Certifying the Concurrent State Table Implementation in a Surgical Robotic System

Yanni Kouskoulas[1] Ming Fu[2,3] Zhong Shao[4] Peter Kazanzides[5]

Abstract

This paper describes the application of formal methods to the reduction of defects in software used to control a surgical robot. We use a recently developed program logic called History for Local Rely/Guarantee (HLRG) to verify that the software implementation behaves according to the intended design. HLRG enables precise description of a system's functionality, its desired behavior, and facilitates rigorous, mathematical proofs about properties of the system via sound inference rules. During this process, we found a subtle bug in the system, corrected it, and were able to formally prove that within the component we were analyzing, with respect to two critical properties, the system had no flaws in its design or implementation.

1 Introduction

The overall goal of this research is to help develop a practical technique to improve the reliability of safety-critical software systems that control robotic machinery. In particular, we apply a recently developed program logic (HLRG) to help reduce defects in the Surgical Assistant Workstation (SAW), a software framework designed to run surgical robots. This work will help ensure that patient safety is not adversely affected by flaws or bugs in the SAW software.

The main contributions in this paper are twofold: first, we begin the process of transitioning the HLRG program logic to practical use by applying it to a real system and identifying areas that need further development; second, our work increases the reliability of key components of the SAW framework, illustrating how HLRG can be used on a real system.

[1]Johns Hopkins University, Applied Physics Laboratory
[2]University of Science and Technology of China, School of Computer Science and Technology
[3]Software Security Laboratory, Suzhou Institute for Advanced Study, USTC
[4]Yale University, Dept. of Computer Science
[5]Johns Hopkins University, Dept. of Computer Science

The target of our analysis is a software framework called the Surgical Assistant Workstation (SAW), created by the Engineering Research Center for Computer Integrated Surgical Systems and Technology (CISST ERC) at Johns Hopkins University. It is currently used for research with the da Vinci surgical system (shown in Fig. 1), the JHU microsurgery workstation, and other surgical robotic systems [5].

Because SAW is a software framework, there are many configurations and many applications for which it can be used. For example: it could create a heads-up display, dynamically superimposing pre-operative CT or MRI images onto the surgical field to highlight and orient the surgeon to anatomical features that are not otherwise visible; it could enforce no-cut volumes, preventing the surgeon from cutting into tissue or organs that he identified volumetrically on pre-operative images; or it could actually make cuts based on the surgeon's preoperative plan.

Figure 1: The da Vinci surgical workstation. Photo courtesy of Intuitive Surgical, Inc.

The SAW has stringent timing constraints that are based on the physics of the mechanical components being controlled and the nature of the task at hand, i.e. surgery. We expect that most software that controls robotic machinery has similar requirements. To ensure that the software response is rapid, the SAW features a high level of concurrency and lock-free algorithms.

Concurrent, lock-free algorithms are difficult to develop correctly, and difficult to debug, because of the unpredictable timing of interactions between different threads. Although conventional unit testing may be sufficient to find defects and flaws in some sequential programs, it is inadequate for ensuring safe and correct operation for the concurrent algorithms that we find in the SAW.

During this process, we did not evaluate the effectiveness of the software design, or attempt to correlate the use of the system to patient surgical outcomes, which would be necessary steps to take during the production and deployment of such a system.

2 The SAW State Table

We have chosen to analyze one of the core algorithms that mediate concurrent interaction between threads, which allows threads to read feedback data, such as the position of the robot, while simultaneously allowing the same information to be continually updated. This information is held in a data structure called the state vector, and is constantly changing as the robot moves to reflect its position in space. Depending on the SAW's configuration, the state vector could include position and velocity for each of the joints of the robotic arms, the state of end effectors, and the state of different imaging sensors, among other things.

The difficulty in storing the state vector is that the SAW framework supports multiple, interacting, concurrent processes, and as time progresses, the state vector's value changes rapidly. It must be both updated and read by different processes, simultaneously, in an accurate and reliable manner. The SAW framework has a data structure, called the state table, that provides an interface for the rest of the system to read and write the state vector. The state table is essentially a circular buffer that maintains a time history of the state vector.

