Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2*

Brian Kempa, Pei Zhang,
Phillip H. Jones, Joseph Zambreno, and Kristin Yvonne Rozier

Iowa State University, Ames IA 50014, USA
{bckempa, peizhang, phjones, zambreno, kyrozier}@iastate.edu
http://temporallogic.org/research/R2U2/

Abstract. Robonaut2 (R2) is a humanoid robot onboard the International Space Station (ISS), performing specialized tasks in collaboration with astronauts. After deployment, R2 developed an unexpected emergent behavior. R2's inability to distinguish between knee-joint faults (e.g., due to sensor drift versus violated environmental assumptions) began triggering mid-task, safety-preserving freezes-in-place in the confined space of the ISS, preventing further motion until a ground-control operator determines the root-cause and initiates proper corrective action. Runtime verification (RV) algorithms can efficiently disambiguate the temporal signatures of different faults in real-time. However, no previous RV engine can operate within the limited available resources and specialized platform constraints of R2's hardware architecture. An attempt to deploy the only runtime verification engine designed for embedded flight systems, R2U2, failed due to resource constraints. We present a significant redesign of the core R2U2 algorithms to adapt to severe resource and certification constraints and prove their correctness, time complexity, and space requirements. We further define optimizations enabled by our new algorithms and implement the new version of R2U2. We encode specifications describing real-life faults occurring onboard Robonaut2 using MLTL and detail our process of specification debugging, validation, and refinement. We deploy this new version of R2U2 on Robonaut2, demonstrating successful real-time fault disambiguation and mitigation triggering of R2's knee-joint faults without false positives.

Keywords: Online Runtime Verification · Steam-based Runtime Verification · Real-time Embedded Systems · Temporal Logic Specification

1 Introduction

As the demand for autonomous operation rises at an unprecedented pace, we must find ways for robotic systems to aid or replace human tasks safely. This requires embedding runtime checks for safe operation into specialized domain-specific platforms designed for utility and efficiency. Robonaut2 (R2) [9] is a humanoid robot capable of performing complex tasks on-board the International Space Station (ISS) while interacting safely with humans. [12] Even carefully designed, formally-verified cyber-physical systems experience unanticipated emergent behaviors deployment to complex, dynamic environments like the ISS. In

* Supported by NASA ECF NNX16AR57G and NSF CAREER Award CNS-1552934.

R2's case, position sensors within rotational joints can return faulty position data indistinguishable from a high-torque force to the current control system. Disambiguating between sensing errors and high-torque states would enable autonomous operation, rather than freezing for safety reasons and contacting Houston ground-control for help; choosing the incorrect mitigation action can have disastrous consequences. Autonomous operation demands the real-time reasoning and safety guarantees provided by runtime verification, on increasingly domain-specific hardware, including post-deployment.

This fault-disambiguation problem poses several challenges that previously prevented an effective solution. Runtime Verification (RV) could detect the faults, but R2 is already deployed on the ISS; no new resources will be launched to run an RV engine. Low-level joint control resides on a heavily-optimized Field Programmable Gate Array (FPGA) adjacent to the knee with limited remaining space. Consequently, the only available resources in which to implement a solution are tightly-constrained. RV needs to run in hardware in the remaining space on that critical FPGA with provable non-interference with the existing joint controller. The RV engine must be real-time, online, and stream-based to continuously evaluate faults throughout R2's operation. RV on R2 must be remotely configurable process; we cannot bring R2 back to Earth or requisition astronaut time to change the runtime observer specification. Given that systems on the ISS are frequently repurposed and operate in a continuously-changing environment, we need to be able to change RV observers without re-synthesizing hardware, and quickly adapt to updated conditions and requirements. Most RV tools are implemented in software, require significant resources and overhead, or have incompatible expression languages. R2 is running the Robot Operating System (ROS) [21] and some formal verification tools for ROS exist; however, none of these fit the requirements of the R2 project. Others have developed a generic approach to formally verify real-time properties of ROS-based applications [10], at design time, using timed automata and a model checker in an approach that cannot be scaled to R2's resource constraints. Similarly, ROSCoq extends the Coq theorem prover to enable reasoning about the cyber-physical behavior for developing certified ROS systems [8]. ROSRV [11] and Declarative Robot Safety (DeRoS) [1] integrate RV into ROS by automatically generating ROS nodes that monitor said properties during execution. But, they are software-based, limited to data published on the ROS message bus, and incur significant runtime overhead. `EgMon` eagerly checks for violations of specifications in a future-bounded, propositional metric temporal logic that avoids instrumentation of already-certified components [13]. Reading (previously logged) inputs over a via a CAN bus (similar to the ROS bus) `EgMon` detected safety violations for an autonomous research vehicle. But, `EgMon` is a software implementation that would not work in R2's architecture with significant overhead and with a high level of false positives that would be unacceptable. Formal verification of autonomous robot systems is a burgeoning research area; see [17] for a survey.

