Applying Mathematical Logic to Create Zero-Defect Software

Yanni Kouskoulas, Ming Fu, Zhong Shao, and Peter Kazanzides

This article introduces formal methods (FM) approaches in the context of zero-defect software development and also presents a tutorial-style case study in which we apply a program logic called History for Local Rely/Guarantee (HLRG) to create a zero-defect software component for a surgical robot software library. During this analysis, we found a bug in the system that could adversely affect safety. We demonstrate the additional utility of incorporating FM into the design process by fixing the bug and proving that no more bugs exist in that component that could impact the property we were analyzing.

INTRODUCTION

Motivation

Innovative software engineering solutions are often difficult to design and difficult to implement. Even a small component may have complex behavior, and this can make it difficult for the developer to hold the whole of the design in working memory. There may be unexpected corner cases where unknown inputs cause the system to malfunction. Creating zero-defect software is a challenging goal.

Applying sound mathematical logic to reason about our problems has been the cornerstone of progress in science and engineering. Successful scientists and engineers have a conceptual model of what they are working with in their heads, and they apply sound reasoning to this model to understand and solve the problems they encounter.

In engineering, the more that the model and reasoning are formalized (i.e., documented precisely), the closer we can get to the ideal of developing zero-defect components and systems. We can closely approach this ideal in many areas of engineering. For example, we have good mathematical models of basic physics, circuits, antennas, structural components, aircraft, and even chemical processes. We use continuous mathematics to model and reason about the engineering solutions that we create, solving equations that come from our models and constructing proofs about the models' behavior to aid in the design process.

However, because of the discrete nature of software, it has been difficult to add mathematical rigor to the models we use during software engineering. On the one hand, modeling software does not seem to be a significant obstacle; software is a precise and descriptive formal model of itself. On the other hand, the continuous mathematics we successfully apply in so many other domains does not help us translate the mental reasoning process that we use to think about software into a formal approach.

Formal Methods

To be able to guarantee proper operation of a software component in the same fashion that we compute the behavior of a transistor or the loads on a beam, we must develop discrete logics that are tailored to the programming languages in which we develop our software. Formal methods (FM) is the name that people often apply to approaches that can help us prove that our software does what we design it to do.

FM can provide mathematically rigorous means of ensuring a software component's behavior for all possible inputs, providing a completeness that is unattainable by using testing alone. These approaches can eventually lead to written proofs, which are useful for software components that have complicated algorithms with nonobvious behavior.

These proofs are formalized versions of the models and reasoning that already exist in the heads of the engineers who are doing the system design and implementation. Any engineer who has an idea of why what he or she is building is going to work also has in his or her head an informal proof of correctness. If the engineer has not thought of a proof, then he or she does not have any idea why the code or circuit might work, and it probably will not. The challenge is getting the proof out of the engineer's head and onto paper, then checking all of its details to make sure that there are not any mistakes.

FM for software and other discrete systems have been developed by mathematicians, logicians, and computer scientists for the past 40 years, and perhaps longer depending on when you start counting. This problem is challenging, but the community has made significant progress in recent years, applying a rigorous mathematical logic to different levels of software abstraction and different components.

Each formal method has three components, which allows us to write down our proofs:

1. A language we can use to model the system
2. A language we can use to describe the system's behavior
3. A sound reasoning strategy that will enable a rigorous mathematical proof that the system we modeled has the behavior we describe

The sound reasoning strategy is simply a set of rules that describe the conclusions we can draw about the computation of each instruction. Often, this combination of elements is called a "logic" because it is used to reason about systems in a particular domain. Once we have a logic, we can write down the steps of our proofs and check them to ensure that we have not made any mistakes.

Existing FM are often grouped into two categories: model checking and deductive verification. These categories differ in the expressiveness of the modeling language and the reasoning strategy. In this case, expressiveness of the language is a qualitative measure of the range of different systems it can describe. For example, the NuSMV model checker is limited to analyzing models of nondeterministic, finite-state automata. Some systems can be modeled accurately using this formalism, whereas others cannot.

