Certifying the Concurrent State Table Implementation in a Surgical Robotic System (Extended Version)

Yanni Kouskoulas
Johns Hopkins University
Applied Physics Laboratory

Ming Fu
University of Science and Technology of China
School of Computer Science and Technology

Zhong Shao
Yale University
Dept. of Computer Science

Peter Kazanzides
Johns Hopkins University
Dept. of Computer Science

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.

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:

1. *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.

2. *Data Freshness:* Each read accesses the most recent version of the state vector available at the start of the read.

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.

There are other useful guarantees that we did not address in this research, such as guarantees about reader starvation, i.e. the possibility of the reader never being able to successfully complete a read, or guarantees about there actually being only one writer thread active in the system. These properties are taken as axiomatic for this research, and must be proven separately in the context of the larger system.

What would happen to the system if these properties 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.

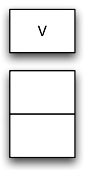


Figure 2: A single copy of the state vector in memory, along with its version number. The top storage location is for the version number, while the lower array is for the state vector itself. The array is shown with two elements, but in general its length is configurable. Writing to and reading from 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 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

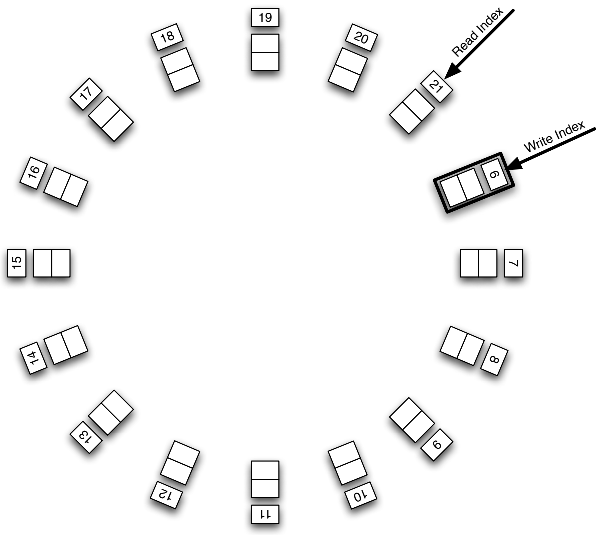
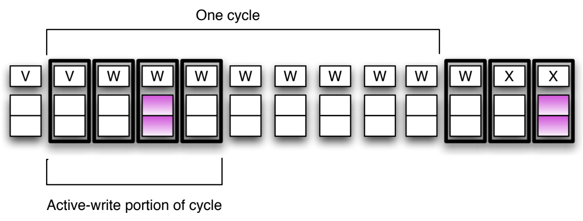


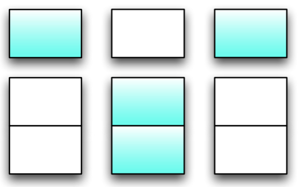
Figure 3: The circular buffer used to store copies of the state vector in memory. The read- and write-indices each independently refer to a single slot in the circular buffer. The read-index is indicated by a dashed outline surrounding the slot to which it refers, and the write-index is indicated by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

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 magenta shading 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.

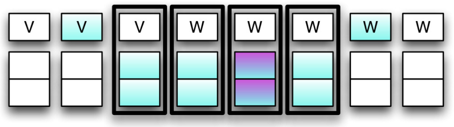


The data contained in a slot changes for each cycle of the write-index around the buffer, so a single read needs to be contained within one cycle to be uncorrupted. Tracing through the algorithm, we can convince ourselves that the state vector data in a slot may not change without that slot first having 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 green shading 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 has both magenta and green shading 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 [11], and on verification of the behavior of sequential programs for robots [8], 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 [7], 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 [10], 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]. HLRG allows us to precisely describe the software, and then apply sound inference rules to reason about it in a mathematically rigorous manner.

HLRG builds upon LRG [1] which combines separation logic [9, 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 [6] 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 $\texttt{writeindex}$ 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 $\texttt{readindex}$ 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.

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

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, $\texttt{writeindex}$, 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:

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

This step must follow the $\mathsf{UpdData}$ step, with some number of intervening steps, all of which must be steps that do not

```
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

change shared state, Id, so we use the temporal operator $\rhd$ 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 the $\mathsf{UpdWrite}$

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

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.

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

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 $\mathsf{UpdRead}$ step, again with some number of transitions that do not change shared state.

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

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.

$$G \stackrel{\text{def}}{=} (\mathsf{Id} \vee \mathsf{UpdData} \vee \mathsf{UpdWrite} \vee \mathsf{UpdVer} \vee \mathsf{UpdRead}) \wedge (I \ltimes I)$$

$$R \stackrel{\text{def}}{=} \mathsf{Id} \wedge (I \ltimes I)$$

$$\mathcal{M} \stackrel{\text{def}}{=} \boxminus(R \vee G)$$

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

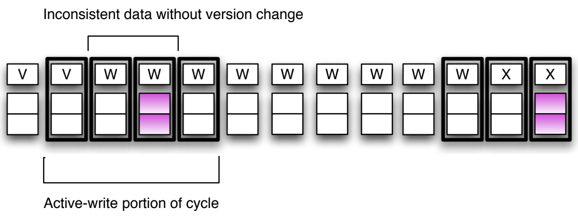
Going through this process is straightforward, once one proves the following, which we call the Stable Data Lemma:

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

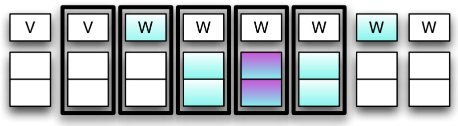
This is an invariant tied to our `Read` algorithm, that says: If, at some time in the past, we looked at the value of $\texttt{version}{+}h$, and then we looked at the state vector in slot h, $\mathsf{Vector}(h)$, and we look at $\texttt{version} + h$ in the present and its value matches what we saw 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.


Inconsistent data without version change
V
V
W
W
W
W
W
W
W
W
W
X
X
Active-write portion of cycle

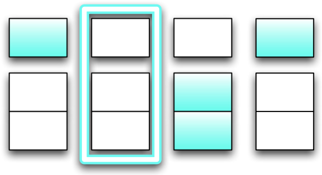
An example interleaving that exhibits this problem is as follows:



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. We modify our `Read` algorithm 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.

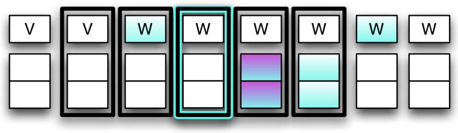


To guarantee that we solved this problem, we rewrote the statement of our lemma to reflect the changes we made:

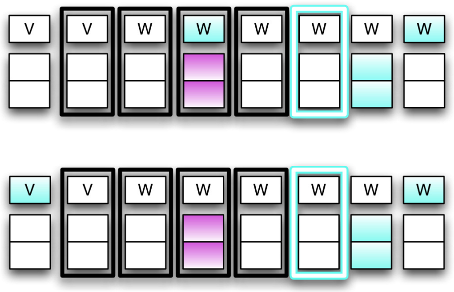
$$((\texttt{version}{+}h \rightsquigarrow X \blacktriangleright \texttt{writeindex} \rightsquigarrow h' \blacktriangleright \mathsf{Vector}(h) \rightsquigarrow D) \wedge (\mathsf{Vector}(h) \rightsquigarrow D' * \texttt{version}{+}h \rightsquigarrow X) \wedge (h \neq h')) \Rightarrow (D = D')$$

This modification of the read algorithm creates 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:

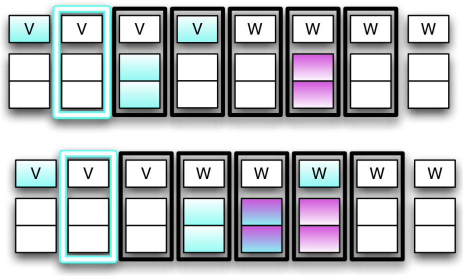


2. When the write-index check indicates that this element is not in the "active write" portion of the cycle, and the active write portion occurred before the write-index check, the data read within this cycle will be consistent unless the read spans multiple cycles, in which case it will be flagged as 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, can now try our proof again. This time, our formal proof by induction succeeds. The failed proof of the stable data lemma, as well as the completed proof of the improved stable data lemma is given in Appendix A, along with a number of other lemmas.

Once the stable data lemma has been proved, we can complete the proof of the read data integrity property that we seek. Adding the write-index check to correct the bug in our program is a small matter. The program, with the appropriate corrections is shown in Fig. 5. The completed proof of read data integrity is given in Appendix B.

4.3 Proving Data Freshness

Unlike the data integrity property, data freshness is expressed as a program invariant. We say that a slot is fresh if the copy of the state vector it contains is the last one changed by any part of the program. We want to make sure that `readindex` always points to the freshest slot. We state this invariant as follows:

$$\begin{array}{l}\mathsf{Vector}(k) \rightsquigarrow X' * \circledast_{i\neq k}\mathsf{Vector}(i) \rightsquigarrow D_i \rhd \\ \mathsf{Vector}(k) \rightsquigarrow X * \circledast_{i\neq k}\mathsf{Vector}(i) \rightsquigarrow D_i \wedge \\ \texttt{readindex} \rightsquigarrow k \wedge (X' \neq X)\end{array}$$

When we attempt this proof, as in the previous case, we discover that it is not true of our system. The details of this proof attempt are given in Appendix C.

What went wrong here? Just as in the previous case, there is a gap of time between when the writer completes the last write in the state vector element, and when it updates the read-index. There is no way to simultaneously complete the last atomic write action and update the read-index. Unlike the previous case, this bug is not cause for alarm, it is the result of an imprecise statement of what we really expect the system to provide for us. We will have to amend our statement of freshness to accept the most recent, or the second most recent copy of the state vector.

We rewrite our invariant to allow the read-index to point

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

```
void Write(int data[N]){
  local old, i, tmp, wr;
  old = writeindex;
  for (i=0;i<N;i++)
    Vector[i][old] = data[i]
  wr = (old + 1) mod H;
  writeindex = wr;
  tmp = version[old] + 1;
  version[wr] = tmp;
  readindex = old;
}
```

