed, and a prototype system that we have implemented to generate the sentry automatically for any given concurrent C program. Finally, we present experimental results that show that the overhead incurred by the sentry is on average no worse than 10% for snapshots of up to 6 variables, and that the loss of snapshots prevents the sentry's detection of an single violation in less than 4% of the cases. Recurring errors are detected at a rate of 100%.

Keywords: run-time monitoring, assertion checking, parallel and distributed systems.

1 Introduction

The sentry of a concurrent program P is a program that observes the execution of P and determines whether P is behaving "correctly". The correct behavior of P is determined by the

execution's compliance with specified logical properties of P. A sentry can observe two classes of properties–safety properties and progress properties. Safety properties define what program states are valid as P executes. Progress properties define a set of state transitions that will occur during execution.

In designing an observer to observe the logical properties of programs, a compromise has to be made between two conflicting requirements: the precision of observation on one hand, and the cost of observation on the other. For example, an observer that keeps detailed traces of program execution requires a large amount of storage and may greatly reduce the speed of program execution. In general, observers that provide high precision are costly in terms of needed storage and execution speed.

The sentry is a low precision, low cost observer, that evaluates safety and progress properties by reading snapshots of the program state. The low cost of the sentry system is achieved by requiring that the observer, or the sentry, satisfy the following:

- *Linear storage:*
  The amount of shared storage between the program being observed and the sentry is small, in fact linear in the number of program variables being observed.
- *Wait freedom:*
  The program being observed never waits for the sentry. Thus, the execution speed of the program is completely independent of the execution speed of the sentry.
- *Mutual exclusion:*
  Access to the shared storage is synchronized to guarantee that every snapshot read by the sentry is consistent.

Most observer systems do not satisfy one or more of these requirements. For example, real-time monitors [6] [9] do not satisfy the linear-storage requirement. Recovery block systems [2] [10] do not satisfy the wait-freedom requirement. Neither the linear-storage requirement nor the wait-freedom requirement is satisfied in many low-level debuggers [11][16] or in-line assertion checkers such as Anna [12] and Gypsy[7].

In order to satisfy these requirements in the sentry system, the program being observed is allowed to overwrite a snapshot in the shared storage (between the program and the sentry), before the sentry reads that snapshot. In other words, some snapshots can be lost before they are checked by the sentry, hence the low precision of our system.

Despite the loss of observations, the sentry system has the following correctness properties:

- **Soundness**
  If the sentry detects a violation, then a violation has occurred in the program execution.

- **Completeness**
  If a violation occurs and persists in the program execution, then the sentry will detect the violation.

Although snapshots can be lost in the sentry system, our experimental results demonstrate that the loss of a snapshot can be compensated for by subsequent snapshots in most cases. I.e. if a snapshot that reveals a violation is lost, the probability that subsequent snapshots will reveal the same or other violations is very high.

As mentioned earlier, the sentry observes both safety and progress properties. These properties can be either local or global. A property of program P is *local* if it refers to variables in a single process in P. A property of P is *global* if it refers to variables in two or more processes in P. The sentry that observes the correctness of local properties is called a *local sentry.* The sentry that observes the correctness of global properties is called a *global sentry.*

Local and global properties that involve only scalar variables are propositional, and thus contain no quantification. To allow predicates to be checked, we incorporate into the sentry system the observation of larger data structures over which quantification can be defined. We introduce the observation of predicates into both the local and global sentry systems.

The rest of this paper is organized as follows. We discuss sentries for local properties in Section 2, and sentries for global properties in Section 3. We extend local and global sentries with limited quantification in Section 4. In Section 5 we define the annotation language in which the properties are written. We discuss the automatic generation of sentries for programs with local and global properties in Section 6. Section 7 presents some experimental results, and the conclusions appear in Section 8.

2 Local Sentries

A property of a concurrent program P is a *local* property if it involves variables from only a single process p_i of P. A sentry that observes the correctness of such local properties is called a *local sentry.* In [5] we develop a series of local sentries to observe the local properties of processes in concurrent programs. These local properties consist of several progress and safety properties. Each progress property corresponds to the termination function of a loop in the process–in any execution of the process, the loop is guaranteed to have a bounded number of iterations. A safety

property states that if an execution of the process reaches a specified position, a corresponding property holds. During execution, the sentry reads snapshots of the process state and checks that the progress and safety properties are met.