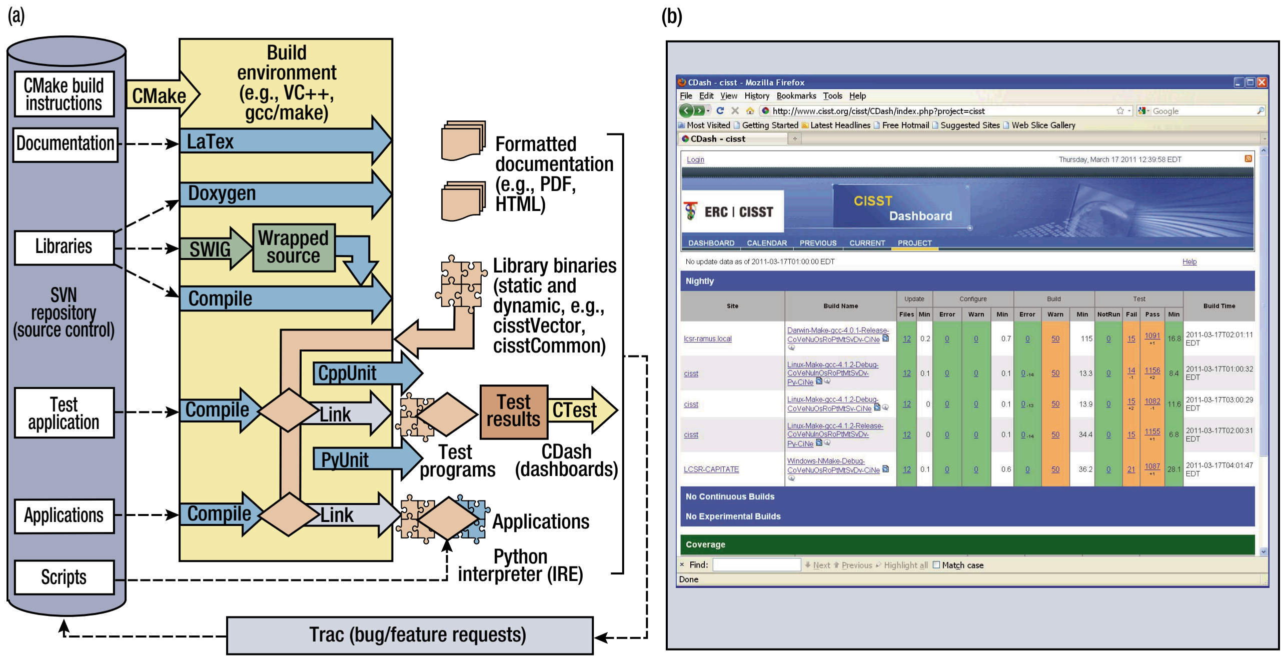
Models for model checking are limited in some way, often to some sort of state machine, so that the state space can be limited; the sound reasoning strategy for model checkers is a brute-force exploration of the state space. An often difficult aspect of model checking is creating the model with just enough detail to capture the essence of the problem but not so much detail that it causes the state space to be too large to explore. Once the model is constructed and the desired behavior is described, proving a property of a system by using a model checker is as easy as pressing a button and waiting for the computer to explore all of the states.

Models for deductive verification are often written in a very expressive modeling language. It is often the case that because of the modeling language's flexibility, much of the difficulty inherent in deductive verification is in doing the proof, not in creating the model. The proofs for such an approach cannot be fully automated in all cases and require human intervention and guidance to show the essence of why the property we seek is true.

We will describe a case study, presented in Refs. 1 and 2, in which we apply a program logic called History for Local Rely/Guarantee (HLRG)[3] to certify the integrity of a data-exchange mechanism in a concurrent system. The system is part of a library called the Surgical Assistant Workstation (SAW), which is designed to support surgical robots.

SURGICAL ROBOT SOFTWARE

The target of our analysis is a software framework called the SAW, which was created by the National Science Foundation Engineering Research Center for Computer Integrated Surgical Systems and Technology (CISST ERC) at The Johns Hopkins University (JHU), in partnership with Intuitive Surgical, Inc., developer of the da Vinci® surgical robot. The SAW, described in Ref. 4, provides a modular, open-source component-based software framework to support prototyping of medical robotics and computer-assisted surgery systems. It utilizes the *cisst* C++ libraries to provide basic foundation classes (data types such as vectors, matrices, and transformations, and tools such as class and object registries, logging, etc.) and a component-based framework that supports different execution models, such as periodic threads, callbacks, and event-based programming. The



http://www.cisst.org/cisst/CDash/index.php?project=cisst

Thursday, March 17 2011 12:39:58 EDT

No update data as of 2011-03-17T01:00:00 EDT

Nightly

Site	Build Name	Update		Configure			Build			Test				Build Time
		Files	Min	Error	Warn	Min	Error	Warn	Min	NotRun	Fail	Pass	Min	
lcsr-ramus.local	Darwin-Make-gcc-4.0.1-Release-CoVeNuOsRoPtMtSvDv-CiNe	12	0.2	0	0	0.7	0	50	115	0	15	1091 +1	16.8	2011-03-17T02:01:11 EDT
cisst	Linux-Make-gcc-4.1.2-Debug-CoVeNuInOsRoPtMtSvDv-Py-CiNe	12	0.1	0	0	0.1	0 -14	50	13.3	0	14 -1	1156 +2	8.4	2011-03-17T01:00:32 EDT
cisst	Linux-Make-gcc-4.1.2-Debug-CoVeNuOsRoPtMtSv-CiNe	12	0	0	0	0.1	0 -13	50	13.9	0	15 +2	1082 -1	11.6	2011-03-17T03:00:29 EDT
cisst	Linux-Make-gcc-4.1.2-Release-CoVeNuInOsRoPtMtSvDv-Py-CiNe	12	0	0	0	0.1	0 -14	50	34.4	0	15	1155 +1	6.8	2011-03-17T02:00:31 EDT
LCSR-CAPITATE	Windows-NMake-Debug-CoVeNuOsRoPtMtSvDv-CiNe	12	0.1	0	0	0.6	0	50	36.2	0	21	1087 +1	28.1	2011-03-17T04:01:47 EDT

No Continuous Builds

No Experimental Builds

Coverage

Figure 1. (a) The *cisst*/SAW software development process and tools. (b) The automated test suite is based on CppUnit and PyUnit for the C++ and Python code, respectively; results are reported to the CDash web-based dashboard (right).