```
int Read(int data[N]){
  local rd, wr, curTic1,
        curTic2, i;
  rd = readindex;
  curTic1 = version[rd];
  wr = writeindex;
  if (rd == wr)
    return 0;
  for (i=0;i<N;i++)
    data[i] = Vector[i][rd];
  curTic2 = version[rd];
  if (curTic1 == curTic2)
    return 1;
  else return 0;
}
```

Figure 5: Corrected code

to the second freshest slot:

$$\begin{aligned}
&((\mathsf{Vector}(j) \rightsquigarrow Y * \mathsf{Vector}(k) \rightsquigarrow X' * \\
&\circledast_{i \neq j,k} \mathsf{Vector}(i) \rightsquigarrow D_i) \rhd \\
&(\mathsf{Vector}(j) \rightsquigarrow Y' * \mathsf{Vector}(k) \rightsquigarrow X' * \\
&\circledast_{i \neq j,k} \mathsf{Vector}(i) \rightsquigarrow D_i)) \rhd \\
&(\mathsf{Vector}(j) \rightsquigarrow Y' * \mathsf{Vector}(k) \rightsquigarrow X * \\
&\circledast_{i \neq j,k} \mathsf{Vector}(i) \rightsquigarrow D_i) \wedge \\
&(\texttt{readindex} \rightsquigarrow k \vee \texttt{readindex} \rightsquigarrow j) \wedge \\
&(X' \neq X) \wedge (Y' \neq Y))
\end{aligned}$$

This time, our proof by induction succeeds. The completed proof is presented in Appendix D.

5 Conclusion

We have been able to provide very strong guarantees about the correct functioning of a library that is intended to control a surgical robotic systems. During this process, we found and corrected a subtle bug associated with concurrency. Furthermore, we proved that with respect to these properties, there are no more flaws or bugs in our system. Figure 6 illustrates the process graphically.

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 checkeed, or maintained during further development of the software.

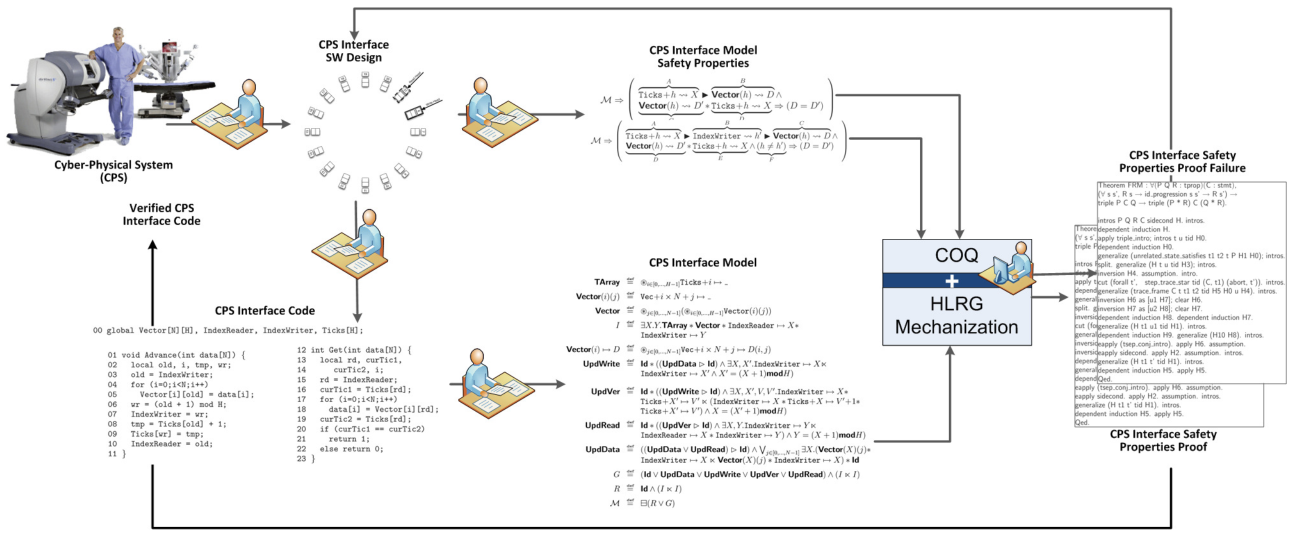
Third, the approach was applied after the software was developed. In order to make this approach practical, there should be a strategy that integrates the approach with the software development process, allowing a flawed design to be identified and corrected sooner.

Fourth, our guarantees of correct operation are local to a particular component of the SAW and very specific in referring to implementation details and behavior. 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 more detailed 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.

A Proof of Improved Stable Data, and Other Lemmas

There are a number of properties that are useful in proving our read data integrity property. In this section, we will state these as lemmas, and prove them. We end this section with a complete proof of the Improved Stable Data lemma.


Cyber-Physical System (CPS)
CPS Interface SW Design
CPS Interface Model Safety Properties
CPS Interface Safety Properties Proof Failure
Verified CPS Interface Code
CPS Interface Code
CPS Interface Model
COQ
+
HLRG Mechanization
CPS Interface Safety Properties Proof

$$\mathcal{M} \Rightarrow \left(\overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\textbf{Vector}(h) \rightsquigarrow D}^{B} \wedge \underbrace{\textbf{Vector}(h) \rightsquigarrow D'}_{C} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{D} \Rightarrow (D = D') \right)$$

$$\mathcal{M} \Rightarrow \left(\overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\texttt{IndexWriter} \rightsquigarrow h'}^{B} \blacktriangleright \overbrace{\textbf{Vector}(h) \rightsquigarrow D}^{C} \wedge \underbrace{\textbf{Vector}(h) \rightsquigarrow D'}_{D} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{E} \wedge \underbrace{(h \neq h')}_{F} \Rightarrow (D = D') \right)$$

```
global Vector[N][H], IndexReader, IndexWriter, Ticks[H];

void Advance(int data[N]) {
  local old, i, tmp, wr;
  old = IndexWriter;
  for (i=0;i<N;i++)
    Vector[i][old] = data[i];
  wr = (old + 1) mod H;
  IndexWriter = wr;
  tmp = Ticks[old] + 1;
  Ticks[wr] = tmp;
  IndexReader = old;
}

