2012 IEEE International Conference on Robotics and Automation
RiverCentre, Saint Paul, Minnesota, USA
May 14-18, 2012

Proving the Correctness of Concurrent Robot Software

Peter Kazanzides Yanni Kouskoulas Anton Deguet Zhong Shao

Abstract—

Component-based software has been proposed as a methodology for improving software reuse and has increasingly been adopted by robot software developers. At the same time, robot systems typically have real-time performance requirements and performance gains can often be obtained by multi-threading. It is challenging, however, to create correct multi-threaded software, especially when standard mutual exclusion primitives, such as mutexes and semaphores, are eschewed in favor of more efficient, lock-free mechanisms. It is even more difficult to find these errors, as they can remain dormant for years until triggered by just the "right" conditions. Our approach, therefore, is to apply Formal Methods to reason about the correctness of these mechanisms. As a first step, we adopted a recently-developed program logic called History for Local Rely/Guarantee (HLRG) and applied it to prove the correctness (after first finding and fixing an error) of one such mechanism in the open source *cisst* software package. This strategy is not specific to *cisst* and can be applied to other packages.

I. INTRODUCTION

The complexity of robot systems is increasing as they become integrated with other devices, such as cameras and imaging devices. Component-based software engineering has been proposed as a paradigm for dealing with complex systems[1], and this approach has been adopted by a number of robot software frameworks [2]. For example, Player [3], ORCA [4], and the more recent ROS [5] all utilize a component-based approach, where the components consist of separate processes (on one or more computers) that communicate via a middleware layer. These frameworks, however, do not directly address the real-time requirements of low-level robot control. This has led to the integration of packages that provide real-time computing within popular frameworks such as ROS. For example, both OROCOS [6] and our own *cisst* package [7] support real-time processing and "bridges" have been created between these packages and the widely-adopted ROS package [8], [9].

Software packages for real-time processing obviously must support one or more real-time operating systems, but they also tend to use multi-threading to take advantage of multi-core computers. This is true of both OROCOS and *cisst*, which use lock-free methods for efficient data exchange between the different threads. The question, then, is how to validate the correct operation of multi-threaded robot control software.

For some system components, testing techniques (e.g. [10]) are an effective way to reduce defects in operation. The *cisst* software uses the CDash package and the CppUnit and PyUnit testing frameworks to implement an automated nightly test suite. Several computers, with different operating systems and compilers, perform nightly tests and report the results to a central web page. Unfortunately, although this test suite can find syntax errors or (sequential) program logic errors, it generally cannot find errors in concurrent execution (as is well known, "timing" errors can lay dormant in code for a long time until triggered).

Our approach, therefore, is to apply recent advances in formal methods to reason about the correctness of the concurrent data exchange mechanisms, which are the most difficult to verify using standard test suites. This complements other validation activities, such as design reviews, code walk-through, and regression testing. One limitation of our approach is that it assumes that programmers actually use the concurrent data exchange mechanisms provided by the software package; for example, it cannot check for unsafe sharing of data between threads (e.g., using a global variable).

This paper uses the *cisst* software package (available at trac.lcsr.jhu.edu/cisst) as a test case, but the concepts can be applied to any package that provides mechanisms for data exchange in concurrent multi-threaded programs.

P. Kazanzides and A. Deguet are with the Dept. of Computer Science, Johns Hopkins University; Y. Kouskoulas is with the Applied Physics Lab, Johns Hopkins University; Z. Shao is with the Dept. of Computer Science, Yale University

II. OVERVIEW OF *cisst* PACKAGE

The *cisst* package is a collection of component-based open-source C++ software libraries developed to support our research in computer-assisted intervention systems. It originally supported a multi-threaded (single-process) environment [7] and was later extended to networked (multi-process) configurations [11]. This section describes the thread-safe data exchange mechanisms that exist within the *cisst* package.

One challenge with multi-threaded programming is the management of access to shared resources (mutual exclusion), which is especially problematic because all threads share a common address space. Modern operating systems provide several mechanisms for controlling access to shared resources, such as semaphores, mutexes, and critical sections. Once one thread has locked a resource, other threads that attempt to lock it are either blocked until the resource is unlocked or allowed to continue without accessing the resource. This behavior is undesirable for high-frequency real-time threads, such as those used for robot control. In addition, improper use can lead to deadlocks, which are conditions where where each thread is waiting for a mutex or semaphore to be unlocked. It is possible to use a tool, such as D-Finder in GenoM/BIP [12], to check for deadlock. Our approach, however, is to use lock-free data structures for inter-thread data exchange. Specifically, the *cisst* library uses state tables and message queues, as shown in Fig. 1.