SAW includes a number of interface components, which encapsulate hardware devices, and software components such as robot motion, collaborative control, speech recognition, 3-D user interface, and video processing. A key aspect is the definition of interface standards, within the SAW framework, that enable "plug-and-play" configuration of systems. For example, although robots are physically different, their interfaces (e.g., command names and parameters) have been standardized as much as possible.

SAW is currently used for research with the da Vinci surgical system, the JHU microsurgery workstation, and other surgical robotic systems.[4] 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 preoperative computed tomography or magnetic resonance images onto the surgical field; it could enforce no-cut volumes, preventing the surgeon from cutting into tissue or organs that he or she identified volumetrically on preoperative images; or it could actually make cuts based on the surgeon's preoperative plan.

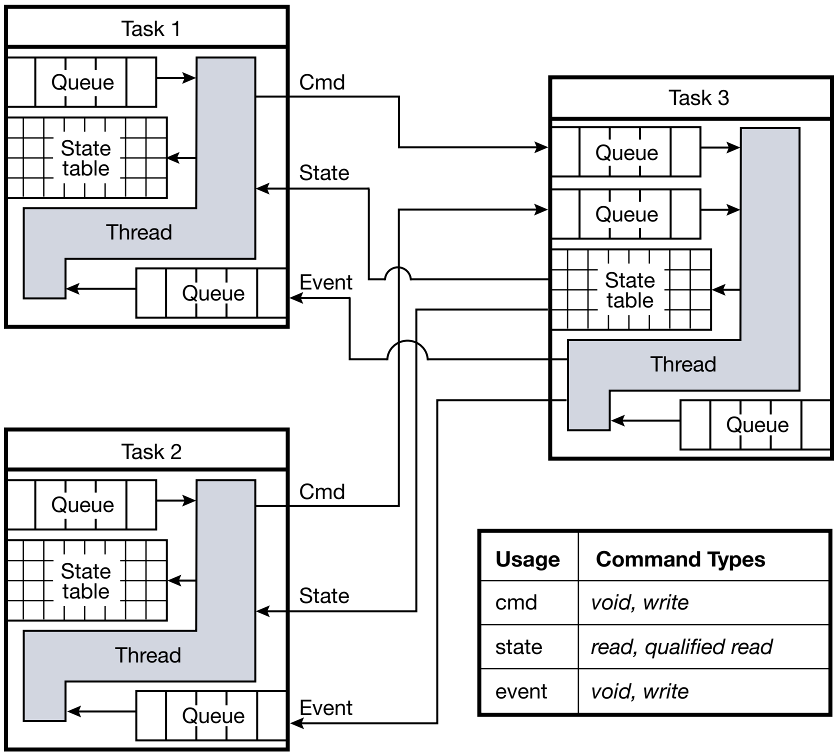
From the outset, SAW and *cisst* have been developed by using a well-defined process and set of tools (see Fig. 1). SAW uses the CppUnit and PyUnit testing frameworks to implement an automated nightly test suite, which consists of more than 1100 regression tests. This nightly test suite can find syntax errors or (sequential) program logic errors, but it generally cannot find errors in concurrent execution (e.g., timing errors, mutual exclusion problems, etc.). This is problematic because the eventual goal of many research projects is to translate the technology into the operating room for clinical testing.

Therefore, our approach is to apply recent advances in FM to reason about the correctness of these concurrent data-exchange mechanisms, which are the most difficult to verify using standard test suites (as is well known, "timing" errors can lay dormant in code for a long time until triggered). This approach complements other validation activities, such as design reviews, code walk-through, and regression testing.

PROVING CORRECTNESS FOR CONCURRENT SOFTWARE IN SURGICAL ROBOTS

The SAW has a concurrent architecture, and the communication mechanism that it uses to send data among the concurrent threads (shown in Fig. 2) uses no locks or blocking. Concurrent systems often have functionality that is not obvious; the main reason for this is that different threads of execution can interleave in unexpected patterns. For these systems, testing is unreliable when the goal is exhaustively finding bugs because it cannot adequately cover the state space, and software in a concurrent environment usually does not have a way of controlling the interleaving during execution, making the exact conditions of tests unrepeatable. For this reason, we chose to focus on the concurrent aspects of the system, to help eliminate bugs and nonobvious behavior.

We chose to analyze a core algorithm that mediates concurrent data exchange among threads and that uses a data structure called the *state table*. It enables one thread (the writer thread) to maintain and update data critical to the robot's understanding of its state, such as the position and velocity of its joints in space, while



Usage	Command Types
cmd	*void, write*
state	*read, qualified read*
event	*void, write*

Figure 2. Diagram of data structures used in communication channels among different threads in the SAW.