R2 requires a hardware-based solution with consideration for resource constraints; Table 1 summarizes four options. IoTA considers some resource con-

straints in implementing RV, but for software [6]. RVS is the only other mod-

Tool	P2V[16]	BusMOP[19]	HW-CBMC [18]	R2U2 [22]
Method	Automata synthesis			Formula decomposition
Type	Hard-coded			Programmable
Target	Software	COTS Peripheral	HW-SW Co-design	Sensor
Spec Logic	Past time only			Future/past time
Last Update	2007	2008	2017	2019

Table 1. Comparison of Hardware Monitor Tools.

ern hardware RV implementation; its limited expression language only produces monitors the internal behavior of the kernel and requires resynthesis to change monitored properties [24]. The Realizable Responsive Unobtrusive Unit [23] (R2U2) is the only RV tool that starts an encoding with the resource constraints and then optimizes the verification to reliably detect as many faults as possible, rather than, e.g., starting with runtime monitors and creating resource-consuming implementations. R2U2's online, stream-based, hardware (FPGA) implementation, provable unobtrusiveness, and ability to change monitors without resynthesis fit the R2 project. R2U2's compositional, hierarchical design and more flexible specification language made it most likely to fit in the space left over on R2's knee joint's FPGA. However, an intial trial proved that even R2U2's most optimized configuration would not fit; *no currently existing RV tools were capable of on-board, real-time verification of R2's knee joint fault.* In order to solve this practical problem, we must create a tool to bridge that gap. We use R2U2 as a base, design and prove correct new observer encoding algorithms suitable for R2, and develop optimizations until we are able to successfully deploy RV on the real Robonaut2 knee joint.

This paper contributes: (1) a significant revision of all asynchronous future-time MLTL monitor encodings of [22] with new proofs of correctness; (2) optimizations to online RV for operation under resource constraints using these encodings; (3) an implementation of these monitors with an empirical evaluation showing improvement in resource consumption; (4) specification design, debugging, validation, refinement techniques, and lessons learned from the deployment of RV on an autonomous robot; (5) a case study embedding online, stream-based, hardware RV on Robonaut2 hardware on loan from NASA, demonstrating successful real-time fault disambiguation in this resource-constrained environment. Section 2 overviews the logic MLTL and notation used. New monitoring encodings are detailed and proven in correct in section 3, then implemented with optimization in section 4 and presented with experimental performance characterization. Section 5 covers embedding of these observers on Robonaut2 and development of specifications for fault disambiguation. Finally, lessons learned and opportunities for future work are highlighted in section 6.

2 Preliminaries

R2U2 uses mission-time LTL (MLTL) for future-time temporal specification [22, 15]. MLTL is based on MTL [3] which adds time bounds to the temporal opera-

tors of LTL, but with tighter constraints on the time intervals. A closed interval over naturals $I = [a, b]$ ($0 \leq a \leq b$ are natural numbers) is a set of naturals $\{i \mid a \leq i \leq b\}$. I is *bounded* iff $b < +\infty$; otherwise I is *unbounded*. MLTL temporal operators are restricted to bounded intervals. MLTL abstracts Metric Temporal Logic (MTL) [2], which includes open and half-open intervals. Every MTL open or half-open bounded interval corresponds to an equivalent closed bounded interval, e.g., (1,2) = $\emptyset$, (1,3) = [2,2], (1,3] = [2,3], etc. Let $\mathcal{P}$ be a set of atomic propositions ($\mathcal{AP}$) and φ, φ_1 and φ_2 be MLTL formulas. The definition of MLTL is given below:

Definition 1. *(MLTL Syntax) The syntax of an MLTL formula φ over a set of atomic propositions $\mathcal{AP}$ is recursively defined as:*

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box_I \varphi \mid \Diamond_I \varphi \mid \varphi_1 \mathcal{U}_I \varphi_2 \mid \varphi_1 \mathcal{R}_I \varphi_2$$

where $p \in \mathcal{AP}$ is an atom, φ_1 and φ_2 are MLTL formulas. I is an interval $[lb, ub]$, $lb \leq ub$ and $lb, ub \in \mathbb{N}$, or simply $[ub]$ if $lb = 0$. Given two MLTL formulas φ_1, φ_2, we denote $\varphi_1 \equiv \varphi_2$ if they are semantically equivalent. In MLTL semantics, we define $\mathsf{false} \equiv \neg\mathsf{true}$, $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\neg(\varphi_1 \mathcal{U}_I \varphi_2) \equiv (\neg\varphi_1 \mathcal{R}_I \neg\varphi_2)$ and $\neg\Diamond_I \varphi \equiv \Box_I \neg\varphi$. MLTL keeps the standard operator equivalences from LTL, including $\Diamond_I \varphi \equiv (\mathsf{true}\ \mathcal{U}_I \varphi)$, $(\Box_I \varphi) \equiv (\mathsf{false}\ \mathcal{R}_I \varphi)$. Notably, MLTL discards the next ($\mathcal{X}$) operator, which is essential in LTL, since $\mathcal{X}\varphi$ is semantically equivalent to $\Box_{[1,1]}\varphi$ (see [15]). The semantics of MLTL satisfiability is defined as follows:

Definition 2. *(MLTL Semantics) The satisfaction of an MLTL formula φ, over a set of propositions $\mathcal{AP}$, by a computation/trace π starting from position i (denoted as $\pi, i \models \varphi$) is recursively defined as:*