2.1 State Table Requirements

We wish to guarantee *Data Integrity:*

> For each successful read of the state vector, no writer altered or was in the process of altering data during the read; and successful reads are distinguishable from unsuccessful reads.

We assume there is exactly one process that updates the state vector, and many concurrent readers, each of which can start at any time, and progress at any speed.

When we prove this property, we will have a strong guarantee that the code correctly implements one aspect of the algorithm. However, the developer must understand what this guarantee does not provide: A correctly completed proof of this property does not give a complete set of desirable properties for this algorithm, does not make guarantees about the operation of the design satisfying the larger purpose of the system, and depends on axioms and the function's correct use in the environment of the SAW. For example, this guarantee does not address reader starvation, i.e. the possibility of the reader never being able to successfully complete a read. It cannot ensure that the algorithm is appropriate for the system, e.g. that the other system components necessarily need to know the state information that it provides. It also cannot ensure that the state table is used properly in the context of the overall system, e.g. that there is only one writer thread in the system as we are assuming. These additional properties must be shown separately.

What would happen to the system if this read property did not hold? A thread might read corrupted state information, and act on it, causing serious malfunction. If, for example, the SAW were used to enforce restricted volumes of movement for a surgical robot, the enforcement might be applied incorrectly, allowing the robot to injure the patient.

2.2 State Table Algorithm Design

To do its job, the state table maintains space for storage of H copies of the state vector, each with an identifying version number attached. Figure 2 shows a single copy of the state vector, along with its version number.

Figure 2: A single copy of the state vector in memory, along with its version number. The top storage location is for the version number, while the lower array is for the state vector itself. The array is shown with two elements, but in general its length is configurable. Writing to and reading from the state vector is not an atomic operation, but writing to and reading from the version number is.

Shared memory used by this algorithm is organized as a circular buffer, as shown in Fig. 3, where each element in the circular buffer is a copy of the state vector at some point in time, and its version number. We call each element of the buffer a "slot." It also maintains two slot indices.

The writer updates the slots sequentially using the following algorithm: it writes a fresh state vector into the slot referred to by the write-index; it advances the write-index clockwise; it updates the version number of the slot newly pointed to by the write-index; and it advances the read-index to point to the slot that was previously referred to by

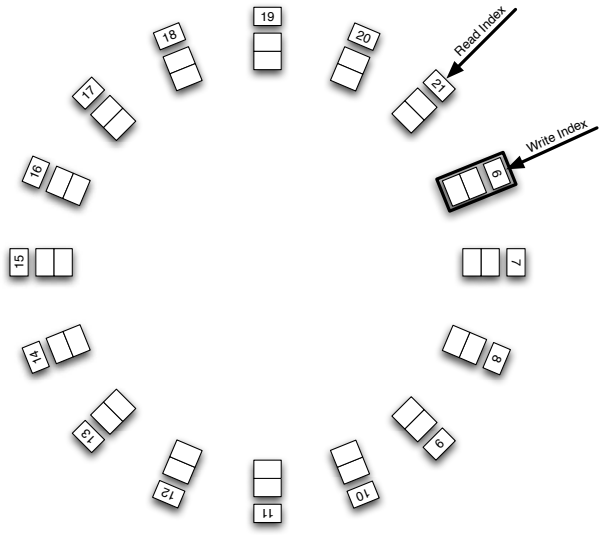
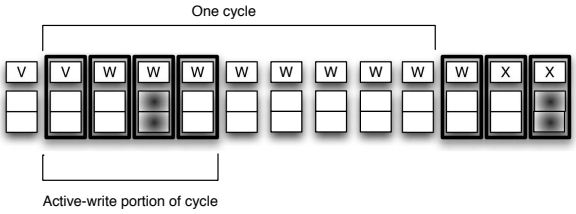