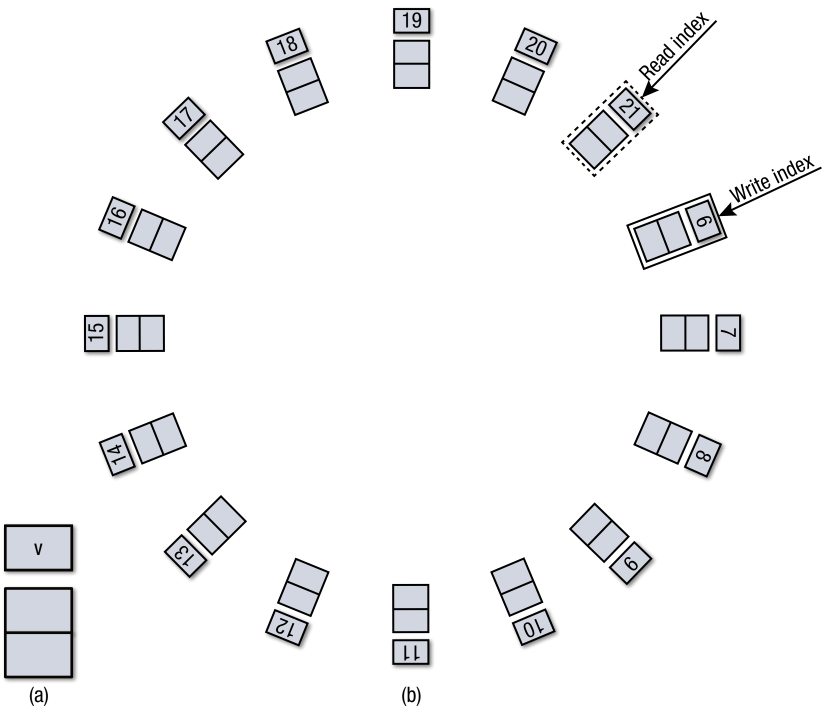


Figure 3. (a) A copy of the state vector. (b) The state table.

simultaneously allowing many other threads (the reader threads) to read uncorrupted, fresh values of these data for their own use. A single snapshot of the robot's state is called a state vector. The state table is a circular buffer, shown in Fig. 3a, with elements, as in Fig. 3b, that each contain a complete copy of the state vector. The state table algorithm maintains read and write indices, each referring to a single slot, shown by dashed and solid outlines, respectively.

An HLRG model of a concurrent system is made of source code. The code for this component of the library is written in C++ and is approximately 2000 lines long. We created a model of the code by computing a union of backward program slices whose slicing criteria are the different variables used in the state table's data structures. We inlined the various functions in the call tree and converted the resulting code to a simplified subset of C. The result is collected into two functions, which are shown in Fig. 4; one represents the relevant code executed repeatedly by the writer thread, and the other is the code that is executed by a reader thread every time it wants to read the system's state.

Examining the writer thread's algorithm closely, we find that the writer updates a slot by writing a fresh state vector into the write-indexed slot, advances the write index clockwise, updates the version number of the new write-index slot, and then advances the read index clockwise. There is only a single writer in the system, and the only function of the writer is to continually write fresh state-vector data into this buffer, filling it with data in a clockwise direction.

Fresh state-vector data are written in a slot each time the write index progresses around the buffer; therefore, to be uncorrupted, a single read needs to happen before the writer thread returns. By examining the algorithm, we can convince ourselves that the data in a slot may not change before their version numbers are updated.

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

Figure 4. Model of code.

To detect when corrupted data have been read, the reader thread compares version numbers before and after the read; if the version numbers do not match, the read is assumed to have been corrupted and must be tried again. If they do match, we assume that the read was uncorrupted.

Modeling Concurrent Software Behavior

We need a language to describe the behavior of concurrent software, without reference to its specific implementation; to do this, HLRG uses logical statements, or predicates. A predicate is a precise, logical statement about a snapshot of the system's state, which can be satisfied or not (true or false) at any given moment in the system's evolution. HLRG's predicates are made up of an innovative combination of operators from first-order logic, separation logic,[5,6] and modal operators borrowed from temporal logics. Within HLRG, predicates are not confined to the current state of the system but refer to a vector of system states, called a trace, that represents the history of the system's evolution over time. Table 1 shows the different operators available in the predicate and provides a brief description of the function and meaning of each.

Using these predicates, we can describe the effects of a program on memory, without reference to its details. For example, we can use the separating conjunction and point-to operator to assert, "At this instant, the program has allocated only two memory locations at addresses 100 and 104, which contain the values 23 and 36, respectively, and the rest of the virtual address space is unmapped," by writing $(100 \mapsto 23) * (104 \mapsto 36)$. Another example might be to use the temporal operators to assert, "In the past, address 100 contained some value, but at some subsequent point in time, it contained value 23 at the same time that address 200 contained the original value," by writing $\exists X.(100 \rightsquigarrow X) \blacktriangleright ((100 \rightsquigarrow 23) * (200 \rightsquigarrow X))$.