- $\pi, i \models \mathsf{true}$,
- $\pi, i \models p$ *iff* $p \in \pi[i]$,
- $\pi, i \models \neg\varphi$ *iff* $\pi, i \not\models \varphi$,
- $\pi, i \models \varphi_1 \wedge \varphi_2$ *iff* $\pi, i \models \varphi_1$ *and* $\pi, i \models \varphi_2$,
- $\pi, i \models \varphi_1 \mathcal{U}_{[lb,ub]} \varphi_2$ *iff* $|\pi| \geq i + lb$ *and, there exists* $j \in [i + lb, i + ub]$ *such that* $\pi, j \models \varphi_2$ *and for* $k < j$ *every* $k \in [i + lb, i + ub]$ $\pi, k \models \varphi_1$,

where computation π is a finite trace (i.e., sequence) of propositions over the set $\mathcal{AP}$ with length $|\pi|$. $p \in \mathcal{AP}$ is an atom. The set of propositions at index i of π is denoted as $\pi[i] (i \geq 0)$.

2.1 Abstract Syntax Tree and Execution Sequence

R2U2 executes runtime reconfigurable specifications by constructing an *Abstract Syntax Tree* (AST) of logical observers wherein each node produces an *execution sequence* as output which can be used by other nodes in the tree.

Definition 3. *(Execution Sequence) (adapted from [22]) An* **execution sequence** *for an MLTL formula φ, denoted by $\langle T_\varphi \rangle$, over computation π is a sequence of verdict tuples $T_\varphi = (v, \tau)$ where $\tau \in \mathbb{N}_0$ is a time stamp and $v \in \{true, false\}$ is a verdict. We use a superscript integer to access a particular element in $\langle T_\varphi \rangle$, e.g., T_φ^0 is the first element in execution sequence $\langle T_\varphi \rangle$.*

We write $T_\varphi.\tau$ to access τ and $T_\varphi.v$ to access v of such an element. We say T_φ holds if $T_\varphi.v$ is true and T_φ does not hold if $T_\varphi.v$ is $false$. For a given execution sequence $\langle T_\varphi \rangle = T_\varphi^0, T_\varphi^1, T_\varphi^2, T_\varphi^3, \ldots$, the tuple accessed by T_φ^n corresponds to a section of satisfication of φ as follows: for all time stamps $i \in [T_\varphi^{n-1}.\tau + 1, T_\varphi^n.\tau]$, $\pi, i \models \varphi$ in case $T_\varphi^n.v$ is true and $\pi, i \not\models \varphi$ in case $T_\varphi^n.v$ is $false$.

2.2 Propagation Delay

Each temporal operator in MLTL is accompanied by a closed natural integer bound, $I = [lb, ub]$. As these operators chain together, the decidability of a given node becomes a function of its bound and the bounds of its inputs.

Definition 4. *(Propagation Delay) The propagation delay of an MLTL formula φ is the the time between when a set of propositions $\pi[i]$ (i.e., input) arrives at φ, and when it is known if $\pi, i \models \varphi$ (i.e., output). A node's* worst case delay *(**wcd**) is the maximum propagation delay it can experience, and the minimum value is the* best case delay *(**bcd**).*

Definition 5. *(Propagation Delay Semantics) Let φ be an MLTL formula where $\varphi.bpd$ is the best-case propagation delay of formula φ and $\varphi.wpd$ is its worst-case propagation delay. If φ is a unary operator, then let its direct subformula be ψ; else, if φ is a binary operator, then let ψ_1, ψ_2 be its direct subformulas. Let Propagation Delay of formula h be defined as follows:*

$$\text{if } \varphi \in \mathcal{AP} : \begin{cases} \varphi.wpd & = 0 \\ \varphi.bpd & = 0 \end{cases} \qquad \text{if } \varphi = \neg\psi : \begin{cases} \varphi.wpd & = \psi.wpd \\ \varphi.bpd & = \psi.bpd \end{cases}$$

$$\text{if } \varphi = \Box_{[\varphi.lb, \varphi.ub]}\psi \text{ or } \varphi = \Diamond_{[\varphi.lb, \varphi.ub]}\psi : \begin{cases} \varphi.wpd & = \psi.wpd + \varphi.ub \\ \varphi.bpd & = \psi.bpd + \varphi.lb \end{cases}$$

$$\text{if } \varphi = \psi_1 \vee \psi_2 \text{ or } \varphi = \psi_1 \wedge \psi_2 : \begin{cases} \varphi.wpd & = max(\psi_1.wpd, \psi_2.wpd) \\ \varphi.bpd & = min(\psi_1.bpd, \psi_2.bpd) \end{cases}$$

$$\text{if } \varphi = \psi_1 \mathcal{U}_{[\varphi.lb, \varphi.ub]}\psi_2 \text{ or } \varphi = \psi_1 \mathcal{R}_{[\varphi.lb, \varphi.ub]}\psi_2 : \begin{cases} \varphi.wpd & = max(\psi_1.wpd, \psi_2.wpd) + \varphi.ub \\ \varphi.bpd & = min(\psi_1.bpd, \psi_2.bpd) + \varphi.lb \end{cases}$$

3 New Future-Time Algorithms for R2U2

To improve real-time performance and reduce resource usage, we contribute new encodings of asynchronous, future time MLTL operators. Single-writer, many-reader, ring buffers called *shared connection queues* (SCQs) replace the FIFOs of the original operators [22]. The SCQ backed operators allows further implementation optimizations, discussed in section 4. While developed to reduce real-time resource requirements, we found SCQ backed operators necessary for other advancements like model-predictive runtime verification [27].