The local progress properties for a process can be stated in terms of a set of termination functions–one for each terminating loop in the process. Consider the process in Figure 1 where the loop **do** $B \rightarrow S$ **od** has a termination function F. As such, F satisfies three conditions[8]:

```
process p0:
[   R0;
    do B ⟶ S od;
    R1
]
```

Figure 1: A simple process in a concurrent program.

its value is a natural number (well-foundedness); it is computed from a vector V of local variables of the process; and its value is monotonically decreased in each loop iteration to a lower bound (0, without loss of generality) until B becomes true.

Process p_0 can be extended to write V to the sentry at each iteration. The extended process p_0 and its sentry appear in Figure 2. To detect whether a given loop is executing correctly, vector V is written to the sentry at every iteration. The sentry reads successive values of V, computes the corresponding $F(V)$ and determines whether the values are monotonically decreasing. If the sentry detects a violation, it reports the violation.

A local safety property P of a process is a boolean expression, computed from a vector V of local variables of the process, that holds at a given point in the execution of the process. When the process reaches that point, it writes V to the sentry. The sentry reads vector V, and evaluates $P(V)$. If P does not hold, the sentry reports a violation to the process.

For the sentry's behavior to be correct, we need a synchronization mechanism that preserves linear storage and wait freedom, while guaranteeing mutual exclusion between the observed process and the sentry, and that every snapshot read by sentry is consistent. We can impose mutual exclusion between the process and sentry, by constructing within the shared memory a queue of buffers for V with associated synchronization and history variables. The number of buffers in the queue is at least two, for non-blocking mutual exclusion. In order to prevent the sentry from blocking the process, and to keep the length of the queue between the process and sentry bounded, we allow the process to overwrite the most recent unread snapshot with

```
shared var
    b : array [0..1] of vector V value;
    full : array [0..1] of boolean;
    nxt : 0..1
```

```
process p0:

var
    w : 0..1

[   R0;
    do B →
        b[w] := V;
        full[w] := true;
        if full[w +2 1] →
            skip
        [] ¬full[w +2 1] →
            w := w +2 1;
            nxt := w
        fi;
        S
    od;
    R1
]
```

```
process sentry0:

var
    r : 0..1;
    new, old : integer

*[  if full[r] ∧ r ≠ nxt →
        new := F(b[r]);
        if (0 ≤ new < old) →
            old := new
        [] ¬(0 ≤ new < old) →
            violation
        fi;
        full[r] := false;
        r := r +2 1
    [] ¬full[nxt] ∨ r = nxt →
        skip
    fi
 ]
```

Figure 2: The extended process and its sentry.

a new snapshot. In the case of a progress property, this has no effect–a violation of a loop's termination function can be detected even though the process overwrites information, because the termination function is monotonically decreasing. Thus, the comparison of an earlier value with any later value is sufficient to detect an violation. In the case of a safety property, a particular violation may be missed due to overwriting, but the sentry construction is based on the assumption that a serious fault will persist, and so can eventually be detected.

A discussion of local sentries for processes with more complex structures, such as non-terminating loops, nested loops, etc., can be found in [5].

3 Global Sentries

The global sentry observes the global state of a set of processes rather than the local state of one process. Its purpose is to observe the execution of multiple processes and check the validity of properties involving variables from different processes. This implies the need for a snapshot of the variables of several processes, from which the global state is computed. To fit the sentry paradigm, the snapshot must be obtained without blocking any of the executing processes. The sentry must also read consistent snapshots.

A correct global snapshot algorithm generates a consistent program state in which values from individual processes (could have) coexisted in time. To meet the sentry paradigm, a snapshot algorithm also needs to preserve our requirements of linear shared storage, wait-freedom, and mutual exclusion. The local sentry snapshot algorithm is no longer applicable, as it cannot obtain consistent snapshots of the global state without blocking the program. In the local sentry system, a single process writes its state to the sentry, and controls the synchronization between itself and the sentry. To guarantee non-blocking, consistent snapshots of a set of processes, only the sentry can synchronize communications. Allowing the processes to control the synchronization leads either to the blocking of some of the processes in the program, or to inconsistent snapshots.