Reasoning About Concurrent Software

Our goal will be to ensure that when a reader thread completes a read of the state vector, it can accurately distinguish between a successful read that faithfully copied the values that were written and an unsuccessful read, where the values may be corrupted. In the latter case, the reader can try again.

First, we create an invariant we call I to describe what memory is allocated in our program. Some programs allocate and deallocate memory, so this can be complicated and time dependent in general, but in the case of this program, it is simple. Shared state in our case consists of the read and write indices, the state vector elements, and their version numbers.

Table 1. Different operators available in HLRG to express software behavior

$a \wedge b$	logical and
$a \vee b$	logical or
$\neg a$	negation
$\exists x P(x)$	existential quantification
$\forall x P(y)$	universal quantification
`addr` $\mapsto$ val	assertion of the existence of memory at the given address, and that its contents are value, and that there is no other memory mapped
`addr` $\rightsquigarrow$ val	same as above, but there may be other memory mapped whose location and contents are left unspecified (this is called an imprecise binding assertion)
$a * b$	separating conjunction is true when both a and b are true, and each are satisfied by memory at each step in the trace that does not overlap or coincide with the other at that point
$a \triangleright b$	a held at some point in the past, and b has held at every step since
$a \blacktriangleright b$	a held at some point in the past, and b held at some point subsequently
$a \ltimes b$	a held one step ago, and b holds now
$\boxminus a$	a holds at every step in the trace
$\diamondminus a$	a held at some point in the past

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

Underscores mean there is some value there, but we do not specify or need to know what it is. By doing this, we describe the shape of memory (i.e., what is allocated) without being specific about the values that memory contains. $\circledast$ is to $*$ as Σ is to +, representing a prefix version of the separating conjunction used to conjoin a series of terms. The assertion of this invariant is implicit in every reasoning step we take. That is to say, every predicate P that we write about this system implicitly means $P \wedge I$; I is so ubiquitous that we do not write it.

To reason about how these threads interact, HLRG uses a technique called rely/guarantee reasoning. In a sequential program, the only alterations to memory are from the instructions themselves; each instruction makes certain deterministic changes to the computer's state, according to its semantics. In a concurrent program that is being analyzed by using HLRG, the program's state changes with every instruction that is executed, but additional changes to state can occur in between instructions because of the actions of other concurrent threads in memory. To use HLRG, we need a way to express and summarize the effects that other threads can have on memory during the execution of the thread being analyzed.

To model the effects of other threads, HLRG uses the same predicate language it uses to describe behavior. However, when using these predicates to model the effects of other threads, the predicate represents an assertion that something is true during a particular time step, rather than a logical statement that may or may not be satisfied. This subtle difference allows predicates to be used to model atomic state transitions in the logic.

For the surgical robot, an executing reader thread does not know the state of the writer thread, so we need to summarize what atomic changes the writer can make to the computer's state in between instructions. This description of behavior is necessary because some of the reasoning rules require it.

Lines 6 and 7 in Fig. 4 are one atomic action from the perspective of shared state; this action updates the write index, `writeindex`, to point to the next element in the buffer. A description of this atomic action can be written as an assertive predicate:

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

This also asserts that the most recent step must follow the `UpdData` step, with intervening steps that do not change shared state, `Id`. We use the temporal operator $\triangleright$ to enforce this sequencing. This predicate, along with all of the others that describe atomic steps, has an `Id` connected to it with a separating conjunction. This assertion says that there is other memory that is disjoint from the memory explicitly referenced in the rest of the expression and that it has not changed between the last two state transitions in the thread. This makes it so we do not have to write imprecise assertions (i.e., using $\rightsquigarrow$) in these atomic transitions. Other shared memory is allowed (accounted for by `Id`) and is asserted to remain unchanged.

Lines 8 and 9, 10, and 4 and 5 also constitute atomic blocks, and follow similarly:

$$\begin{aligned}
\mathsf{UpdVer} &\stackrel{\text{def}}{=} \mathsf{Id} * ((\mathsf{UpdWrite} \triangleright \mathsf{Id}) \wedge \exists X, X', V, V'. \\
&\quad\ \texttt{writeindex} \mapsto X * \texttt{version}{+}X' \mapsto V' \ltimes \\
&\quad\ (\texttt{writeindex} \mapsto X * \texttt{version}{+}X \mapsto V'{+}1 * \\
&\quad\ \texttt{version}{+}X' \mapsto V') \wedge X = (X'+1)\mathsf{mod}\ H) \\[1ex]
\mathsf{UpdRead} &\stackrel{\text{def}}{=} \mathsf{Id} * ((\mathsf{UpdVer} \triangleright \mathsf{Id}) \wedge \exists X, Y.\texttt{writeindex} \mapsto Y \ltimes \\
&\quad\ \texttt{readindex} \mapsto X * \texttt{writeindex} \mapsto Y) \wedge \\
&\quad\ Y = (X+1)\mathsf{mod}\ H) \\[1ex]
\mathsf{UpdData} &\stackrel{\text{def}}{=} ((\mathsf{UpdData} \vee \mathsf{UpdRead}) \triangleright \mathsf{Id}) \wedge \textstyle\bigvee_{j\in[0,\ldots,N-1]} \exists X.(\mathsf{Vector}(X)(j) * \\
&\quad\ \texttt{writeindex} \mapsto X \ltimes \mathsf{Vector}(X)(j) * \texttt{writeindex} \mapsto X) * \mathsf{Id}
\end{aligned}$$

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.