3.1 Shared Connection Queues

A SCQ is a circular buffer of verdict tuples with one write pointer and one or more read pointers, and buffers verdicts from a child subformula to be read by multiple parent expressions. These supplant the synchronization queues utilized in [22]. The Shared Ring Buffer is a similar structure used in multi-threading software (e.g., [25, 14]), which inspired the SCQ. Figure 1 shows how SCQs are embedded in an MLTL AST, with read pointers for each parent and a write pointer for the child.

Reading and Writing Algorithms 1 and 2 show SCQ read and write operations. Each SCQ manages a write pointer while observers maintain read pointers for each child queue. SCQs store verdict intervals using *aggregation* [22], wherein the latest tuple's timestamp is overwritten by subsequent timestamp values if their truth values are equal. For example, if the SCQ contains $\{(\mathsf{true}, 10), (\mathsf{false}, 15)\}$, then during the timestamp interval $[11, 15]$ the verdicts are all false. If the next input is $(\mathsf{false}, 16)$, the content becomes $\{(\mathsf{true}, 10), (\mathsf{false}, 16)\}$.

To enforce monotonic reads from an SCQ, the variable τ_e tracks the last timestamp a reader used from a SCQ and prevents the output of a result with a smaller timestamp than any previous result. When reading, the read pointer *rd_ptr* searches the SCQ until encountering a value greater or equal to τ_e, else it returns empty. The circular topography of the SCQ is omitted from the algorithms for clarity. In practice, the pointer increments and decrements by the size of a verdict tuple, modulo the size of the queue.

Algorithm 1: Function: write()

```
   input: execution sequence tuple: data
1  if SCQ[SCQ.wr_ptr].v == data.v then //Setup aggregation
2  |  SCQ.wr_ptr − −;
3  end
4  SCQ[SCQ.wr_ptr] ← data;
5  SCQ.wr_ptr + +;
```

Algorithm 2: Function: read()

```
   input  : &rd_ptr, τ_e
   output: Empty or data
1  if ∗rd_ptr == SCQ.wr_ptr then
2  |  return Empty;
3  end
4  data ← SCQ[∗rd_ptr];
5  while data.τ < τ_e do
6  |  rd_ptr + +;
7  |  if ∗rd_ptr == SCQ.wr_ptr
   |    then
8  |  |  return Empty;
9  |  end
10 |  data ← SCQ[∗rd_ptr];
11 end
12 return data;
```

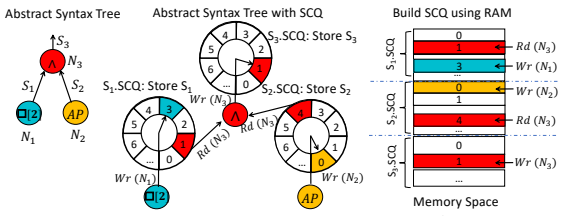


Fig. 1. Implementation of SCQs, in the AST and in memory with rules of operation.

Queue Sizing The required buffer size for each observer is computed a priori by recursively sizing the SCQs in an MLTL AST based the best and worst-case delays of their subexpressions. The size of the output queue for a node g with sibling nodes $\mathcal{S}_g$ that share a common parent:

$$size(g.Queue) = max(max\{s.wpd \mid \forall s \in \mathbb{S}_g\} - g.bpd, 0) + 1$$

Sizing queues based on the worst-case-delay, guarantees verdicts are consumed by the parent nodes before the write pointer recirculates and old data is overwritten – firmly bounding the required memory. Software RV monitors can use statically allocated memory to avoid undesirable effects of allocation since the memory bounds are known in advance. In hardware RV deployments, the SCQ is built from Block RAMs (BRAMs), an FPGA memory resource. Each BRAM can be partitioned into multiple SCQs.

3.2 MLTL Operator Observers with SCQs

Algorithms 3–6 in Figure 2 demonstrate our new encodings of the four required future-time MLTL asynchronous observers using SCQs. Whereas [22] used one of two observers for $\Box_{[lb,ub]}$ depending on the bounds, this encoding only uses one observer. We also include Algorithm 3 which denotes the addition of an atomic proposition into the SCQs.

3.3 Correctness of New MLTL Observers

Correctness of algorithms 3–3 follows immediately from the SCQ read and write operations.