int Get(int data[N]) {
  local rd, curTic1,
     curTic2, i;
  rd = IndexReader;
  curTic1 = Ticks[rd];
  for (i=0;i<N;i++)
    data[i] = Vector[i][rd];
  curTic2 = Ticks[rd];
  if (curTic1 == curTic2)
    return 1;
  else return 0;
}
```

$$\begin{aligned}
\textbf{TArray} &\overset{\text{def}}{=} \circledast_{i\in[0,\ldots,H-1]}\texttt{Ticks}+i \mapsto \_ \\
\textbf{Vector}(i)(j) &\overset{\text{def}}{=} \texttt{Vec}+i \times N + j \mapsto \_ \\
\textbf{Vector} &\overset{\text{def}}{=} \circledast_{j\in[0,\ldots,N-1]}(\circledast_{i\in[0,\ldots,H-1]}\texttt{Vector}(i)(j)) \\
I &\overset{\text{def}}{=} \exists X.Y.\textbf{TArray} * \textbf{Vector} * \texttt{IndexReader} \mapsto X * \\
&\quad \texttt{IndexWriter} \mapsto Y \\
\textbf{Vector}(i) \mapsto D &\overset{\text{def}}{=} \circledast_{j\in[0,\ldots,N-1]}\texttt{Vec}+i \times N + j \mapsto D(i,j) \\
\textbf{UpdWrite} &\overset{\text{def}}{=} \textbf{Id} * ((\textbf{UpdData} \rhd \textbf{Id}) \wedge \exists X, X'.\texttt{IndexWriter} \mapsto X \ltimes \\
&\quad \texttt{IndexWriter} \mapsto X' \wedge X' = (X+1)\textbf{mod}H) \\
\textbf{UpdVer} &\overset{\text{def}}{=} \textbf{Id} * ((\textbf{UpdWrite} \rhd \textbf{Id}) \wedge \exists X, X', V, V'.\texttt{IndexWriter} \mapsto X * \\
&\quad \texttt{Ticks}+X' \mapsto V' \ltimes (\texttt{IndexWriter} \mapsto X * \texttt{Ticks}+X \mapsto V'+1 * \\
&\quad \texttt{Ticks}+X' \mapsto V') \wedge X = (X'+1)\textbf{mod}H) \\
\textbf{UpdRead} &\overset{\text{def}}{=} \textbf{Id} * ((\textbf{UpdVer} \rhd \textbf{Id}) \wedge \exists X, Y.\texttt{IndexWriter} \mapsto Y \ltimes \\
&\quad \texttt{IndexReader} \mapsto X * \texttt{IndexWriter} \mapsto Y) \wedge Y = (X+1)\textbf{mod}H) \\
\textbf{UpdData} &\overset{\text{def}}{=} ((\textbf{UpdData} \vee \textbf{UpdRead}) \rhd \textbf{Id}) \wedge \textstyle\bigvee_{j\in[0,\ldots,N-1]} \exists X.(\textbf{Vector}(X)(j) * \\
&\quad \texttt{IndexWriter} \mapsto X \ltimes \textbf{Vector}(X)(j) * \texttt{IndexWriter} \mapsto X) * \textbf{Id} \\
G &\overset{\text{def}}{=} (\textbf{Id} \vee \textbf{UpdData} \vee \textbf{UpdWrite} \vee \textbf{UpdVer} \vee \textbf{UpdRead}) \wedge (I \ltimes I) \\
R &\overset{\text{def}}{=} \textbf{Id} \wedge (I \ltimes I) \\
\mathcal{M} &\overset{\text{def}}{=} \boxminus(R \vee G)
\end{aligned}$$

```
Theorem FRM : ∀(P Q R : tprop)(C : stmt),
(∀ s s', R s → id_progression s s' → R s') →
triple P C Q → triple (P * R) C (Q * R).