Authorized licensed use limited to: Yale University. Downloaded on May 12,2020 at 15:23:13 UTC from IEEE Xplore. Restrictions apply.

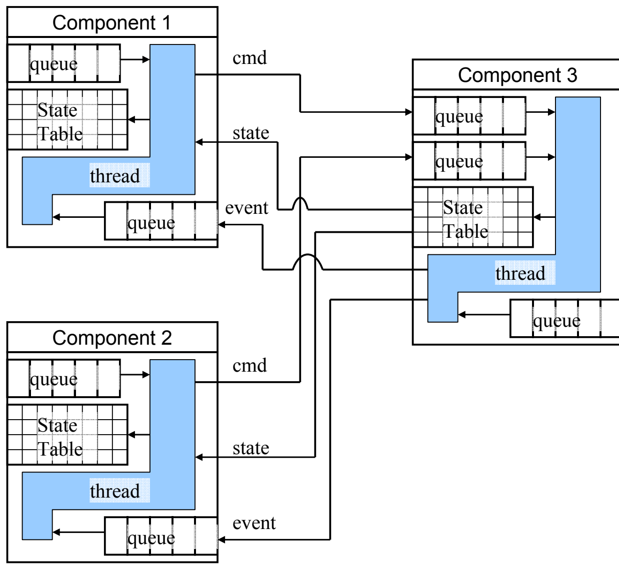

Fig. 1: Component interconnection in *cisst*

A state table is a time-indexed circular (ring) buffer, which has a single writer (the owning component) but potentially many readers (client components). The important component state (typically, some or all of the class data members) are automatically archived in the state table at the end of each execution cycle. The readers use a time index that is one less (older) than the write index. Thus, a reader either (1) obtains fresh data (the expected case, with a sufficiently large ring buffer) or (2) is informed that the data has been corrupted (i.e., the ring buffer "wrapped around" and overwrote the data). The time indexing also provides a snapshot of the history of the real-time system and can be used for data collection as well as for debugging (i.e., to provide a "flight data recorder" functionality). The state table is similar to publish/subscribe systems (e.g., ROS topics), except that information is published locally and subscribers must pay the cost of retrieving the data. On the other hand, a high-frequency publisher does not overwhelm the network middleware.

A message queue is also a ring buffer, but is restricted to a single writer (the client component) and a single reader (the server component that owns the provided interface that contains the message queue). This can also be implemented in a lock-free manner, with the assumption that pointer updates are atomic. As shown in Fig. 1, commands (cmd) sent from the client component (1 or 2) to the server component (3) are queued. To ensure a single writer, yet still allow multiple clients, a new buffer is automatically allocated for each client component when the connection is established. Essentially, we create a new provided interface "instance" for each connected client's required interface.

III. Overview of Formal Methods

In the context of this research, formal methods are mathematically rigorous approaches to reasoning carefully about a system, to ensure that it operates as we intend. Broadly, there are two major categories of formal approaches: deductive verification approaches, and model-checking approaches. Each approach offers a method of describing a system, a logic used to express the specification of a system, and an approach to proving (or disproving) that a system matches a particular specification.

Model-checking approaches [13], [14] allow their users to describe systems in terms of the semantics of finite-state automata interacting concurrently through communication channels. Model-checking approaches typically use linear temporal logic (LTL) as their specification language, and have the advantage of being fully automated, because their proof strategy is a brute-force search through the space of possible state transitions. This search sometimes follows a rely-guarantee formalism [15] which allows reasoning about concurrency. In this formalism, each sequential component provides a guarantee to the others about the state transitions it may take, and from these, each component constructs a description of the environment on which it can rely, based on the guarantees of the other components. In between each state transition, zero or more atomic steps conforming to the environment may occur. Model-checking adds optimization to its proof search, but still has the disadvantage that if the system described has too large of a state space, the proofs may not complete in any reasonable time frame.