Theorem 1 (Correctness of the $\Box$-operator). *The observer in Algorithm 5 correctly implements $\langle T_\varphi \rangle, \pi, i \models \Box_J \varphi$ for any $\langle T_\varphi \rangle$.*

Proof (Proof of Theorem 1). In [22] it is shown that:

$$\forall i : (i - lb \in [n, n + ub - lb] \rightarrow \pi, i \models \varphi) \Leftrightarrow \Box_{lb,ub}\varphi$$

Therefore the verdict of $\Box_{lb,ub}\varphi$ at time n is only dependent on $\pi, i \models \varphi$ for values of i that satisfy:

- $(i - lb) \geq n$: Since $(ub - lb) \not< 0$ and $n \in \mathbb{N}_0$ then $i - lb \geq 0$ which is upheld by the check on line 11 of Algorithm 5 and returns false time steps where $\pi, i \models \neg\varphi$ while $i - lb \geq 0$. Intuitively, this suppresses false verdicts unless the lower bound has been met.
- $(i - ub) \leq n$: By the same logic $(i - ub) \leq 0$ which is upheld by the second check on line 8 does not allow. Intuitively, this suppresses true verdicts unless the full duration of the interval has been observed.

Thus, a rising edge of φ (captured by lines 4-5) must be seen at a time $\leq (i - lb)$ and no falling condition can be seen before a time $> (i - ub)$. The first check on line 8 ensures φ has held for at least the duration of J, satisfying this condition. With output occurring iff the original equivalence relation is satisfied, the theorem follows.

Due to size, proofs for Algorithm 4 and Algorithm 6 will be available online[1].

[1] http://temporallogic.org/research/FORMATS20/index.html

Algorithm 3: LOAD (ATOMIC) Operation

```
   input : atomic proposition: AP, current timestamp: t_R
1  for each new arriving timestamp do
2  |  N.SCQ.write([AP, t_R]);
3  end
```

Algorithm 4: NEGATION Operator: $\neg$

```
   Var: τ_e
   Init: τ_e ← 0
1  data ← N.iSCQ.read(τ_e, &N.rd_ptr);
2  while data is not Empty do
3  |  data ← N.iSCQ.read(τ_e, &N.rd_ptr);
4  |  N.SCQ.write([!out.v, out.τ]);
5  |  τ_e ← out.τ + 1;
6  end
```

Algorithm 5: AND Operation: $\wedge$

```
   Var: τ_e
   Init: τ_e ← 0
1  result ← [false, −1];
2  data_1 ← N.iSCQ_0.read(τ_e, &rd_ptr);
3  data_2 ← N.iSCQ_1.read(τ_e, &rd_ptr);
4  while data_1 is not Empty || data_2 is not Empty do
5  |  if both data_1 and data_2 are not empty then
6  |  |  if data_1.v and data_2.v then
7  |  |  |  result = [true, min(data_1.τ, data_2.τ)];
8  |  |  else if !data_1.v and !data_2.v then
9  |  |  |  result = [false, max(data_1.τ, data_2.τ)];
10 |  |  else if data_1.v then
11 |  |  |  result = [false, data_2.τ];
12 |  |  else
13 |  |  |  result = [false, data_1.τ];
14 |  |  end
15 |  else if data_1 is Empty then
16 |  |  if !data_2.v then
17 |  |  |  result = [false, data_2.τ];
18 |  |  end
19 |  else if !data_1.v then
20 |  |  result = [false, data_1.τ];
21 |  end
22 |  if result.τ! = −1 then
23 |  |  N.SCQ.write(result);
24 |  |  τ_e ← result.τ + 1;
25 |  else
26 |  |  Break;
27 |  end
28 |  data_1 ← N.iSCQ_0.read(τ_e, &rd_ptr);
29 |  data_2 ← N.iSCQ_1.read(τ_e, &rd_ptr);
30 end
```

Algorithm 6: GLOBAL Operation: $\Box_{[lb,ub]}$

```
   Var: m_↑, [v_pre, τ_pre], τ_e
   Init: [v_pre, τ_pre] ← [false, −1], τ_e ← 0
1  data = N.iSCQ.read(τ_e, &rd_ptr);
2  while data is not Empty do
3  |  τ_e = data.τ + 1;
4  |  if data.v and !v_pre then
5  |  |  m_↑ ← τ_pre + 1;
6  |  end
7  |  if data.v then
8  |  |  if data.τ − m_↑ ≥ ub − lb and data.τ − ub ≥ 0
   |  |   then
9  |  |  |  N.SCQ.write([true, data.τ − ub]);
10 |  |  end
11 |  else if data.τ − lb ≥ 0 then
12 |  |  N.SCQ.write([false, data.τ − lb]);
13 |  end
14 |  [v_pre, τ_pre] ← data;
15 |  data = N.iSCQ.read(τ_e, &rd_ptr);
16 end
```