intros P Q R C sidecond H. intros.
dependent induction H.
apply triple_intro; intros t u tid H0.
dependent induction H0.
generalize (unrelated_state_satisfies t1 t2 t P H1 H0); intros.
split. generalize (H t u tid H3); intros.
inversion H4. assumption. intro.
cut (forall t',   step_trace_star tid (C, t1) (abort, t')). intros.
generalize (trace_frame C t t1 t2 tid H5 H0 u H4). intros.
inversion H6 as [u1 H7]; clear H6.
inversion H7 as [u2 H8]; clear H7.
dependent induction H8. dependent induction H7.
generalize (H t1 u1 tid H1). intros.
dependent induction H9. generalize (H10 H8). intros.
eapply (tsep_conj_intro). apply H6. assumption.
eapply sidecond. apply H2. assumption. intros.
generalize (H t1 t' tid H1). intros.
dependent induction H5. apply H5.
Qed.
```

Figure 6: The process of verifying code after it has been developed. The left-hand side of the figure shows the software developer looking at system requirements, creating a design, and implementing it in code. The design is used to create properties we wish to guarantee, and the code is transformed into an HLRG representation that models its features. We then reason about the model, using sound inference rules, to show that the properties hold. If the proofs fail, then (illustrated by the topmost arrow) the design must be modified, as was the case for this study. If the proof succeeds, then (as illustrated by the bottommost arrow) the software may be used in the system, and we can assert and expect that the guarantees we proved hold in practice.

A.1 Cyclical Update Lemma

The program cycles endlessly through a fixed sequence of atomic transitions described by the following list:

$$\mathsf{UpdWrite} :: \mathsf{UpdVer} :: \mathsf{UpdRead} :: \mathsf{UpdData}+$$

List elements are separated by double colons ::, and $+$ is the Kleene plus, indicating that the preceeding state may have occurred more than once.

We prove this lemma by inpection of the predicates that we used to describe atomic transitions with attention to the portion of the predicates that enforce ordering:

$$\begin{aligned}
\mathsf{UpdWrite} &\overset{\text{def}}{=} \cdots(\mathsf{UpdData} \rhd \mathsf{Id})\cdots \\
\mathsf{UpdVer} &\overset{\text{def}}{=} \cdots(\mathsf{UpdWrite} \rhd \mathsf{Id})\cdots \\
\mathsf{UpdRead} &\overset{\text{def}}{=} \cdots(\mathsf{UpdVer} \rhd \mathsf{Id})\cdots \\
\mathsf{UpdData} &\overset{\text{def}}{=} \cdots(\mathsf{UpdData} \vee \mathsf{UpdRead}) \rhd \mathsf{Id})\cdots
\end{aligned}$$

The definition of $\mathsf{UpdWrite}$ says that this state transition must have been preceeded by zero or more identity transitions, which were in turn preceeded by an $\mathsf{UpdData}$ state transition. The only variation in the predictable, straight-line sequence of steps is the possible repetition of the $\mathsf{UpdData}$ predicate more than once. We also note that the structure of the predicate does not contain any other disjunctive terms, so the linear ordering follows. ∎

A.2 Monotonic Version Lemma

A given version number is monotonic.

$$\texttt{version}+h \rightsquigarrow X \blacktriangleright \texttt{version}+h \rightsquigarrow X' \Rightarrow (X \le X')$$

The only step in a trace that affects $\texttt{Ticks}+h$ is the UpdVer step, and only when $\texttt{writeindex} \rightsquigarrow h$. Each time a trace enters this state, it increments $\texttt{Ticks}+h$ by 1. Assume the implicant. A given trace will enter this state $n \in \mathbb{N}$ times. Consequently, $X' = X + n$, so for $X \in \mathbb{N}$, $(X \le X')$. ∎

A.3 Constant Version Lemma

If we measured the version of a slot at some time in the past, and its value then is the same is its value now, its value at some time in between these points is also the same.

$$\texttt{version}+h \rightsquigarrow X \blacktriangleright \texttt{version}+h \rightsquigarrow X' \blacktriangleright \texttt{version}+h \rightsquigarrow X'' \Rightarrow ((X = X'') \Rightarrow (X = X'))$$

Assume the implicants, that $\texttt{version}+h \rightsquigarrow X \blacktriangleright \texttt{version}+h \rightsquigarrow X'$, $\texttt{version}+h \rightsquigarrow X' \blacktriangleright \texttt{version}+h \rightsquigarrow X''$, and $(X = X'')$. Apply the monotonic version lemma to the first two terms, and conclude that $X \le X'$ and $X' \le X''$. Substitute $X = X''$ in the second term, and we find $X \le X' \le X$. Consequently, $(X = X')$. ∎

A.4 Continuously Constant Version Lemma

If we measured the version of a slot at some time in the past, and its value then is the same as its value now, its value between these points was the same at every point in time. Our formalization for this lemma includes some English:

If

$$\diamondminus\, (\texttt{Ticks}+X \rightsquigarrow Y) \wedge (\texttt{Ticks}+X \rightsquigarrow Y)$$

then we can conclude that from the initial time when we first measured the version, until now, $(\texttt{Ticks}+X \rightsquigarrow Y)$ held continuously.

Assume the implicant, that the first term is satsfied by a state in the trace T, s_i, and that the second term is satisfied by the current state in the trace $T.last$.

For each state s' in between s_i and $T.last$, we can measure $(\texttt{Ticks}+X)$, and by application of the constant version lemmawe can show that $(\texttt{Ticks}+X \rightsquigarrow Y)$ forall s'. States s_i and $T.last$ satisfy $(\texttt{Ticks}+X \rightsquigarrow Y)$, by the assumption of the implicant.

Consequently, $(\texttt{Ticks}+X \rightsquigarrow Y)$ held continuously, from the first time we looked at `Ticks`, until the present moment. ∎

A.5 Stable Data Lemma (Attempted)

$$\overbrace{\texttt{version}+h \rightsquigarrow X}^{A} \blacktriangleright \mathsf{Vector}(h) \rightsquigarrow D \wedge$$
$$\mathsf{Vector}(h) \rightsquigarrow D' * \underbrace{\texttt{version}+h \rightsquigarrow X}_{B} \Rightarrow (D = D')$$

We will do a proof by induction, inducting over the steps in a trace. We begin by proving that $i \Rightarrow i+1$. Different terms of the implicant have been named using capital letters, as indicated by the brackets in the statement of the lemma. These letters will be used during the proof to refer to each term.

Assume we have a trace T that satisfies this lemma. Show that any step taken produces a trace T' that also satisfies this lemma. The possible steps that can be taken are:

1. UpdWrite Changes `writeindex` in the current state, but the predicate refers to this variable in the past, so it does not affect the truth of the predicate.

2. UpdVer Changes $\texttt{Ticks}+h$ when $\texttt{writeindex} \rightsquigarrow h$. It could falsify the implicant, but this could only preserve the truth of the predicate.

3. UpdRead Does not affect any terms in the prediate.

4. UpdData This step could change D' when $\texttt{writeindex} \rightsquigarrow h$, and falsify the implicand. When this occurs, we know from the cyclical update lemma that it must have been preceeded by the following state-changing steps, in the following order:

   $$\mathsf{UpdWrite} :: \mathsf{UpdVer} :: \mathsf{UpdRead} :: \mathsf{UpdData}+$$