Several existing algorithms can obtain snapshots of the state of concurrent programs. However, these algorithms violate either the linear storage requirement or the wait-freedom requirement. The Chandy/Lamport algorithm [4] blocks both the reader and the writers in generating the snapshot, thereby violating our wait-freedom requirement. The snapshot algorithms of Anderson[3] and Afek et al[1] are wait-free, but these algorithms require at least quadratic storage, violating our linear storage requirement. Furthermore, they also have very high overhead (quadratic to exponential).

The global sentry snapshot algorithm meets our requirements. To ensure the consistency of

a snapshot, each process of the concurrent program writes the value of an observed variable to the sentry immediately after modifying the variable.

To guarantee linear storage, there are two buffers in shared memory for each observed variable. Because the sentry controls the synchronization, only two buffers need ever be in use at any given time–one for the sentry to read from and one for the program to write to.

Wait-freedom is ensured because there is always a buffer available for any process of the program to write to. Mutual exclusion is ensured by forcing the sentry to wait until changes to the synchronization variable have propagated to all writing processes, before the sentry can read snapshots from those processes.

Within a single process, it is straightforward to identify a specific point in its execution at which a local property holds. It is far more difficult to identify the corresponding point in the execution of a concurrent program at which a global property holds. Thus, the properties observed by the global sentry are invariants that are maintained throughout the entire program execution rather than assertions associated with a particular point in the execution. During execution, the global sentry reevaluates the observed properties as new snapshots are read. Note that the global snapshot algorithm can be used to observe local properties of a single process. However, the global snapshot algorithm is more expensive, due to the necessity of writing snapshots to the sentry after every modification to an observed variable.

The global sentry continuously observes and evaluates global properties of a concurrent program. It executes in an endless loop. In each iteration of the loop, the sentry waits until some process writes new state information. The sentry then reads new variables from their buffers, preserving mutual exclusion with the processes, and enables the checking of any property involving those variables. Only properties involving new data will be tested. After the sentry has read the available state information from all processes, it checks the enabled properties. If any violations are detected, they are reported. The processes involved may choose to initiate some recovery action. Recovery is the responsibility of the processes, not of the global sentry.

We cannot halt all processes simultaneously to access their states and then resume execution. As a result, the snapshot is always of a state that could have occurred, but is not guaranteed to be a state that did occur in this execution. However, if a violation-state could have occurred, the underlying program is flawed.

In certain cases, it may be necessary to express a property or a set of properties by introducing local auxiliary variables to the processes. For example, the definition of mutual exclusion is that only one process may be in the critical section at any given time. This property is difficult to state in terms of existing program variables, especially if the mutual exclusion mechanism is not

```
process p_i:
*[
    /* write 1 into a_i when entering CS */
    in[i] := true;
    w[i] := nxt;
    a_i[w[i mod 2]] := 1;
    in[i] := false;

            /* critical section */

    /* write 0 into a_i when leaving CS */
    in[i] := true;
    w[i] := nxt;
    a_i[w[i mod 2]] := 0;
    in[i] := false;

            /* non-critical section*/
]
```

```
process mutex_sentry:
*[
    /* wait for any new data */
    or_wait( buffers[nxt mod 2]);
    /* processes write to b[nxt mod 2] */
    nxt := nxt + 1;
    /* sentry reads other buffer */
    r := (nxt - 1) mod 2;
    j := 0;
    /* for each process j */
    do (j < N) ⟶
        /* wait until safe to read from P_j */
        do (w[j] != nxt) ∧ in[j] ⟶ skip od;
        if (a_j[r] ≠ -1) ⟶ A[j] := a_j[r];
        [] (a_j[r] = -1) ⟶ skip
        fi;
        /* reset buffers to empty */
        a_j[r] := -1;
        j := j + 1
    od;
    /* at most one process in CS? */
    if ∑_{i=0}^{N-1} A[i] ≤ 1 ⟶ skip
    [] ∑_{i=0}^{N-1} A[i] > 1 ⟶ violation
    fi
]
```

Figure 3: Process p_i and the sentry for the N-process mutual-exclusion program.

accessible to the programmer, or if adding write statements to it affects its atomicity. Instead, it is simpler to add an auxiliary variable a_i to each process p_i, which is set to 1 when a process enters the critical section, and to 0 when the process leaves the critical section. (See Figure 3.) These auxiliary actions are the first and last actions taken within the critical section. The sentry in Figure 3 detects a violation when more than one process is in the critical section, as the sum of the auxiliary variables exceed 1.