Deductive verification approaches allow more expressive languages to be used to describe systems, because their proof strategy depends on the application of inference rules about the system to prove different system properties. These approaches can be traced back to Hoare Logic [16], which provides a set of inference rules that allow us to reason about the effects that programs have on computer state. The rules enable one to take a precondition (a logical predicate about the state of the computer, referenced to the beginning of the program) and propagate it forward through a program to its end, and guarantee that the resulting postcondition holds if the program terminates. Deductive approaches allow us to reason about systems with extremely large state spaces without using a brute-force approach.

Recently, the FLINT group at Yale University developed a program logic called History for Local Rely/Guarantee (HLRG) [17]. HLRG builds upon LRG [18] which combines separation logic [19], [20] to enable local reasoning, with rely-guarantee reasoning to make guarantees about multiple processes accessing the same data structures concurrently. HLRG enables the prover to write specifications using temporal operators to describe time-based properties of data structures, and to reason about concurrent programs using sound Hoare-style inference rules. To describe temporal properties, assertions in HLRG apply to sequences of historical system states which are collected in a vector called a trace. To our knowledge, this is one of the most practical and powerful approaches that facilitates and simplifies reasoning about concurrency at the program implementation level.

IV. Application of formal methods to *cisst*

We have chosen to analyze the *cisst* state table (see Section II), which is a circular array of state vectors, indexed by time,

Authorized licensed use limited to: Yale University. Downloaded on May 12,2020 at 15:23:13 UTC from IEEE Xplore. Restrictions apply.

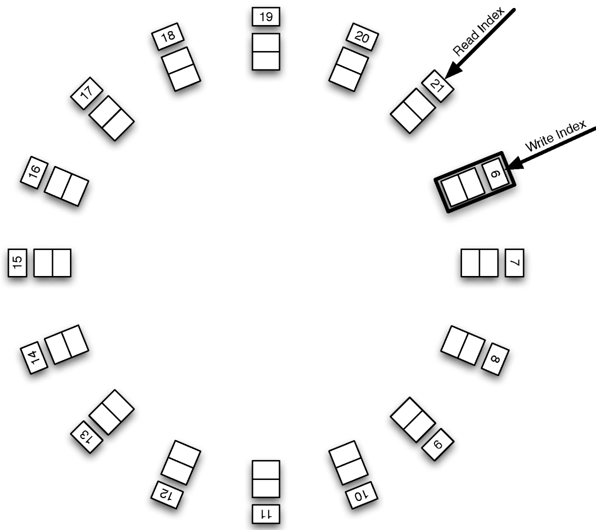

Fig. 2: The circular buffer used to store copies of the state vector in memory. The read- and write-indices each independently refer to a single slot in the circular buffer. The read-index is indicated by a dashed outline surrounding the slot to which it refers, and the write-index is indicated by a solid outline around the slot to which it refers.

as illustrated in Fig. 2. We wish to guarantee the following two state table requirements:

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 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.

A. State table algorithm

To do its job, the state table maintains space for storage of H copies of the state vector, each with an identifying version number (the top storage location) and a lower array 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. 2, 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 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.

B. Formal verification

The state table algorithm uses optimistic concurrency, a low-latency lock-free approach to interacting in a shared memory environment. To our knowledge there are very few logics that are available that can be applied to this algorithm to model the properties in which we are interested, and prove their correctness at the program implementation level. Aside from History for Local Rely/Guarantee (HLRG) [17] (see Section III), [21] is the only other logic we are aware of that is designed to address this problem.

To guarantee that there are no defects with respect to the properties that we describe, we applied HLRG. HLRG allows us to precisely describe the software, and then apply sound inference rules to reason about it in a mathematically rigorous manner. This section summarizes the results of our verification; for further details, refer to [22], [23].

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 (a trace), where each element represents the system state at a particular step in its evolution. This allows our assertions to refer to the history of the system (the origin of the "H" in HLRG).

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. 3.

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 read- and write- indices, the state vector copies and their version numbers.

The next step is to write all of the atomic actions that are taken on shared state as predicates. These atomic actions are UpdWrite (lines 6-7 in Fig. 3) that update the write-index, UpdData (lines 4-5) to update the data (state vector), UpdVer(lines 8-9) to update the version, and UpdRead (line 10) to update the read-index. We present the HLRG notation for the UpdWrite atomic step to illustrate this process; explanation of the notation and details for the other atomic actions (predicates) can be found in [22], [23]. The UpdWrite

Authorized licensed use limited to: Yale University. Downloaded on May 12,2020 at 15:23:13 UTC from IEEE Xplore. Restrictions apply.

```
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 }
```

Fig. 3: Model of Code

atomic action updates the write-index, `writeindex`, to point to the next element in the buffer. We write a predicate that is satisfied with a trace that has just taken this step as follows:

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

This step must follow the $\mathsf{UpdData}$ step, with some number of intervening steps, all of which must be steps that do not change shared state, Id, so we use the temporal operator $\rhd$ to enforce this sequencing.