   In order to show that this step does not falsify the predicate, we must show that every time the implicand is falsified, the implicant is falsified as well.

   If term A were satisfied by a state in the trace before the UpdVer step in the sequence, then the most recent UpdVer step would have been taken with $\texttt{writeindex} \rightsquigarrow h$, falsifying term B, preserving the truth of the predicate, since UpdVer never repeats version numbers and no other predicate changes the Ticks binding.

   If term A were satisfied by a state in the trace after the UpdVer step in the sequence, then terms A and B would be satisfied, and the implicant would be true. This falsifies our predicate.

This lemma cannot be proven!

A.6 Improved Stable Data Lemma

$$\overbrace{\texttt{version}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\texttt{writeindex} \rightsquigarrow h'}^{B} \blacktriangleright$$
$$\mathsf{Vector}(h) \rightsquigarrow D \wedge \mathsf{Vector}(h) \rightsquigarrow D' *$$
$$\underbrace{\texttt{version}+h \rightsquigarrow X}_{E} \wedge \underbrace{(h \neq h')}_{F} \Rightarrow (D = D')$$

Our proof is by induction, inducting over the steps in a trace. We begin by proving that $i \Rightarrow i+1$. Different terms of the implicant have been named using capital letters, as indicated by the brackets in the statement of the lemma. These letters will be used during the proof to refer to each term, as before.

Assume we have a trace T that satisfies this lemma. Show that any step taken produces a trace T' that also satisfies this lemma. The possible steps that can be taken are:

1. UpdWrite Changes `writeindex` in the current state, but the predicate refers to this variable in the past, so it does not affect the truth of the predicate.

2. UpdVer Changes $\texttt{Ticks}+h$ when $\texttt{writeindex} \rightsquigarrow h$. It could falsify the implicant, but this could only preserve the truth of the predicate.

3. UpdRead Does not affect any terms in the prediate.

4. UpdData This step could change D' when `writeindex` $\rightsquigarrow h$, and falsify the implicand. When this occurs, it must have been preceeded by the following state-changing steps, in the following order:

   $$\mathsf{UpdWrite} :: \mathsf{UpdVer} :: \mathsf{UpdRead} :: \mathsf{UpdData}+$$

   In order to show that this step does not falsify the predicate, we must show that every time the implicand is falsified, the implicant is falsified as well.

   If term A were satisfied by a state in the trace before the UpdVer step in the sequence, then the most recent UpdVer step would have been taken with `writeindex` $\rightsquigarrow h$, falsifying term E, preserving the truth of the predicate, since UpdVer never repeats version numbers and no other predicate chanes the Ticks binding.

   If term A were satisfied by a state in the trace after the UpdVer step in the sequence, then terms A and E would be satisfied, and points in the trace available to satisfy term B must occur after the UpdVer step as well, when `writeindex` $\rightsquigarrow h$, falsifying term F, and preserving the truth of this predicate.

Now prove the base case to be true. Any trace that only has one state, where the write-index does not point to the slot h satisfies the implicant and satisfies the whole predicate. Traces with a single state where the write-index points to h falsify the implicant, also satisfying the predicate.

We have shown that the base case satisfies our predicate, and we have shown that for any trace that satisfies the predicate, any one step also produces a trace that satisfies the predicate. By induction, we can conclude that our predicate holds for any trace produced by our system. ■

B Proof of Data Integrity

This appendix gives the completed proof of data integrity property, following the strategy we described in Section 4.2.

Figures 7 and 10 show the propagation of the predicate starting with the precondition of true, and ending at `return 1;` indicating a successful read. In these figures, lines of the function are shown in between predicates so it is clear where in the program each predicate applies. Predicates are shown in curly braces.

Figures 8 and 9 show in detail the transformations that we apply to the predicate to reason about the non-atomic reading of state (i.e. lines 20 and 21 in the program) in the presence of a concurrent thread running `Write`. In the original SAW C++ source code, this read is a single instruction; here it has been transformed into a loop to remind ourselves of its non-atomic nature.

The precondition for the successful completion of the program shown in Fig. 10 contains `data` $= D$, and at some range of time time during this program, $\mathsf{Vector}(X) \rightsquigarrow D$. We know from the properties of the `Write` that for any range of time where a slot is guaranteed not to be written, the slot may be considered uncorrupted.

We conclude that when the read successfully completes, the value returned accurately reflects what was stored in memory for that state vector element during the read; and that value was stable and uncorrupted during the read, i.e. no writer was altering it or may have altered it during that time. ■

C Proof of Data Freshness (Attempted)

We attempt to prove:

$$\begin{aligned}&\mathsf{Vector}(k) \rightsquigarrow X' * \circledast_{i\neq k}\mathsf{Vector}(i) \rightsquigarrow D_i \rhd\\ &\mathsf{Vector}(k) \rightsquigarrow X * \circledast_{i\neq k}\mathsf{Vector}(i) \rightsquigarrow D_i \wedge\\ &\texttt{readindex} \rightsquigarrow k \wedge (X' \neq X)\end{aligned}$$

We will do a proof by induction, inducting over the steps in a trace. We begin by proving that $i \Rightarrow i+1$.

Assume we have a trace T that satisfies this lemma. Show that any step taken produces a trace T' that also satisfies this lemma. The possible steps that can be taken are:

1. UpdWrite Has no effect.

2. UpdVer Has no effect.

3. UpdRead This step changes the read-index, pointing it to the element that the write-index was pointing to before its current position.

   We again invoke the cyclical update lemma and observe that this step must have been preceeded by the following sequence of state-changing steps (with the possible addition of identity steps between each of these):

   $$\mathsf{UpdRead} :: \mathsf{UpdData}+ :: \mathsf{UpdWrite} :: \mathsf{UpdVer}$$

   If before this step, the read-index was pointing to the freshest index k, then `readindex` $\rightsquigarrow k$ and the last UpdData must have been conducted with `writeindex` $\rightsquigarrow k$. However `writeindex` is now something other than k, since it was changed by the most recent UpdWrite. The read-index is going to be changed to what the write-index was before, namely k. So the read-index remains pointing at the most recently changed slot and the predicate continues to hold for the new trace T'.

4. UpdData This action could change the state vector element and falsify our predicate. If the predicate is

$\{\mathsf{true}\}$