4 Predicate Sentries

Both the local and global sentries described in the previous sections can be thought of as propositional sentries, because they only observe properties involving scalar variables. However, many interesting properties involve more complex data structures. In this section, we discuss the possibilities and tradeoffs inherent in observing large data structures, and how the sentry is extended to allow it.

The most straightforward solution to allow a sentry to evaluate a property involving a non-scalar data structure is to copy the structure to the sentry, as we copied scalars in the propositional sentries. However, given our desire that the sentry not impinge overly on the timing and performance of the program, this is not a viable option. For consistency's sake, we cannot allow the sentry to read the data structure without some synchronization. Nor can we block the program to allow the sentry to read the data structure, as that violates the wait-freedom requirement.

To allow reasonable performance and consistent evaluation of a predicate property, we introduce a notion of timestamping to the sentry system. Each observed data structure is made readable by the sentry, and is given an associated timestamp. This timestamp is visible to the sentry, but written only by the program. Immediately before the data structure is changed, the program increments the timestamp. Upon completion of the modification, the program sends the current timestamp to the sentry in a buffer, using whatever communication mechanism exists already. When the sentry evaluates a property involving a non-scalar data structure, it not only reads the data structure from the program, but also the structure's current timestamp and the buffered timestamp. If the current timestamp matches the buffered timestamp, the snapshot is consistent. If not, the sentry discards the snapshot, as its consistency is indeterminate, and seeks a new snapshot to instantiate the property.

Consider a program that measures the temperature of a substance over a period of time, and remembers the last 50 readings in array T. (See Figure 4.) A safety property on this program might be that every k probes the average temperature is always greater than 32 degrees, i.e.

$(n\_probes \textbf{ mod } k = 0) \wedge (\sum_{i=0}^{49} T[i])/50 > 32$. The annotation language (q.v.) allows arithmetic and logical quantification to be expressed in properties.

```
int last = 0, n_probes = 0;
...
while (1) {
    ...
    /*{ begin_modify( T) }*/
    T[last] = read_temp();
    /*{ end_modify( T) }*/
    n_probes++;
    last = (last + 1) % 50;    /* circular array of size 50 */
    /*{_p:  (n_probes % k) == 0 && (sum i:  0 <= i < 50 :  TRUE : T[i]) / 50 > 32 }*/
    ...
}
```

Figure 4: Annotated temperature program fragment.

5 The Annotation Language

In the previous sections, we have described how the snapshots required to evaluate the progress and safety properties of a program are communicated to the sentry. In this section, we discuss the annotation language in which the properties are expressed.

Properties are composed of the equality/inequality relations, the logic connectives (and, or, not), the arithmetic operators, the logical quantifiers ($\forall, \exists$) and the arithmetic quantifiers (Σ, Π). User-defined functions are not (yet) permitted, and the use of quantification is strictly limited to ensure the computability of the properties.

For propositional sentries, the annotation language is the propositional calculus combined with the integer-arithmetic operators and relations. Local properties are either termination functions, which annotate loops, or safety properties, which annotate code blocks. A termination function is written as a non-negative, integer-valued expression. A safety property is a boolean-valued expression. The table in Figure 5 contains a summary of the syntax for propositional properties.

Keyword	Meaning	Keyword	Meaning
`&&`	and	`<`	less than
`||`	or	`char`	character var
`!`	not	`float`	real var
`%`	modulo	`int`	integer var
`*`	multiply	`local`	local var
`/`	divide	`private`	private var
`+`	plus	`FALSE`	false
`-`	minus	`TRUE`	true
`==`	equal	`_f:`	header for term.
`!=`	not equal		function
`>=`	at least		
`<=`	at most	`_p:`	header for safety
`>`	greater than		property

Figure 5: Annotation language for propositional sentries.