Once all predicates are defined, we can create rely and guarantee predicates describing the operation of the `Write` program.

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

The guarantee predicate G, is a guarantee about the behavior of the thread executing the `Write` function: it tells us how a step taken by that thread affects shared state. R assures us that the rest of the concurrent processes (namely the multitude of possible readers executing `Read`) have no effect on the state. $\mathcal{M}$ 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 does not 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.

C. *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. We use $\rightsquigarrow$ to be an imprecise binding assertion, i.e. $\mathsf{Vector}(X) \rightsquigarrow D$ means that the heap has at least the memory cell at address $\mathsf{Vector}(X)$ which contains value D, and may have more state as well.

This implies not just that the `Read` read data 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 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`.

D. *Proving the Stable Data Lemma*

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

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

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 at the state vector in slot h, $\mathsf{Vector}(h)$, and if the present value of $\texttt{version} + h$ matches what we saw the first time, then the value of the present $\mathsf{Vector}(h)$ is 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 lemma by inducting over the steps in a trace, we found that it was not true, 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

Authorized licensed use limited to: Yale University. Downloaded on May 12,2020 at 15:23:13 UTC from IEEE Xplore. Restrictions apply.

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.

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 is accomplished by inserting the following three lines after line 16 of the `Read` method in Fig. 3:

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

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

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

With this modification, the formal proof by induction succeeds and we can complete the proof of the read data integrity property that we seek.

E. 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, or the one directly before that. We want to make sure that `readindex` always points to the freshest slot at any point in time. We state this invariant as follows:

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

Because of the mechanics of the writer thread, we can only ensure that one of the two most recently written vectors available at the beginning of the read will be accessed. This is necessary because it is possible that the writer has finished writing but not yet marked its slot as "completed." We approach the proof by induction, inducting over the steps in a trace. We begin by proving that $i \Rightarrow i + 1$.

Assuming that we have a trace T that satisfies this lemma, we can show that any step taken produces a trace T' that also satisfies the lemma. To illustrate the process, we consider the UpdData step, which 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, which is acceptable to this predicate. 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 (Section IV-F. The first is that this UpdData step is the first that applies to this index in this cycle, and the history looks something like the following, with the previous UpdData applying to another index:

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

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.

The second possibility is that the state transition history looks like this:

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

In this case, we 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 because 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, including at the beginning of each read of the state vector.

F. 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 portions that enforce ordering:

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

The definition of UpdWrite says that this state transition must have been preceded by zero or more identity transitions, which were in turn preceded by an UpdData state transition. The only variation in the predictable, straight-line sequence of steps is the possible repetition of the 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.

Authorized licensed use limited to: Yale University. Downloaded on May 12,2020 at 15:23:13 UTC from IEEE Xplore. Restrictions apply.

V. Discussion

As a first step towards proving the correctness of concurrent robot software, we applied the HLRG program logic to one lock-free data exchange mechanism in *cisst* – the State Table, which is a time-indexed circular buffer that has a single writer (the owning component) but potentially many readers (client components). In this case, we manually reverse-engineered the existing C++ software, modeled it using HLRG, and then attempted to prove its correctness. One potential issue is that errors during the reverse-engineering process may result in models that do not accurately reflect the actual code. A model-based approach with a (validated) automatic code generation tool would be less error-prone.

Whether reverse-engineering code or using model-driven development, robotics engineers may be discouraged by the complexity of a language such as HLRG. A good compromise may be to rely on experts to develop models of the key concurrent data exchange mechanisms and prove their correctness; then, robotics engineers can use familiar programming constructs to achieve thread-safe data exchange.

An alternative solution is to discard the multi-threaded programming model and replace it with new mechanisms such as coordination languages [24]. If one considers computation and coordination to be distinct program activities, then it is possible to have one language for computation and another for coordination [25]. This approach would, however, require a rewrite of existing robotics software.