```
12 int Get(int data[N]){
13   local rd, wr, curTic1,
14         curTic2, i;
15   rd = readindex;
```

$\{\exists X. \diamondminus\ \mathtt{readindex} \rightsquigarrow X \wedge (\mathtt{rd} = X)\}$

```
16   curTic1 = Ticks[rd];
```

$\{\exists XY.\mathtt{readindex} \rightsquigarrow X \blacktriangleright \mathtt{version}{+}X \rightsquigarrow Y \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X)\}$

$\{\exists XY. \diamondminus\ \mathtt{version}{+}X \rightsquigarrow Y \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X)\}$

```
17   wr = writeindex;
```

$\{\exists X X' Y.\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge (\mathtt{wr} = X') \wedge$
$(\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X)\}$

```
18   if (rd == wr)
19     return 0;
```

$\{\exists X X' Y.\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge (\mathtt{wr} = X') \wedge$
$(\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X')\}$

Figure 7: Beginning the proof of the data integrity property. A straightforward collection of state. Before line 17, we drop the information about having the value of the read-index, because it is not necessary for the proof of data integrity.

$\{\exists X X' Y.\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge (\mathtt{wr} = X') \wedge$
$(\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X')\}$

$\{\exists X X' Y.\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge \mathtt{version}{+}X \rightsquigarrow Y' * \mathsf{Vector}(X) \rightsquigarrow D \wedge$
$(\mathtt{wr} = X') \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X')\}$

$\{\exists X X' Y.$
$(\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge \mathtt{version}{+}X \rightsquigarrow Y' * \mathsf{Vector}(X) \rightsquigarrow D \wedge$
$(\mathtt{wr} = X') \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X') \wedge (Y \neq Y')) \vee$
$(\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge \mathtt{version}{+}X \rightsquigarrow Y * \mathsf{Vector}(X) \rightsquigarrow D \wedge$
$(\mathtt{wr} = X') \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X'))\}$

Figure 8: We apply sound inference rules to transform the postcondition of line 19 into two cases, one where the version for the slot from which we are reading has changed, and the other where the version has stayed constant.

$\{\exists X X' Y.$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \blacktriangleright (\texttt{version+}X \leadsto Y' * \mathsf{Vector}(X) \leadsto D) \wedge$
$(\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X') \wedge (Y \neq Y')) \vee$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \blacktriangleright (\texttt{version+}X \leadsto Y * \mathsf{Vector}(X) \leadsto D) \wedge$
$\wedge \texttt{version+}X \leadsto Y * \mathsf{Vector}(X) \leadsto D' \wedge (\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X')) \vee$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \blacktriangleright (\texttt{version+}X \leadsto Y * \mathsf{Vector}(X) \leadsto D) \wedge$
$\wedge \texttt{version+}X \leadsto Y'' * \mathsf{Vector}(X) \leadsto D' \wedge (\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X') \wedge (Y \neq Y''))\}$

$\{\exists X X' Y.$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \blacktriangleright (\texttt{version+}X \leadsto Y' * \mathsf{Vector}(X) \leadsto D) \wedge$
$(\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X') \wedge (Y \neq Y')) \vee$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \blacktriangleright (\texttt{version+}X \leadsto Y * \mathsf{Vector}(X) \leadsto D) \wedge$
$\wedge \texttt{version+}X \leadsto Y * \mathsf{Vector}(X) \leadsto D' \wedge$
$(\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X'))\}$

$\{\exists X X' Y.$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \blacktriangleright (\texttt{version+}X \leadsto Y' * \mathsf{Vector}(X) \leadsto D) \wedge$
$(\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X') \wedge (Y \neq Y')) \vee$
$(\texttt{version+}X \leadsto Y \blacktriangleright \texttt{writeindex} \leadsto X' \triangleright (\texttt{version+}X \leadsto Y * \mathsf{Vector}(X) \leadsto D) \wedge$
$(\texttt{wr} = X') \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X) \wedge (X \neq X'))\}$

$\{\exists X X' Y.$
$Q_1 \vee Q_2\}$

Figure 9: The first predicate shown is the result of allowing our system to evolve a single step past line 19, allowing us to transform the resulting predicate in Fig. 8 to distinguish two new cases. The first disjunctive term is the same as the first disjunctive term of the result of Fig. 8, except time has passed that has made the then present state part of the history. The second and third terms are the new cases associated with having the second version check match the first, and having a third version checks match and not match, respectively. The second predicate shown is the result of collapsing the first and third disjunctive terms into one by waiting for the present moment to pass into history and ignoring some of the state we have collected in the third. The third predicate uses the three identical measurements of the version in the history to apply the continuous version lemma, and show that the version number must have been this value continuously between the original accumulation of the $\mathsf{Vector}(X) \leadsto D$ term in this predicate and the present moment. We can then apply the stable data lemma to show that since $\texttt{version+}X \leadsto Y$, $\mathsf{Vector}(X) \leadsto D$ during this time period. Finally, in the fourth predicate, we have simply named the disjunctive terms of the third predicate Q_1 and Q_2 for ease of reference. This is an invariant that applies at each moment during the for loop in lines 20 and 21.

$\{\exists X X' Y.\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \wedge (\mathtt{wr} = X')\wedge$
$(\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X')\}$

```
20   for (i=0;i<N;i++)
21     data[i] = Vector[i][rd];
```

$\{\exists X X' Y.$
$(\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \blacktriangleright (\mathtt{version}{+}X \rightsquigarrow Y' * \mathsf{Vector}(X) \rightsquigarrow D)\wedge$
$(\mathtt{wr} = X') \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X') \wedge (Y \neq Y') \wedge (\mathtt{data} = D'))\vee$
$(\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \triangleright (\mathtt{version}{+}X \rightsquigarrow Y * \mathsf{Vector}(X) \rightsquigarrow D)\wedge$
$(\mathtt{wr} = X') \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X') \wedge (\mathtt{data} = D))$

$\{\exists X X' Y.(Q_1 \wedge (\mathtt{data} = D'))\vee$
$(Q_2 \wedge (\mathtt{data} = D))\}$