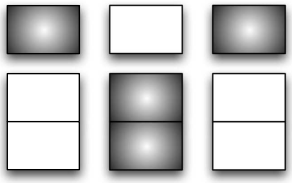
Figure 3: The circular buffer used to store copies of the state vector in memory. The read- and write-indices each independently refer to a single slot in the circular buffer. The read-index is indicated by a dashed outline surrounding the slot to which it refers, and the write-index is indicated by a solid outline around the slot to which it refers.

the write index, that has a freshly updated version of the state vector. This process is repeated, with the write-index progressing around the circle in a clockwise fashion. The figure below shows the sequence of events in a write of the state vector. It displays the condition of a single slot repeatedly over time, so different positions on the horizontal axis represent different steps. A square with its center shaded indicates inconsistent data in memory due to a non-atomic write. The bold outline indicates that the write-index refers to this slot during this time step; the read-index is not shown on this time progression.

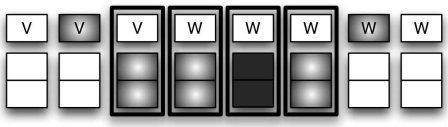


The data contained in a slot changes for each cycle of the write-index around the buffer, so a single read needs to be contained within one cycle to be uncorrupted. Tracing through the algorithm, we can convince ourselves that the state vector data in a slot may not change without that slot first having had its version number updated. Consequently, the reader strategy compares version numbers before and after a read to determine whether the read is valid: if the version numbers match, the read is assumed to be uncorrupted; if not, an error is returned.

The figure below illustrates this read strategy by showing the same slot repeatedly, with the horizontal axis representing the progression of time. A square with its edges shaded indicates that a particular data element is being read during this point in time.



If the version is not the same at the beginning and at the end of the read, we can conclude that the read spanned a cycle boundary, and the data read may be inconsistent. An example of a corrupted read is shown by the figure below:



The step where the state vector is completely shaded represents simultaneous reading and writing of the data. Note that this read took quite a bit longer, and so the intermediate states where the data was being read are more numerous. Because the reader began before the write point, and continued during and after it, the read was corrupted. In this case, the read strategy correctly identifies this read as corrupted based on the altered version number after read completion.

3 Related Work

The literature related to building higher-assurance software for control of robotic systems approaches the problem from a wide variety of perspectives. However, we are not aware of any work that could directly address verification of the concurrent algorithms found within the SAW.

For example, there is work on adding assurance to the software development process [12], and on verification of the behavior of sequential programs for robots [9], which while related, does not address concurrency.

There are testing approaches that are designed to apply to concurrent software that are also inapplicable to our problem. For example, [3] is an approach based on sequences of synchronization primitives, but the state-table algorithm we wish to analyze has none. Another example is [8], which offers an approach that explicitly "is not designed to test for

synchronization errors that lead to rare probabilistic faults," while we remain interested in preventing such faults.

The closest paper in the literature to the current work that we are aware of is [11], which applies a model-checking approach to verify properties in a concurrent system. However, this work verifies the system at a higher level of abstraction, and could not produce the specific, local guarantees relating implementation with design that HLRG does. This technique is complementary to the current work, and could help produce higher-level inferences by reasoning about specific local guarantees in the context of the larger system. It also offers an interesting approach to formalizing and reasoning about analog characteristics of the robotic system.

4 Formal Verification

To guarantee that there are no defects with respect to the properties that we describe, we applied a recently developed program logic called History for Local Rely/Guarantee (HLRG) [2]. We chose HLRG because it enabled us to precisely describe the software, and then apply sound inference rules to reason, in a mathematically rigorous manner, about algorithms that include optimistic concurrency.

HLRG builds upon LRG [1] which combines separation logic [10, 4] to enable local reasoning, with rely-guarantee reasoning to make guarantees about multiple processes accessing the same data structures concurrently, with trace-based assertions and temporal operators to describe time-based properties of the algorithm.