In the annotations, termination functions are prefixed with the string "`_f:`", and safety properties are prefixed with the string "`_p:`". C syntax is used for relations and operations. The annotations are written within specially-formatted comments that are recognized by the sentry generator, but ignored by an ordinary C compiler. These comments appear in the places where the properties are to be evaluated.

Global properties are not associated with any given process, and are continually reevaluated during program execution, upon the receipt of new state information. Thus, instead of being written in a given process, all the properties of a set of processes are written in a separate file. Each property is labeled with a unique name. Every process has to declare which of variables in

```
/*{ send_rcv:  p_sent >= q_rcvd }*/
/*{ window_5:  5 + q_rcvd >= p_sent }*/
```

Figure 6: Constraints for reader/writer program.

the properties it writes. A variable is declared by type and by whether it is local to the process or is readable by other processes. In the first case, the variable is called a *local* variable, in the second, it is called a *private* variable. Figure 6 contains the predicate file for a reader/writer program with a window size of 5. Figure 7 contains code for the writer, in which the only annotation is the declaration of local variable `p_sent`. The reader's code would contain the declaration of local variable `q_rcvd`.

```
int *channel;     /* channel is shared between processes p & q */
int n_acks = 0;       /* number of acknowledgments */
int p_sent = 0;       /* number of data items sent by p */
/*{ int local p_sent; }*/      /* observe local var p_sent */

...

    if ((p_sent - n_acks < 5) && empty( channel)) {
        ns++;
        p_sent += 1;
        send( data[ns], channel);
    }
...
```

Figure 7: Writer code fragment.

Constraints involving non-scalar data types, especially arrays, require extending the annotation language to allow limited first-order expressions. Both logical ($\forall$, $\exists$) and arithmetic (Σ, Π) quantification are permitted, but only across a fully instantiated subrange. A quantified expression consists of a quantifier-free proposition (either logical or arithmetic) surrounded by any number of singly-indexed quantifiers with fully instantiated ranges. For example, the property that a 10×10 array A is sorted is represented as

```
(FORALL i: 0 <= i < 9: TRUE:
    (FORALL j: 0 <= j < 9: TRUE:
        (A[i,j] <= [i, j+1]) && (A[i,9] <= A[i+1, 0])))
```

even though the second conjunct does not involve the index j. A boolean function can be used to restrict the domain further.

In addition, a set of *modify* functions are added to the annotation language, so that the sentry can be informed about changes to a non-scalar data structure. The table in Figure 8 contains the extended syntax for predicate properties. The programmer places a `begin_modify( A)` annotation immediately before changing data structure `A`, and an `end_modify( A)` call upon completing the changes.

6 Implementation

We have constructed prototype local and global sentry generators that automatically construct, from annotated source programs, a sentry and a version of the source that can interact with

Keyword	Meaning
`EXISTS`	there exists
`FORALL`	for all
`PROD`	product
`SUM`	sum
`begin_modify`	begin to modify data struct
`end_modify`	end of modification

Figure 8: Extension to annotation language for predicates.

the sentry. The prototypes described below are the basis of a tool that allow programmers to check the progress and safety properties of sequential and concurrent programs during execution, supplementing static analysis.

6.1 Implementation of the Local Sentry Generator

The current implementation of the local sentry generator constructs a sentry and extended source program from an annotated C program. The local sentry generator uses a scanner/parser (based on lex and yacc) and templates to expand the program annotations into C code, and to create the main loop and validation functions for the sentry. The system consists of approximately 4000 lines of commented C code, with slightly more than half due to the grammar for the parser.

The current prototype accepts a single annotated C program P.c as input. Five files are generated from the input file–P.make, P.src.c, P.sentry.c, P.consts.h, and P.writer.h. The makefile P.make contains the compiler directives to create two executable programs, P.src and P.sentry, and the underlying communication structures. Communications and shared storage are implemented using the UNIX System V IPC shared memory package.