VI. Conclusions

This paper presented preliminary work in using formal methods to prove the correctness of concurrent (multi-threaded) robot software. Specifically, we used HLRG to prove correctness of the state table mechanism in the *cisst* package. 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, the system had no flaws in its design or implementation.

This study was motivated by the observation that standard software testing activities are suitable for detecting some types of errors (e.g., program logic), but typically fail to find errors due to the interleaving of concurrent threads. Fortunately, component-based software packages, such as *cisst*, are suitable candidates for formal methods because they provide a small set of inter-thread communication mechanisms. If we prove correctness of all concurrency mechanisms and if programmers rely on these mechanisms for data exchange between components, we can hope to create better robot software. Our process is not restricted to the *cisst* package or to surgical robots and can be applied to other software packages and application domains.

VII. Acknowledgments

We thank Russell H. Taylor and the many contributors to the *cisst* package. Ming Fu assisted with the application of HLRG. This work is supported in part by National Science Foundation EEC 9731748, EEC 0646678, MRI 0722943, CNS 0915888, CNS 0910670, and DARPA FA8750-10-2-0254.

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[2] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robotics and Automation Magazine*, pp. 84–96, Dec 2009.

[3] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proc. Intl. Conf. on Advanced Robotics (ICAR)*, 2003, pp. 317–323.

[4] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), Workshop on Robotic Standardization*, Dec 2006.

[5] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[6] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 2, Sep 2003, pp. 2766–2771.

[7] A. Kapoor, A. Deguet, and P. Kazanzides, "Software components and frameworks for medical robot control," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2006, pp. 3813–3818.

[8] Orocos ROS package. [Online]. Available: http://www.ros.org/wiki/kul-ros-pkg

[9] *cisst* ROS package. [Online]. Available: http://code.google.com/p/jhu-lcsr-ros-pkg

[10] K. Sen, "Concolic testing," in *Proc. 22nd IEEE/ACM intl. conf. on Automated Software Engineering (ASE)*, 2007, pp. 571–572.

[11] M. Y. Jung, A. Deguet, and P. Kazanzides, "A component-based architecture for flexible integration of robotic systems," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010, pp. 6107–6112.

[12] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan, "A verifiable and correct-by-construction controller for robot functional levels," *J. of Software Engin. for Robotics (JOSER)*, vol. 2, no. 1, pp. 1–19, Sep 2011.

[13] E. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science*, ser. LNCS, S. Ramesh and G. Sivakumar, Eds. Springer Berlin / Heidelberg, 1997, vol. 1346, pp. 54–56.

[14] S. Bäumler, M. Balser, A. Dunets, W. Reif, and J. Schmitt, "Verification of medical guidelines by model checking – a case study," in *Model Checking Software*, ser. LNCS, A. Valmari, Ed. Springer Berlin / Heidelberg, 2006, vol. 3925, pp. 219–233.

[15] T. A. Henzinger, M. Minea, and V. Prabhu, "Assume-guarantee reasoning for hierarchical hybrid systems," in *In: HSCC. Volume 2034 of LNCS*. Springer, 2001, pp. 275–290.

[16] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, Oct. 1969.

[17] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang, "Reasoning about optimistic concurrency using a program logic for history," in *CONCUR 2010 - Concurrency Theory*, ser. LNCS, P. Gastin and F. Laroussinie, Eds. Springer Berlin / Heidelberg, 2010, vol. 6269, pp. 388–402.

[18] X. Feng, "Local rely-guarantee reasoning," in *Proc. 36th ACM Symp. on Principles of Programming Languages*, 2009, pp. 315–327.

[19] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th Annual IEEE Symp. on Logic in Computer Science*, July 2002, pp. 55–74.

[20] P. W. O'Hearn, "Resources, concurrency and local reasoning," in *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, ser. LNCS, vol. 3170, 2004, pp. 49–67.

[21] V. Vafeiadis, "Modular fine-grained concurrency verification," Ph.D. dissertation, PhD thesis, University of Cambridge, 2007.

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

[23] ——, "Certifying the concurrent state table implementation in a surgical robotic system," in *3rd Joint Workshop On High Confidence Medical Devices, Software, and Systems (HCMDSS) and Medical Device Plug-and-Play Interoperability (MDPnP)*, Apr 2011.

[24] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[25] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, pp. 97–107, Feb 1992.

Authorized licensed use limited to: Yale University. Downloaded on May 12,2020 at 15:23:13 UTC from IEEE Xplore. Restrictions apply.