Algorithm 7: UNTIL Operation: $\mathcal{U}_{[lb,ub]}$

```
   Var: m_↓, [v_pre, τ_pre], τ_e, τ_res
   Init: [v_pre, τ_pre] ← [false, 0] or [v_pre, τ_pre] ← [true, −1],
         τ_res ← 0, τ_e ← 0
1  data_1 = N.iSCQ_0.read(τ_e, &rd_ptr);
2  data_2 = N.iSCQ_1.read(τ_e, &rd_ptr);
3  while data_1 is not Empty and data_2 is not Empty
   do
4  |  result = [false, −1];
5  |  t_min = min(data_1.τ, data_2.τ);
6  |  τ_e ← t_min + 1;
7  |  if v_pre and !data_2.v then
8  |  |  m_↓ ← τ_pre + 1;
9  |  end
10 |  if data_2.v then
11 |  |  result ← [true, t_min − lb];
12 |  else if !data_1.v then
13 |  |  result ← [false, t_min − lb];
14 |  else if t_min ≥ ub − lb + m_↓ then
15 |  |  result ← [false, t_min − ub]
16 |  end
17 |  if result.τ ≥ τ_res then
18 |  |  τ_res ← result.τ + 1;
19 |  |  N.SCQ.write(result);
20 |  end
21 |  [v_pre, τ_pre] ← data_2;
22 |  data_1 = N.iSCQ_0.read(τ_e, &rd_ptr);
23 |  data_2 = N.iSCQ_1.read(τ_e, &rd_ptr);
24 end
```

Fig. 2. Implementations of asynchronous, future-time MLTL observers using SCQs. For each algorithm: $\mathcal{N}$ is the current node, $\mathcal{N}.SCQ$ is the output SCQ of $\mathcal{N}$, and $\mathcal{N}.iSCQ$ is input SCQ being read. In binary operators, there are two input queues: $\mathcal{N}.iSCQ\_0$ and $\mathcal{N}.iSCQ\_1$

4 Optimizations and Experimental Performance Analysis

In an MLTL formula, repeated sub-expressions generate redundant observer instructions, needlessly increasing required queue space and execution time. Compilers use *common subexpression elimination* (CSE) [7] to share the output of repeated expressions. Figure 3 demonstrates the application of CSE to MLTL ASTs. This was not possible with the observer encodings from [22] and uses SCQs with multiple readers. Algorithm 8 is used to remove duplicate branches of the formula AST.

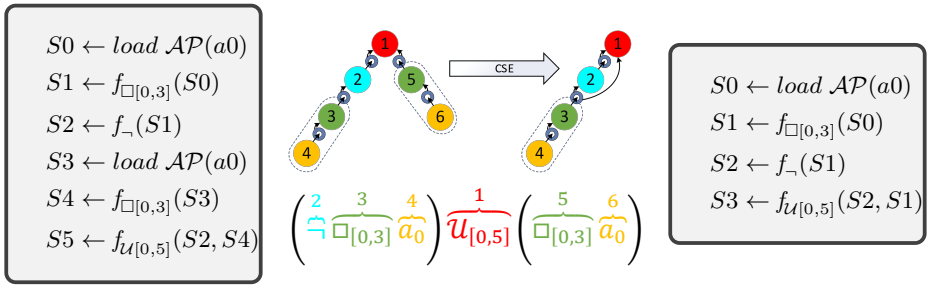


Fig. 3. Example of CSE on an MLTL formula where nodes 3 and 5 have identical children. Sharing the output of node 3 removes one repetition of this sequence, saving two queues and two instructions.

Experimental Demonstration of Improved Average Performance To measure the impact of CSE with SCQs as presented in Section 4, the 10,000 random MLTL benchmark formulas used in [15] were converted to observer trees and queue configurations with and without CSE enabled. The benchmark set varies the length, number of variables, and probability of the $\mathcal{U}$-operator.

The R2U2 configuration compiler is a single-threaded python application and was ran in parallel (12 instances at a time) on a 2019 MacBook Pro with a i9-9880H Intel Core i9 and 32 GB system RAM. In total, the 20,00 runs across 12 processes completed in under 15 seconds wall time and the duration of each process was measured using the Python 3.7.7 standard time library process_time function which counts system and user CPU (but not sleep) time with the most precise clock available.

Over the whole set, the number of R2U2 observer nodes dropped 27.06% from 788,095 to 574,822 and the total queue depth required decreased 4.28% from 42,300,361 to 40,491,507 with only a −10.25% increase in running time from 57.66 to 63.57 seconds of CPU time. Figure 4 shows histograms of AST and SCQ reduction respectively. Only 30 of the 10,000 saw no improvement.

The reduction in AST nodes is significant and translates proportionally to execution time. An mean and median savings of 24% on random formulas are impressive and the greater repetition in human written specification gives hope that this may be a low average compared to real specifications. The queue space reduction was less impressive by percentage; however it still saved an median of 100 slots per formula which is important as BRAMs are less plentiful on FPGAs.

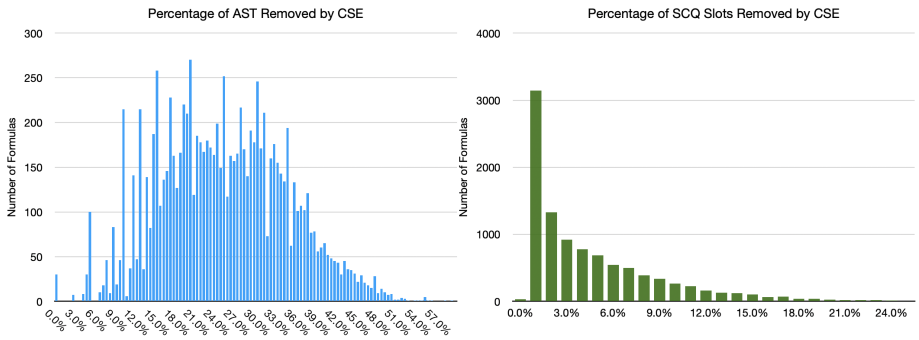


Fig. 4. Reduction in AST nodes (left) and SCQ slots (right) as percentage of unoptimized size. Hight indicates number of formulas from MLTL benchmark set [15].