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

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

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

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 execute a step in the `Read` function. Furthermore, ($I \ltimes I$) tells us that the invariant that describes the domain of the program does not change from step to step (i.e., no memory is mapped or unmapped during this program). Via this process, we have described the effect of `Write` on shared state as well as its interaction with other concurrent processes.

Proving Data Integrity

To prove data integrity, we began with a predicate of **true** as a precondition to `Read` and used the sound reasoning rules (known as inference rules) associated with HLRG to move that predicate forward through the program. As it moves past instructions and the gaps between instructions where concurrent threads can operate, the predicate changes form.

Starting at lines 15 and 16, which are the first lines that have an effect on the program's state, our predicate is referenced to the point right before line 15, and **true** means that the predicate is satisfied by any possible configuration of memory.

$$\{\mathsf{true}\}$$

```
15   rd = readindex;
```

The first reasoning step is to apply inference rules that describe how executing an assignment transforms a predicate that holds before the assignment into a predicate that holds after the assignment. This allows us to write down the appropriate predicate after line 15:

```
15   rd = readindex;
```

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

This says "Immediately after the atomic step described by the assignment on line 15, the local variable `rd` has the same value as the read index." Next, we have to consider the gap between instructions 15 and 16 because the writer thread can change shared state. This is where we use G because the rules for reasoning require a predicate that describes how the other threads change state. First, we know that at each step, $\mathcal{M} \overset{\text{def}}{=} \boxminus(R \vee G)$, which means our state is altered according to either *R* or G. Because *R* does not change the state, G describes the different atomic changes that can affect our shared memory. A finite number of these can occur before line 16 executes. These atomic steps include modification of the read index because it is shared state. Propagating the predicate past the gap, we find:

```
15   rd = readindex;
```

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

The difference is that now we have a $\diamondminus$ operator. This says, "At some point in the past, the read index had some value, and now `rd` has the same value as it had then." Notice that we cannot say much about the value that the read index has right now because it may have changed during the gap between lines 15 and 16 as a result of the operation of other concurrent threads. We also don't know exactly how far in the past its value matched what is currently in `rd`, only that it did at some time. Notice that `rd` is not shared state but is local to this function, so it will not change unpredictably because of the action of other concurrent threads. Now we can use the predicate we have so far and apply it to the point immediately before line 16, which is also an assignment:

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

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

We take similar steps with the predicate, propagating it past line 16 and past the gap between 16 and 17.

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

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

Here we used that $p \blacktriangleright q \overset{\text{def}}{=} \Diamond(\Diamond p \wedge q)$ to simplify the resulting predicate. The result says, "At some point in the past, the read index had some value X, and sometime after it, the Xth element of the version array had some value Y, and the present local variables `curTic1` and `rd` contain values Y and X, respectively."

It turns out that the knowledge that the read index once held X is not useful to us during the rest of this proof, and so we can use another reasoning rule to "forget" about this and drop this term, to make it easier to read.

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

$$\{\exists XY.\Diamond\, \texttt{version+}X \rightsquigarrow Y \wedge (\texttt{curTic1} = Y) \wedge (\texttt{rd} = X)\}$$

As we can see, as we move the predicate forward through the program, it changes according to the semantics of the different instructions as they are executed as well as the operation of concurrent threads in the gaps between instructions. There are many choices to be made about which reasoning steps to take and what sort of information needs to be expressed in the predicate at each step.

We seek to guarantee that when the `if` statement takes the `return 1;` branch, the postcondition of the computation of the branching condition guarantees that `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 and D represents the values in the memory array containing the state vector we just read.

This implies not just that `Read` read data were constant during the copy but also 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 because 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`.

Going through this process is straightforward, once one proves the following, which we call the stable data lemma:

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

This is an invariant tied to our `Read` algorithm, which says: If, at some time in the past, we looked at the value of `version+`h and then looked at the state vector in slot h, `Vector`(h), and `version+`h in the present matches what we saw the first time, then the value of `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 `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 corrupted 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 when the write index references a slot 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 our being able to detect it. We cannot guarantee that this situation never happens because the change of version happens separately from the update of index-writer.

Fixing a Potential Data Integrity Bug

The bug we found is a subtle one that requires a particular timing of the interactions among reader and writer threads. In every situation that we can think of that causes the problem above, the problem occurs when the initial version check and the read occur during the period when the write index is referencing the element being read; consequently, the state in between the initial version check and the read must occur when the element being read is referenced by the write index. 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.