The file P.src.c contains the original source process, with the annotations replaced by calls to write macros. Each annotation in the source file generates a buffer queue, and the corresponding write macro that selects the correct buffer and writes the corresponding variable vector. For the most part, the C code in the input file P.c is reproduced in P.src.c without interpretation. However, in order to handle correctly a more complex process than we discussed in Section 2, such as one with nested loops, it is necessary to keep track of the nesting structure of the process. A macro call to increment the session number is added at the end of each level of nesting. At the beginning of the program, a call to an initialization macro sets up the communications structures. Macros are used both for legibility and for efficiency.

The file P.sentry.c contains the code for the sentry. The sentry generator uses the annotations

in P.c to create the code to read the variables from the buffers and to evaluate the appropriate safety or progress property. The main loop of the sentry does not change, except for a few constants based on the input program, and thus can be read from a template. These source-dependent constants–the maximum depth of the loops, the number of termination functions and safety properties, and the maximum number of variables–are stored in the file P.consts.h

To check the execution of the sequential program, the two executables P.src and P.sentry are run in parallel. As the sentry initializes the shared memory, it must be started first. P.src writes its variables to the sentry as it executes. The sentry reads the variables and computes the appropriate function. If a violation is detected, the sentry sends a signal to the program, which may initiate some user-defined recovery action based on the type of fault.

6.2 Implementation of the Global Sentry Generator

The global sentry generator consists of a scanner and parser (lex and yacc based), a code generator, and a template from which the sentry is constructed. The generator contains approximately 5000 lines of commented C code. Again, the parser grammar account for slightly more than half the code.

The global sentry generator takes as input a file of properties and the annotated C files that make up the concurrent program. A debug flag allows the user to follow the parsing and code generation in detail. The properties file contains the global (safety) properties that apply to (a subset of) the processes in the program. Because the properties are associated with a set of processes, rather than an individual process, it is more straightforward to keep them in a separate file.

Each process is annotated with a declaration of the observed variables that it writes. The sentry generator automatically recognizes changes to scalar variables, as long as no aliasing occurs, and inserts the necessary write macros calls as needed. For non-scalar variables, the programmer adds "modify" annotations, to inform the sentry that the structure is about to be changed.

Every property is expanded into a corresponding validation function in the sentry. The sentry generator uses the variable declarations and modify calls to complete the validation function.

Ideally, the programmer should not have to change her source code at all to use a sentry. However, for the sentry to observe data structures shared among several processes, the programmer has to allow the sentry to determine where in shared memory those data structures are to be located. Both scalar and non-scalar shared structures are affected. Shared memory

not observed by the sentry can be allocated independently of the sentry's memory.

Given a properties file X.props and C programs X_1.c...X_n.c as input, the current global sentry generator produces augmented source files X_1.src.c...X_n.src.c, write macros for each input file X_1.macros.h...X_n.macros.h, and a sentry file X.sentry.c. In this version, the programmer will have to modify the existing makefile to incorporate the sentry and its accompanying files. The required `#include` statements are automatically inserted into the modified source programs.

To observe the execution of the concurrent program described in X_1.c...X_n.c, the sentry is run in parallel with the the new executables. The sentry is responsible for allocating and initializing shared memory, so it must be started first. The processes $X_1 \ldots X_n$ write their state information to the sentry as it changes. The sentry reads the state as it becomes available, and evaluates the properties for which new information was obtained. Errors can be reported by signal to the processes involved in a violated property, so that they may take appropriate recovery actions.

6.3 Implementation of the Predicate Sentry Generator

In Section 4, we introduced the notion of a "predicate sentry," i.e. a sentry to observe properties about non-scalar data structures. In order to make the data structure visible to the sentry, the sentry generator automatically moves the structure from the program's memory to the sentry system's shared memory. The shared memory allocation is done at compile time, thus dynamically allocated data structures are beyond the scope of the sentry generator. As in the propositional sentries, shared memory is allocated for the synchronization variables and the buffers for propositional variables. For each structure to be observed, the generator allocates a timestamp and a set of buffers for that timestamp.