```
22   curTic2 = Ticks[rd];
```

$\{\exists X X' Y.(Q_1 \blacktriangleright \mathtt{version}{+}X \rightsquigarrow Y'' \wedge (\mathtt{data} = D') \wedge (\mathtt{curTic2} = Y'') \wedge (Y' \neq Y))\vee$
$(Q_2 \wedge (\mathtt{data} = D) \wedge (\mathtt{curTic2} = Y))\}$

```
23   if (curTic1 == curTic2)
```

$\{\exists X X' Y.Q_2 \wedge (\mathtt{data} = D) \wedge (\mathtt{curTic2} = Y)\}$

$\{\exists X X' Y.\mathtt{version}{+}X \rightsquigarrow Y \blacktriangleright \mathtt{writeindex} \rightsquigarrow X' \triangleright (\mathtt{version}{+}X \rightsquigarrow Y * \mathsf{Vector}(X) \rightsquigarrow D)\wedge$
$(\mathtt{wr} = X') \wedge (\mathtt{curTic1} = Y) \wedge (\mathtt{rd} = X) \wedge (X \neq X') \wedge (\mathtt{data} = D) \wedge (\mathtt{curTic2} = Y)\}$

```
24     return 1;
25   else return 0;
26 }
```

Figure 10: Using the series of transformations described in Figs. 8 and 9, we can reason about the propagation of the predicate past the non-atomic read in lines 20 and 21. At the completion of the read, $\mathtt{data} = D'$, and some subset of these moments in the period between the first version check and the present are used to read the state vector into the local variable `data`. If the value of $\mathsf{Vector}(X)$ was constant, then we can conclude that at the end of the read, $D = D'$.

pointing to the most recently changed slot, then changing data in another slot automatically makes this into the second most recently changed element. The second most recently changed element is not the same as the most recently changed element, since elements are altered sequentially by the action of the writer. We have falsified our predicate.

This statement of freshness is not provable for our program.

D Proof of Data Freshness

We prove:

$$\begin{aligned}
&((\mathsf{Vector}(j) \rightsquigarrow Y * \mathsf{Vector}(k) \rightsquigarrow X' * \\
&\circledast_{i\neq j,k}\mathsf{Vector}(i) \rightsquigarrow D_i) \triangleright \\
&(\mathsf{Vector}(j) \rightsquigarrow Y' * \mathsf{Vector}(k) \rightsquigarrow X' * \\
&\circledast_{i\neq j,k}\mathsf{Vector}(i) \rightsquigarrow D_i)) \triangleright \\
&(\mathsf{Vector}(j) \rightsquigarrow Y' * \mathsf{Vector}(k) \rightsquigarrow X * \\
&\circledast_{i\neq j,k}\mathsf{Vector}(i) \rightsquigarrow D_i) \wedge \\
&(\texttt{readindex} \rightsquigarrow k \vee \texttt{readindex} \rightsquigarrow j) \wedge \\
&(X' \neq X) \wedge (Y' \neq Y))
\end{aligned}$$

We approach the proof by induction, inducting over the steps in a trace. We begin by proving that $i \Rightarrow i+1$.

Assume we have a trace T that satisfies this lemma. Show that any step taken produces a trace T' that also satisfies this lemma. The possible steps that can be taken are:

1. UpdWrite Has no effect.

2. UpdVer Has no effect.

3. UpdRead This step changes the read-index, pointing it to the element that the write-index was pointing to before its current position.

   We again invoke the cyclical update lemma and observe that this step must have been preceeded by the following sequence of state-changing steps (with the possible addition of identity steps between each of these):

   $$\mathsf{UpdRead} :: \mathsf{UpdData}+ :: \mathsf{UpdWrite} :: \mathsf{UpdVer}$$

   If before this step, the read-index was pointing to the freshest index k, then the argument we presented in our earlier attempt is still valid, and the predicate continues to hold for the new trace T'.

   If before this step, the read-index was pointing to the second-freshest index j, then the most recently change to the read-index points to what the write-index was before its most recent change; which incidentally is k, the index of the most recently altered slot. So if the read-index was pointing to the second-freshest index, and it is updated with this atomic step, it will end up pointing to the freshest index.

4. UpdData This action could change the state vector element and falsify our predicate. If the predicate is pointing to the most recently changed slot, then changing data in another slot automatically makes this into the second most recently changed element. This is acceptable to this predicate, unlike in our first attempt.

   If, however, the read-index is pointing to the second most recently changed slot, j, then there are two possibilities that we can see from the cyclical update lemma. The first is the possibility that this UpdData step is the first that applies to this index in this cycle, and the history looks something like this:

   $$\mathsf{UpdData}+ :: \mathsf{UpdWrite} :: \mathsf{UpdVer} :: \mathsf{UpdRead}$$

   with the previous UpdData applying to another index.

   If that is the case, then the read-index was updated in the state change just previous to this, to be the value of the write-index just before its most recent value. That value is the most recently modified slot. So the read-index must be pointing to the most recent slot as well as the one before that. The second most recently changed index is not equal to the most recently changed index, again using the cyclical update lemma. Thus we have a contradiction, and the read-index must be pointing to the freshest slot.

   If instead, the state transition history looks like this:

   $$\begin{aligned}&\mathsf{UpdData}+ :: \mathsf{UpdWrite} :: \mathsf{UpdVer} :: \\ &\mathsf{UpdRead} :: \mathsf{UpdData}+\end{aligned}$$

   then we can begin with the assumption that the read-index points to the second-most recently changed slot, j, and further modifications to this slot will not change anything, since the most recently modified state vector copy is the current one under the write-index, and any changes still apply to it.

We can conclude that `writeindex` must be pointing to the freshest or the next freshest written slot at all times. ■

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] 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.

[7] 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.

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

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

[10] 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.

[11] 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.