Within HLRG, logical statements, or assertions, about the system, are not confined to the current state of the system, but refer to a vector of system states, where each element represents the system state at a particular step in its evolution. We call this vector of states a trace, and it is what allows our assertions to refer to the history of the system (the origin of the "H" in HLRG). To refer to history, HLRG introduces a number of temporal operators. Intuitively, if a and b are predicates on traces: $a \rhd b$ means that a held at some point in the past, and b has held at every step since; $a \blacktriangleright b$ means that a held at some point in the past, and b held at some point subsequently; $a \ltimes b$ means that a held one step ago, and b holds now; $\boxminus a$ means that a holds at every step in the trace; and $\diamondminus\, a$ means that a held at some point in the past.

Following convention, we use $*$ to represent the separating conjunction, an operator that allows us to reason about local data structures without specifying all of memory. Thus, $a * b$ means that both a and b hold for the current trace, but that the subtrace satisfying a is disjoint from the subtrace satisfying b. (A subtrace is a trace where the state for each time step has a subset of the state contained in the original trace. Disjoint subtraces have disjoint state for each pair of corresponding time steps.) We use $\wedge$ to represent the regular conjunction, and $\vee$ to represent disjunction, as usual. Following [7] and notational convention, we treat heap variables as resources using binding operators that represent assertions about state: $\texttt{writeindex} \mapsto 5$ is an assertion about the heap, namely that the heap in the most recent state of the trace consists of one cell at memory location `writeindex` pointing to the value 5. We can represent multi-celled heaps by using the separating conjunction, so a two-celled heap could be written as: $(\texttt{writeindex} \mapsto 5) * (\texttt{readindex} \mapsto 4)$. We use $\rightsquigarrow$ to be an imprecise binding assertion, i.e. $\texttt{readindex} \rightsquigarrow 5$ means that the heap has at least the memory cell at address `readindex` which containing value 5, and may have more state as well.

4.1 Modeling the System

First, we created a model of the program by computing backwards program slices at all points where shared state is accessed, using the shared state as our slicing criteria. The union of these program slices is then converted to a simple C-like language that makes all complicated semantics explicit. The strength of our guarantees depends on the fidelity of our model, shown in Fig. 4.

Next, we create an invariant I that describes the shape of the heap throughout the evolution of the program (i.e. what memory is mapped), without specifying the values contained in those locations. Shared state in our case, consists of the the read- and write- indices, the state vector copies and their version numbers.

$$
\begin{aligned}
\mathsf{VersArray} &\stackrel{\text{def}}{=} \circledast_{i\in[0,\ldots,H-1]} \texttt{version}{+}i \mapsto \_ \\
\mathsf{Vector}(i)(j) &\stackrel{\text{def}}{=} \texttt{Vec}{+}i \times N + j \mapsto \_ \\
\mathsf{Vector} &\stackrel{\text{def}}{=} \circledast_{j\in[0,\ldots,N-1]} \\
&\quad (\circledast_{i\in[0,\ldots,H-1]} \mathsf{Vector}(i)(j)) \\
I &\stackrel{\text{def}}{=} \exists X.Y.\mathsf{VersArray} * \mathsf{Vector} * \\
&\quad \texttt{readindex} \mapsto X * \\
&\quad \texttt{writeindex} \mapsto Y
\end{aligned}
$$

Underscores are don't cares, while $\circledast$ is to $*$, as Σ is to $+$.

The next step is to write all of the atomic actions that are taken on shared state as predicates.

Lines 6 and 7 in Fig. 4 are one atomic action from the perspective of shared state, that updates the write-index, `writeindex`, to point to the next element in the buffer. We write a predicate that is satisfied with a trace that has just taken this step as follows:

$$
\begin{aligned}
&\mathsf{UpdWrite} \stackrel{\text{def}}{=} \mathsf{Id} * ((\mathsf{UpdData} \rhd \mathsf{Id}) \wedge \exists X, X'. \\
&\texttt{writeindex} \mapsto X \ltimes \texttt{writeindex} \mapsto X' \wedge \\
&X' = (X+1) \mathsf{mod}\ H)
\end{aligned}
$$

```
00 global Vector[N][H], readindex, writeindex, version[H];

01 void Write(int data[N]){
02   local old, i, tmp, wr;
03   old = writeindex;
04   for (i=0;i<N;i++)
05     Vector[i][old] = data[i]
06   wr = (old + 1) mod H;
07   writeindex = wr;
08   tmp = version[old] + 1;
09   version[wr] = tmp;
10   readindex = old;
11 }

12 int Read(int data[N]){
13   local rd, curTic1,
14         curTic2, i;
15   rd = readindex;
16   curTic1 = version[rd];
17   for (i=0;i<N;i++)
18     data[i] = Vector[i][rd];
19   curTic2 = version[rd];
20   if (curTic1 == curTic2)
21     return 1;
22   else return 0;
23 }
```

Figure 4: Model of Code

This step must follow the UpdData step, with some number of intervening steps, all of which must be steps that do not change shared state, Id, so we use the temporal operator $\triangleright$ to enforce this sequencing. This predicate, along with all of the others that describe atomic steps, have an Id connected to them with a separating conjunction, which ensures that any variables in the domain not explicitly referred to in the predicate remain unchanged.

Lines 8 and 9 also constitute an atomic block, updating the version, or `version` element, associated with the slot referred to by the write-index. This step must follow an UpdWrite step.

$$\begin{aligned}&\mathsf{UpdVer} \stackrel{\text{def}}{=} \mathsf{Id} * ((\mathsf{UpdWrite} \triangleright \mathsf{Id}) \wedge \exists X, X', V, V'.\\ &\texttt{writeindex} \mapsto X * \texttt{version}{+}X' \mapsto V' \ltimes\\ &(\texttt{writeindex} \mapsto X * \texttt{version}{+}X \mapsto V' {+} 1 *\\ &\texttt{version}{+}X' \mapsto V') \wedge X = (X'+1) \mathsf{mod}\ H)\end{aligned}$$

Line 10 is an atomic action, updating the read-index, or `readindex`. It must follow the UpdVer step, with some intervening number of identity transitions.

$$\begin{aligned}&\mathsf{UpdRead} \stackrel{\text{def}}{=} \mathsf{Id} * ((\mathsf{UpdVer} \triangleright \mathsf{Id}) \wedge \exists X, Y.\\ &\texttt{writeindex} \mapsto Y \ltimes \texttt{readindex} \mapsto X *\\ &\texttt{writeindex} \mapsto Y) \wedge Y = (X+1) \mathsf{mod}\ H)\end{aligned}$$

The following predicate describes an atomic portion of the update in lines 4 and 5. This predicate can occur any number of times in sequence, but the sequence must always follow the UpdRead step, again with some number of transitions that do not change shared state.

$$\begin{aligned}&\mathsf{UpdData} \stackrel{\text{def}}{=} ((\mathsf{UpdData} \vee \mathsf{UpdRead}) \triangleright \mathsf{Id}) \wedge\\ &\textstyle\bigvee_{j \in [0,\ldots,N-1]} \exists X.\\ &(\mathsf{Vector}(X)(j) * \texttt{writeindex} \mapsto X \ltimes\\ &\mathsf{Vector}(X)(j) * \texttt{writeindex} \mapsto X) * \mathsf{Id}\end{aligned}$$

Finally, we used the description of the program's atomic steps to create rely and guarantee predicates describing the operation of the `Write` program in a fairly straightforward way.

$$\begin{aligned}G &\stackrel{\text{def}}{=} (\mathsf{Id} \vee \mathsf{UpdData} \vee \mathsf{UpdWrite} \vee \mathsf{UpdVer} \vee \\ &\quad\ \mathsf{UpdRead}) \wedge (I \ltimes I)\\ R &\stackrel{\text{def}}{=} \mathsf{Id} \wedge (I \ltimes I)\\ \mathcal{M} &\stackrel{\text{def}}{=} \boxminus(R \vee G)\end{aligned}$$