When parsing the file in which a data structure `A` is modified–each observed structure having exactly one writer–the sentry generator replaces each `begin_modify( A)` annotation with an increment of the timestamp of `A`, and each `end_modify( A)` with the writing of the timestamp to its buffer. The sentry, upon evaluating a property involving structure `A`, will check the consistency of the result by comparing the buffered timestamp with the actual timestamp. If the two do not match, the result is discarded.

```
/*{ sortA: (forall ii : 1 <= ii < 100: 1: A[ii] >= A[ii - 1]) }*/
```

Figure 9: Constraint sortA

```
/* compute the quantified sub-expression */
lq0 = TRUE;
for (ii = 1; ii < 100 && lq0; ii++) {
  if (1) {
    lq0 = lq0 && (A[ii] >= A[ii - 1]);
  }
}
/* now add that to the rest of the property */
valid = (lq0) || (*_tsA != tsA);
```

Figure 10: The validation function computing sortA.

Figure 9 contains the property that an array `A` is sorted. The validation function corresponding to property `sortA`, as generated by the predicate sentry generator, appears in Figure 10. The quantified sub-expression–in this case, the sortedness of array `A`–is computed first, and its value is stored in variable `lq0`. The final subexpression `(*_tsA != tsA)` determines whether array `A` has been changed since the rest of the expression was sent, and thus whether or not the the snapshot is consistent.

Predicate sentry generation has been incorporated into both the local and global sentry generators.

7 Experimental Results

The design of the sentry is based on three assumptions:

1. The sentry has low overhead.
2. The information loss encurred by the overwriting of snapshot information is minimal.
3. Sufficient predicates involving non-scalar structures are evaluated, rather than discarded due to inconsistency, to avoid starvation of the sentry.

Below we empirically test these assumptions. In the following experiments, we will be evaluating variants of two programs. The first is an n-process mutual exclusion program, implementing the algorithm described in Section 3 in which n processes vie for access to a critical section. The second program is a two-process parallel exchange sort that combines two sorted arrays of length N into a single sorted array of length $2 * N$. The execution environment is a multiprocessor Sparc-10, to allow truly parallel execution of the program and sentry.

7.1 Sentry overhead

In this experiment, we examine the overhead introduced by the sentry to a set of processes. This overhead arises primarily from the additional writing done by each process to communicate its variables to the sentry. There are also minor initialization costs. Every write to the sentry involves four assignments, as in Figure 3. The program tested is the parallel exchange sort, in which an integer variable and an array in each process are observed by the sentry. Each exchange triggers three writes to the sentry per process, and with the input fixed ($A[i] = 3 * i, B[i] = 2 * i$), for an array of size N, there are $2N/5$ exchanges per process.

array size	#snapshots per exchange	%overhead process P	process Q	average
100	4	13.52	8.59	11.1
	6	9.48	12.01	10.74
	10	16.63	17.08	16.85
	15	19.52	22.20	20.86
1000	3	1.12	8.99	5.06
	6	6.09	5.72	5.91
	10	10.14	9.69	9.92
	15	10.89	9.62	10.26
10000	3	6.03	2.16	4.09
	6	4.78	4.34	4.56
	10	7.48	2.55	5.02
	15	4.63	4.63	4.63

Figure 11: Overhead of introducing a sentry to the parallel exchange sort program.

The table in Figure 11 shows the overhead incurred by using the sentry with the parallel exchange sort program. Most of the computation in a single exchange involves a linear shift of the array contents, thus the execution time is proportional to the length of the array. We tested this program for arrays of length 100, 1000, and 10,000. For the array of length 100, the array was re-initialized and the sort was executed ten times, to allow for sufficient computation to dominate system costs. For the larger arrays, the exchange sort was executed once. We then forced additional writes to the sentry, by adding a parameterized loop to the source program to assign the observed integer variable to itself. This allowed us to measure the overhead caused by these extra writes, without changing the meaning of the program.

Note that due to the implementation of the sentry as a heavyweight process, a certain amount of the overhead results from context switching. A future thread-based implementation

is planned.