The long time bounds in the benchmark set is hoped to contribute to this, but more testing is needed.

5 Theory into practice: Robonaut2

Algorithm 8: `CSE`(T, S)

Input : AST Tree: T, Set: $S = \{(label, node)\}$
Output: optimized AST: T

```
// Recuse through T in post-order
Let N = root(T)
if leftChild(N) ≠ ∅ then CSE(leftChild(N), S)
if rightChild(N) ≠ ∅ then CSE(rightChild(N), S)
// Build expression label
N.label = ['(']
if leftChild(N) then
    N.label += leftChild(N).label
N.label += N.name
if rightChild(N) then
    N.label += rightChild(N).label
N.label += [')']
// Trim common subexpressions
if (N.label, •) ∉ S then
    S = S ∪ (N.label, N)
else
    Let M∃T such that (N.label, M) ∈ S
    T = T ∪ (parent(N), M)
    T = T − (parent(N), N)
end
```

Robonaut2's legs are comprised of serial elastic actuators with torsional springs [20]. Precise measurement of the spring displacement are used to cap applied force, affording near-human dexterity while remaining safe in confined spaces with astronauts [4]. After deployment, the *Absolute Position Sensors* (APSs) have been observed to sporadically initialize incorrectly by $\approx$ 2.1 rad (120 deg). In this fault condition, safety checks fail due to a perceived high torque loading. This is well beyond the physical hard-stop of the joint, but is not treated as a spurious error.

To increase availability and resilience, Robonaut2 must be able to automatically trigger corrective action without compromising existing safety guarantees.

Constraints Resource constraints and architecture inflexibility are inherent challenges of supplementing an existing system. The Robonaut2 team requested fault disambiguation directly on the joint controller FPGA. This provides increased observability, minimizes additional messages on the control bus, and does not invalidate the flight code certification of the paired microcontroller. However, left-over space is limited and additional logic could not impact the response time of the existing controller. Additionally, the system's remote deployment limited available debug information. Consequently, initial specification was derived from a plain-language description of the fault mechanics by subject-matter experts while awaiting a real trace.

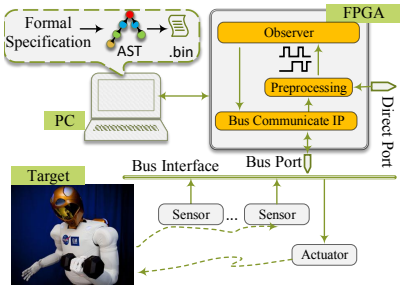


Fig. 5. Location of RV observer. Compiled specifications are loaded onto the FPGA where a RV observer monitors the internal sensor values

Solution Architecture Figure 5 shows the desired architecture. During development, a serial debug port loads specifications and returns verdicts. In flight, Robonaut2's configuration system will handle specification loading. R2U2 is realizable, responsive, and unobtrusive [23]; it embeds observers for Robonaut2's symptoms in hardware, returns observer verdicts at the system clock rate, and is adaptable to the highly-constrained operational environment without affecting existing joint control, respectively. We apply two of R2U2's reasoning layers: signal processing (which processes incoming signals into Boolean atomics) and temporal observation (which evaluates MLTL specifications). Our use case requires early-as-possible identification of failure, necessitating using R2U2's asynchronous mode. The existing flight configuration routes all sensors, actuator control, and communications through the FPGA while a microcontroller runs high-rate model-based control algorithms [5]. Since the FPGA is the nexus of the actuator's sensors, all required data can be accessed on-chip.

5.1 Embedding Runtime Verification

R2U2 allows runtime configuration of the observer specifications, but limits to the size and duration of these specifications are set by design-time parameters. For R2U2 to dynamically reconfigure specifications at runtime (without resynthesis or recertification), we utilize BRAMs for instruction memory, variable memory, and queues; see [23]. R2U2 memory requirements are complex, but deterministic, and driven by queue depth and timestamp length. Design-time calculations have been developed [26] to explore the valid configurations under our memory constraint. Figures 6 and 7 show the impact on FPGA look-up-tables (LUTs) and BRAM respectively. A max queue depth of 20 and time-stamp

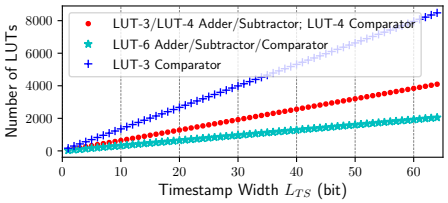


Fig. 6. LUT resource usage for timestamp length L_{TS}. Growth is linear, but the rate is dependent on FPGA process type.