The guarantee predicate G, is a guarantee about the behavior of the thread executing the `Write` function: it tells us how a step taken by that thread affects shared state. R assures us that the rest of the concurrent processes (namely the multitude of possible readers executing `Read`) have no effect on the state at all. $\mathcal{M}$ describes the behavior of the system as a whole: any step in the system will either execute a step in the `Write` function or a step in the `Read` function, and the state of the system changed (or not) accordingly. Furthermore, $(I \ltimes I)$ tells us that the invariant that describes the domain of the program doesn't change from step to step. Through this process, we have described the effect of `Write` on shared state, and its interaction with other concurrent processes.

4.2 Proving Data Integrity

To prove data integrity, we began with a predicate of true as a precondition to `Read`, and used the sound inference rules associated with HLRG to propagate the precondition through the function. Via this process, we sought to guarantee that when the `if` statement takes the `return 1;` branch, the postcondition of the computation of the branching condition guarantees that $\mathsf{Vector}(X) \rightsquigarrow D$ held during

the time period that included copying of state vector elements, where X is the index of the slot we were reading.

This implies not just that `Read` read data that was constant during the copy, but that its contents could not have been altered by a writer during that period, because where updates cannot occur in the `Write` algorithm, the state-vector is considered uncorrupted. This is subtle but important: it guarantees that our read did not occur in the middle of a `Write` that had stalled, leaving the value constant but corrupted.

With such a guarantee, when the `Read` completes successfully, the value that is returned accurately reflects an uncorrupted version of what was stored in that slot by `Write`.

4.2.1 Proving the Stable Data Lemma

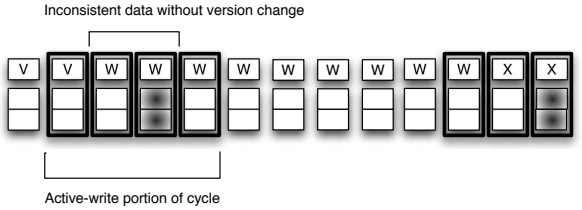
Going through this process is straightforward, once one proves the following, which we call the Stable Data Lemma:

$$((\texttt{version}{+}h \rightsquigarrow X \blacktriangleright \mathsf{Vector}(h) \rightsquigarrow D) \wedge (\mathsf{Vector}(h) \rightsquigarrow D' * \texttt{version}{+}h \rightsquigarrow X)) \Rightarrow (D = D')$$

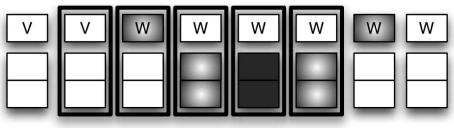
This is an invariant tied to our `Read` algorithm, that says: If, at some time in the past, we looked at the value of $\texttt{version}{+}h$, and then we looked at the state vector in slot h, $\mathsf{Vector}(h)$, and we look at $\texttt{version} + h$ in the present and its value matches what we saw the first time, then the value of $\mathsf{Vector}(h)$ in the present is also the same as what we read in the past. We need this lemma to prove that there is a continuous period of time when $\mathsf{Vector}(h) \rightsquigarrow D$ holds.

When we initially attempted to write down a proof of the stable data lemma by inducting over the steps in a trace, we found that it was not true of our system, and thus the read data integrity property was not true: readers could unknowingly read uncorrupted data. We had found a subtle bug, not by informally examining the system, or by testing it, but by carefully modeling it, writing a lemma, and attempting a formal proof.

The crux of the problem is that there is a very short period of time at the beginning of the "active-write" portion of the cycle, that occurs after the version number has changed but before the data update has been completed. During this time, the data may become inconsistent without the version number changing. If the read occurs during this time, the result may be inconsistent without us being able to detect it. The figure below shows the portion of the cycle that is the problem.