7.2 Information loss

Because the program being observed continually overwrites its snapshots, it is possible that the sentry may miss all occurrences of a recurring violation. The program we tested was the n-process mutual exclusion program, where each process obtains and releases a semaphore 1000 times during the course of its execution. We tested for information loss by deliberately introducing a violation in which a single process releases the semaphore before it has completed its critical section. In the case of two processes, the violation was never missed. The sentry was apparently able to keep up with the processes, reading at least 3800 of a possible 4000 snapshots. In the case of three processes, the violation was missed in 2% of the runs, and in the eight process case, the violation was missed in 4% of the runs. For a violation which occurred twice, the success rate for two to eight processes was 100%.

7.3 Starvation of the predicate sentry

As a result of allowing logical properties involving large data structures, it is possible that the predicate sentry will read inconsistent snapshots that cannot be evaluated. The sentry may starve because it can never obtain and evaluate a consistent snapshot. We tested for inconsistent snapshots by observing the exchange sort program with the property that the subarrays are sorted. For a sorted array of length N, the property $(\forall i : 1 \leq i < N : A[i] \geq A[i-1])$ requires that the entire array A be read (the worst possible case) before the property can be evaluated. Our results (Figure 12) show, as expected, that for very large structures, the sentry discards

array length	%discarded
100	34.06
1000	46.48
10000	95.72

Figure 12: Percentage of snapshots discarded due to inconsistency.

most snapshots. However, even in that case, the sentry never missed a recurring violation.

8 Concluding Remarks

In this paper we have presented a system to check the safety and progress properties of concurrent programs in execution. We have developed sentry algorithms that observe local and global properties during program execution, while ensuring finite storage and mutual exclusion, and without introducing new synchronization constraints on the program. We have also extended the sentry to check logical properties involving large data structures without prohibitive overhead. We have discussed the annotation language with which the programmer defines the properties to be checked, and have described a prototype to generate the appropriate sentry and program modifications automatically from annotated C source programs. Finally, we have presented some encouraging experimental results.

There are a number of approaches that can be taken to increase the applicability and utility of the sentry system, based on the existing sentry algorithms. We will extend the class of properties that can be observed to include temporal properties. Time-critical programs are a natural target for observation during execution, and a sentry, which does not affect the program's synchronization and has limited overhead, seems to be a practical monitor to explore. We will add new languages to the sentry generator, so that it can accept input and construct sentries in other imperative programming languages, for increased utility. We also plan to develop new algorithms to allow the sentry methodology to be applied to distributed and message-passing platforms, in which shared memory is either extremely limited or not available.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merrit, and N. Shavit. Atomic snapshots of shared memory. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–13, August 1990.

[2] I. Anderson and R. Kerr. Recovery blocks in action: a system supporting high reliability. In S. K. Shrivastava, editor, *Reliable Computer Systems: collected papers of the Newcastle Reliability Project*, pages 80–101. Springer-Verlag, 1985.

[3] J. H. Anderson. Composite registers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 15–29, August 1990.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, February 1985.

[5] S. E. Chodrow and M. G. Gouda. The sentry system. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992.

[6] S. E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 74–83, 1991.

[7] R. M. Cohen. *Proving Gypsy Programs.* PhD thesis, The University of Texas at Austin, 1986.

[8] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[9] F. Jahanian, R. Rajkumar, S. E. Chodrow, and S. Raju. Software support for run-time monitoring of real-time systems. Technical report, IBM T. J. Watson Research Center, January 1992.

[10] K. H. Kim and J. C. Yoon. Approaches to implementation of a repairable distributed recovery block scheme. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 50–55, June 1988.

[11] C.-C. Lin and R. J. LeBlanc. Event-based debugging of object/action programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):23–34, January 1989.

[12] D. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Springer-Verlag, 1990.

[13] S. Raju, R. Rajkumar, and F. Jahanian. Monitoring timing constraints in distributed real-time systems. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 57–67, 1992.

[14] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, 1989. STAN-CS-89-1282.

[15] L. M. Smith. *Compiling from the Gypsy verification environment.* PhD thesis, The University of Texas at Austin, 1980.

[16] S. Yemini and D. M. Berry. A modular verifiable exception-handling mechanism. *ACM Transactions on Programming Languages and Software Systems*, 7(2):214–243, April 1985.