To guarantee that we solved this problem, we revised the model of the `Read` algorithm, shown in Fig. 5, and modified the statement of our lemma to reflect the changes we made:

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

We call this the improved stable data lemma, and we are able to prove it using induction as we initially planned. The completed proof of the improved stable data lemma is given in the *Appendix*, along with a number of other lemmas.

```
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.

Once the stable data lemma has been proven, we can complete the proof of the read data integrity property that we seek. After line 19, our predicate looks somewhat intricate but gives a somewhat detailed picture of data structures in our program:

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

The predicate at this point is made of two terms separated by an "or." The first term describes the case where the conditions that we are proposing to use to identify an uncorrupted read have not been met, and the second describes the case where they have. The stable data lemma is in the form of an implication, and the second term of the predicate asserts the necessary conditions to satisfy the implicant (left side) of the implication; consequently, we can conclude the right side (i.e., $D = D'$) and add that term to the right disjunct of the overall predicate. Using induction, we can transform the second disjunctive term into an assertion about the continuous stability of the data in our state vector (i.e., that the data could not have changed during the read):

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

Propagating this predicate past the loop in lines 20 and 21 adds the terms `data` = D' and `data` = D, respectively, to each of the disjuncts in the predicate. Further propagating the predicate into the "if" statement pares off the first disjunctive term. We can examine the remaining predicate and find that if the execution reaches that point, the state vector was unchanging during the read, and the values are accurately reflected in the local variable `data`. This proves the data integrity property we seek: the read function can accurately discern cases where the data are uncorrupted. The completed proof of read data integrity is given in the *Appendix*.

CONCLUSION

The most promising strategy for creating a zero-defect software system is not to layer on additional functionality but to continue developing the mathematical rigor of models and the reasoning we use about the software we are designing. We aim to be able to mathematically prove that our system will not behave unexpectedly under any input conditions.

In this article, we have demonstrated the certification of a single, well-scoped component of a larger software system to be zero defect with respect to a single property: we can guarantee that successful reads are never corrupted by the data-exchange mechanism of the surgical robot library. We chose this component and the property because it is critical to the system; if the data-exchange mechanism corrupted state vector data, the robot would not know where it was operating—a potentially catastrophic condition for a patient undergoing surgery. This decision reduces the possible failure modes of the system.

There are many other things that can go wrong with the system, and we have a long way to go before we can declare the whole system to be zero defect. We have made no guarantees about other components, which could fail and affect this component. Even this component could cause problems because we have only guaranteed one property. For example, we would also want to prove that the data we have read are recent (fresh) and that the reader cannot be starved by the system; we will need to do more work to accomplish this.

This is an exciting time to be in the field of software engineering because FM is beginning to come to a maturity level where it can be used on practical problems. However, there are many problems still to be solved. Scaling our proofs to large systems is still very difficult to do, even with the aid of computers. We have not figured out how to make these techniques easy for someone to use. Often, they are more pedantic than they need to be, and their reasoning steps are too small for the leaps we make in our heads. We need to automate all of the simple reasoning steps and leave the tricky bits to be proved by the engineer, and we need to develop easy-to-use interfaces for the software engineer.

The FM community is in the process of developing the domain-specific logics that can be applied to various levels of abstraction in software systems. The software engineering community can help develop these techniques by experimenting with their application in real systems. This could provide immediate benefit to software engineering projects in the form of proofs of correctness about some of the critical algorithms. More importantly, such experimental applications could also identify problems with the application of these logics that help lead to a vision of what the tools should look like in the future.

REFERENCES

[1]Kazanzides, P., Kouskoulas, Y., Deguet, A., and Shao, Z., “Proving the Correctness of Concurrent Robot Software,” in *Proc. IEEE International Conf. on Robotics and Automation*, St. Paul, MN, pp. 4718–4723 (2012).

[2]Kouskoulas, Y., Ming, F., Shao, Z., and Kazanzides, P., *Certifying the Concurrent State Table Implementation in a Surgical Robotic System (Extended Version)*, Yale University Technical Report, http://flint.cs.yale.edu/flint/publications/statevec-tr.pdf (2011).

[3]Fu, M., Li, Y., Feng, X., Shao, Z., and Zhang, Y., “Reasoning About Optimistic Concurrency Using a Program Logic for History,” *CONCUR 2010 – Concurrency Theory, Volume 6269 of Lecture Notes in Computer Science*, P. Gastin and F. Laroussinie (eds.), Springer, Berlin/Heidelberg, pp. 388–402 (2010).

[4]Kazanzides, P., DiMaio, S., Deguet, A., Vagvolgyi, B., Balicki, M., et al., “The Surgical Assistant Workstation (SAW) in Minimally-Invasive Surgery and Microsurgery,” in *Proc. MICCAI Workshop on Systems and Architecture for Computer Assisted Interventions*, Morrisville, NC, *Midas J.*, pp. 1–9, http://hdl.handle.net/10380/3179 (Jun 2010).

[5]Ishtiaq, S. S., and O’Hearn, P. W., “BI as an Assertion Language for Mutable Data Structures,” in *Proc. 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, New York, pp. 14–26 (2001).

[6]Reynolds, J. C., “Separation Logic: A Logic for Shared Mutable Data Structures,” in *Proc. 17th Annual IEEE Symp. on Logic in Computer Science*, Copenhagen, Denmark, p. 55 (2002).

APPENDIX.

DETAILS OF PROOF OF IMPROVED STABLE DATA LEMMA

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.

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 preceding state may have occurred more than once.

We prove this lemma by inspection of the predicates that we used to describe atomic transitions, with attention to the portion of the predicates that enforce ordering, e.g.,

$$\mathsf{UpdWrite} \quad \overset{\text{def}}{=} \quad \cdots(\mathsf{UpdData} \rhd \mathsf{Id})\cdots$$

Because the structure of the predicates does not contain any other disjunctive terms, the linear ordering above follows.

Monotonic Version Lemma

A given version number is monotonic.

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

The only step in a trace that affects $\texttt{Ticks+}h$ is the $\texttt{UpdVer}$ 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 $X \in \mathbb{N}, (X \leq X')$.

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 $X \leq X'$ and $X' \leq X''$. Substitute $X = X''$ in the second term, and we find $X \leq X' \leq X$. Consequently, $(X = X')$.

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

Assume the implicant, that the first term is satisfied by a state in the trace T, s_i, and that the second term is satisfied by the current state in the $T.last$. For each state s' in between s_i and $T.last$, we can measure $(\texttt{version+X})$, and by application of the constant version lemma, we can show that $(\texttt{version+}X \rightsquigarrow Y)$ for all s'. States s_i and $T.last$ satisfy $(\texttt{version+}X \rightsquigarrow Y)$, by the assumption of the implicant. Consequently, $(\texttt{version+}X \rightsquigarrow Y)$ held continuously, from the first time we looked at $\texttt{version}$ until the present moment.

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 by 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 that 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:

- `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.
- `UpdVer`—Changes $\texttt{version+}h$ when $\texttt{writeindex} \rightsquigarrow h$. It could falsify the implicant, but this could only preserve the truth of the predicate.
- `UpdRead`—Does not affect any terms in the predicate.
- `UpdData`—This step could change D' when $\texttt{writeindex} \rightsquigarrow h$ and could falsify the implicand. When this occurs, it must have been preceded by the following state-changing steps, in the following order: `UpdWrite` :: `UpdVer` :: `UpdRead` :: `UpdData`+. 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 was 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 E and preserving the truth of the predicate because `UpdVer` never repeats version numbers, and no other predicate changes the version binding.
  If term A was 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 $\texttt{writeindex} \rightsquigarrow h$, 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.

DETAILS OF PROOF OF READ DATA INTEGRITY

This appendix completes the proof of data integrity property, following the strategy described in the *Proving Data Integrity* section.

Figures 6 and 7 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 that 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 nonatomic reading of state (i.e., line 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 nonatomic nature.

The precondition for the successful completion of the program shown in Fig. 7 contains `data` = *D*, and at some range of time during this program, `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).

$\{\text{true}\}$

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

$\{\exists X. \Diamond\ \texttt{readindex} \rightsquigarrow X \wedge (\texttt{rd} = X)\}$

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

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

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