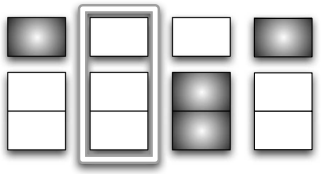
An example interleaving that exhibits this problem is as follows:



We cannot guarantee that this situation never happens, because the change of version happens separately from the update of index-writer. If these updates were one atomic operation, this would not be a problem. This algorithm was deliberately designed to be lock-free, however, so we would like to avoid this solution.

4.2.2 Proving an Improved Stable Data Lemma

We observe that in every situation like the one above, the problem occurs when the initial version check and the read occur within the active write portion of the cycle. If both are in the active portion of the cycle, then the state in between the initial version check and the read must also be in the active write portion of the cycle. In Fig. 5, we modify `Read` so that it checks the status of the write-index between the first version check and the read of the data. This strategy is illustrated below, with the check of the write-index status indicated by the double-line rectangle.

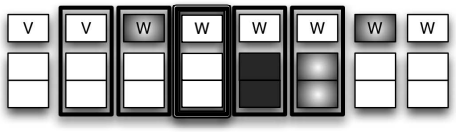


To guarantee that we solved this problem, we rewrote the statement of our lemma to reflect the changes we made:

$$((\texttt{version}{+}h \rightsquigarrow X \blacktriangleright \texttt{writeindex} \rightsquigarrow h' \blacktriangleright \mathsf{Vector}(h) \rightsquigarrow D) \wedge (\mathsf{Vector}(h) \rightsquigarrow D' * \texttt{version}{+}h \rightsquigarrow X) \wedge (h \neq h')) \Rightarrow (D = D')$$

This modification of the read algorithm highlights three cases: one where the write-index points to the current slot, one where it comes directly after the active-read portion of the cycle, and another when it comes directly before the active-read portion of the cycle. The intuition behind these cases is given below.

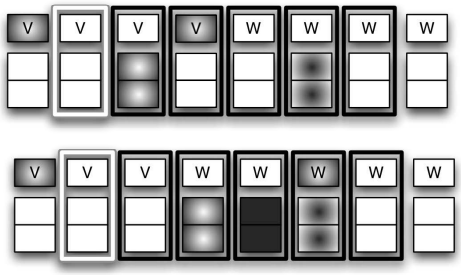
1. When the write-index indicates that this element is in the "active write" portion of the cycle between the version check, we assume the read is inconsistent. All of our bad traces exhibit this feature, and this is the particular case we would like to distinguish. The trace below is an example of this interleaving:



2. When the write-index check indicates that this element is not in the "active write" portion of the cycle, and the active write portion occurred before the write-index check, the data read within this cycle will be consistent unless the read spans multiple cycles, in which case it will be flagged as inconsistent by the version check. Example traces that exhibit this situation are shown below.

We can accept any such reads without fear of having read inconsistent data, although the version check will conservatively reject some uncorrupted reads, as in the second example.

3. When the write-index check indicates that this element is not in the "active write" portion of the cycle, and the active write portion occurs after the initial write-index check, the first version check will read the version associated with the previous cycle. If the data is changed during the read, it will be altered after the version has been changed, and because of the first version check happens during the previous cycle, these traces will be eliminated by comparing the versions before and after the read. If the read completes before the version is changed, then we have a guarantee that the data has not been changed, even if the write-index indicates we are in the active write portion of a new cycle by the end of the read. Traces that exhibit this situation are shown below:



With this modification, our proof of the improved stable data lemma succeeds. The proof is by induction over the steps of the trace; to prove $i \Rightarrow i + 1$, we proceed by case analysis of all possible atomic steps in the system.

We can now complete the proof of the read data integrity property, by propagating the precondition `true` forward through `Read`. Using the improved stable data lemma, the predicate divides into two possibilities, depending on the state of the version and write-index during different stages of the read. If conditions conform to the improved read strategy, the state vector is continuously constant during all stages of the read, and thus uncorrupted; otherwise, we cannot guarantee it is so. The completed proof of read data integrity and the necessary lemmas are presented in [6].

5 Conclusion

We have been able to provide a very strong guarantee about the correct functioning of a component in a library that is intended to control surgical robotic systems. During this process, we found and corrected a subtle bug associated with concurrency. Furthermore, we proved that with respect to this property, there are no more flaws or bugs in the implementation of that component.

5.1 Future Work

HLRG is a leap forward in capability, when compared with non trace-based program logics, making it simple to express and reason about the relationship between data structures at different times. However, there are practical problems applying HLRG widely, and hopefully these can be addressed as the technique matures. We identify five problems with this approach, and improvements that would be necessary to make this technique more practical.

First, transformation of the program from the original language (in this case C++) to HLRG is tedious and error prone as it must be done by hand.

Second, the proof representation does not lend itself to being machine checked, or maintained during further development of the software.

```
00 global Vector[N][H], readindex, writeindex, version[H];

01 void Write(int data[N]){
02   local old, i, tmp, wr;
03   old = writeindex;
04   for (i=0;i<N;i++)
05     Vector[i][old] = data[i]
06   wr = (old + 1) mod H;
07   writeindex = wr;
08   tmp = version[old] + 1;
09   version[wr] = tmp;
10   readindex = old;
11 }

12 int Read(int data[N]){
13   local rd, wr, curTic1,
14         curTic2, i;
15   rd = readindex;
16   curTic1 = version[rd];
17   wr = writeindex;
18   if (rd == wr)
19     return 0;
20   for (i=0;i<N;i++)
21     data[i] = Vector[i][rd];
22   curTic2 = version[rd];
23   if (curTic1 == curTic2)
24     return 1;
25   else return 0;
26 }
```

Figure 5: Corrected code

Third, the approach was applied after the software was developed. There should be a strategy that integrates the approach with the software development process, allowing a flawed design to be identified and corrected sooner.

Fourth, our guarantees of correct operation are local to a particular component of the SAW and link implementation details to its design. If we want to make some higher level guarantee (e.g. "operations within the SAW are thread-safe") we need some way to "knit" together these local guarantees and infer the broader conclusion.

Finally, HLRG in its current form may not be sufficiently expressive to describe properties that we need. We had some difficulty formalizing some of our lemmas (e.g. the Continuously Constant Version Lemma) and had to write them partially in English.

References

[1] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[2] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 388–402. Springer Berlin / Heidelberg, 2010.

[3] G.-H. Hwang, K.-C. Tai, and T.-L. Huang. Reachability testing: an approach to testing concurrent software. In *Software Engineering Conference, 1994. Proceedings., 1994 First Asia-Pacific*, pages 246 –255, dec 1994.

[4] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.

[5] P. Kazanzides, S. DiMaio, A. Deguet, B. Vagvolgyi, M. Balicki, C. Schneider, R. Kumar, A. Jog, B. Itkowitz, C. Hasser, and R. Taylor. The surgical assistant workstation (SAW) in minimally-invasive surgery and microsurgery. In *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal, Jun 2010.

[6] Y. Kouskoulas, F. Ming, Z. Shao, and P. Kazanzides. Certifying the concurrent state table implementation in a surgical robotic system (extended version). Technical report, Yale University, 2011. http://flint.cs.yale.edu/flint/publications/statevec-tr.pdf.

[7] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*, pages 137 –146, 2006.

[8] W. Pugh and N. Ayewah. Unit testing concurrent software. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 513–516, New York, NY, USA, 2007. ACM.

[9] M. Rahimi and X. Xiadong. A framework for software safety verification of industrial robot operations. *Computers and Industrial Engineering*, 20(2):279 – 287, 1991.

[10] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on*, page 55, 2002.

[11] Y. Sun, B. McMillin, X. Liu, and D. Cape. Verifying noninterference in a cyber-physical system – the advanced electric power grid. In *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 363 –369, oct. 2007.

[12] P. Varley. Techniques for development of safety-related software for surgical robots. *Information Technology in Biomedicine, IEEE Transactions on*, 3(4):261 –267, dec. 1999.