```
17   wr = writeindex;
```

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

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

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

Figure 6. Beginning the proof of the data integrity property. This is 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.

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

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

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

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

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

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

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

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

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

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

Figure 7. Using the series of transformations described in Figs. 8 and 9, we can reason about the propagation of the predicate past the nonatomic read in line 20 and 21. At the completion of the read, `data` = *D'*, and some subset of these moments in the period between the first version check and the present is used to read the state vector into the local variable `data`. If the value of `Vector`(*X*) was constant, then we can conclude that at the end of the read, *D* = *D'*.

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

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

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

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.

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

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

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

$$
\begin{array}{l}
\{\exists X X' Y. \\
Q_1 \vee Q_2\}
\end{array}
$$

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 check 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 $\texttt{Vector}(X) \leadsto D$ term in this predicate and the present moment. We can then apply the stable data lemma to show that since $\texttt{version+}X \leadsto Y$, $\texttt{Vector}(X) \leadsto D$ 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.

The Authors

Yanni Kouskoulas is a group chief scientist at APL. His research interests are focused on formal methods and theorem proving and their application to help develop zero-defect software for a variety of practical cyber-physical systems. **Ming Fu** is a postdoctoral collaborator at the USTC–Yale Joint Research Center for High-Confidence Software in Suzhou, China. He published the paper describing the HLRG program logic used during this reasearch. He also helped apply it to the surgical robot system. **Zhong Shao** is a professor of computer science at Yale University. He leads the FLINT group and is interested in building novel certified system software, programming language design, compiler development, formalsemantics and logics, and proof engineering and automation, among other topics. **Peter Kazanzides** is an associate research professor of computer science at The Johns Hopkins University. His research interests include computer-integrated surgery and systems engineering to enable deployment in the real world. While working at Integrated Surgical Systems, he was responsible for the design and implementation of the ROBODOC system, which has been used for more than 20,000 hip and knee replacement surgeries. He helped develop the surgical robot control algorithm that was the subject of analysis during this work. For further information on the work reported here, contact Yanni Kouskoulas. His e-mail address is yanni.kouskoulas@jhuapl.edu.