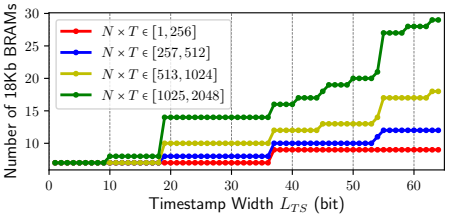


Fig. 7. BRAM resource usage for timestamp length L_{TS} and total binary operators' longest trace delay $N \times T$.

length of 16-bits were selected from expert and system operator's recommendations. This increased the LUT utilization of the FPGA from 51.2% to 79.81% and increased the number of BRAMs used from 2 to 27 our of 32. A video demo[2] shows R2U2 running live on the R2 platform, reasoning over the joint state, evaluating temporal observers, and dynamically configuring specifications without stopping the robot.

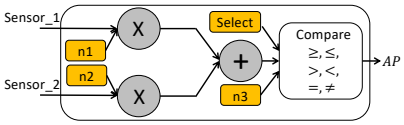


Fig. 8. R2U2 atomic checker. Orange blocks are configurable online.

Boolean Checker Construction R2U2's runtime-configurable Atomic Checkers, shown in Fig. 8, convert the native sensor format to Boolean variables used in specification. For example, the *EncPos* sensors value indicate the rotation degree of the motor. Robonaut2's native encoder format is a 19-bit number, where the highest bit is an error flag and the lower 18 bits represent the encoder count. This presents two challenges: (1) the *EncPos* is reset to 0 at initialization regardless of the actual position; (2) to compare with the APS values, this count must be scaled and offset. Taken together, R2U2 must reconfigure the offset before using encoder values. For *EncPos*, we let **sensor_1** take the raw value as input and **sensor_2** be constant 1. In this configuration, **n1** is the scale factor, and **n2** is the configurable offset. The output AP is result of the selected comparison with the **n3** reference value.

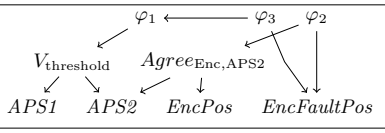
5.2 Specification

Design Our specifications need to disambiguate between three modes (APS1 faulty, APS2 faulty, or no fault) without false positives. We initially encode Robonaut2's team's fault description: *If the differences between APSs are larger than 1 radian, then the APS that disagrees with the encoder by more than 0.01 radian is at fault.* We assume: (1) Agreement with the encoder value implies correct APS position, (2) Agreement between any two position sources implies the minority opinion is incorrect, i.e. sensor voting. To prevent false positives we ensure states hold for at least three timesteps before reporting a fault. An "encoder fault position" signal is set by Robonaut2 when the encoder and APS1 disagree, removing the need for one atomic checker redundant. Our MLTL specifications

[2] http://temporallogic.org/research/R2U2/R2U2-on-R2_demo.mp4

reason over the APS1 position, APS2 position, encoder position, and encoder fault position sensor inputs; see Table 2. The R2U2 configuration requires 17 instructions, 14 SCQs, and 29 queue slots with a max depth of 4 without CSE. Applying CSE reduces that to 14 instructions, 11 SCQs, and 26 queue slots with a max depth of 4.

Table 2. Fault disambiguation specification – revision 1

<table>
<tr><td colspan="2">R2U2 Configuration</td></tr>
<tr><td>Bus Values</td><td>Temporal Formulas</td></tr>
<tr><td>APS1: Position [rad]</td><td>$\varphi_1 = \square_{[0,3]}(V_{\text{threshold}})$</td></tr>
<tr><td>APS2: Position [rad]</td><td>$\varphi_2 = \text{FaultEncPos} \wedge \square_{[0,3]}(Agree_{\text{Enc,APS2}}) \rightarrow \text{APS1}_{\text{Wrong}}$</td></tr>
<tr><td>EncPos: Position [rad]</td><td>$\varphi_3 = \varphi_1 \vee !\text{FaultEncPos} \rightarrow \text{APS2}_{\text{Wrong}}$</td></tr>
<tr><td>EncFaultPos: Encoder Fault [bool]</td><td>Observer Tree</td></tr>
<tr><td>Signal Processing</td><td rowspan="2"> 
</td></tr>
<tr><td>$V_{\text{threshold}} = |\text{APS1} - \text{APS2}| > 1$ rad
$Agree_{\text{Enc,APS2}} = |\text{APS2} - \text{EncPos}| > 0.01$ rad</td></tr>
</table>

Validation After initial specification development, a terrestrial Robonaut2 developed the fault of interest. To validate our specifications, we ran R2U2 over the recorded traces. In Fig. 9-10 the top timeline shows the encoder (red), APS1 (blue, labeled motor), and APS2 (yellow, labeled joint) positions in radians. The lower timeline shows the R2U2 verdicts of the fault-case specifications. In Fig. 9 the APS fault occurs at 43 seconds. The expected > 2.1 rad shift in APS position is flagged by $V_{\text{threshold}}$ correctly. Notice that the encoder jumps to an implausible 998 radians, violating the sensor voting assumption. Figure 10 records an attempted recovery. While appearing successful, the difference between the three sensors after time 19 is still too wide to unlock the e-stop. Additional $V_{\text{threshold}}$ correctly encodes that this is a different failure mode. This data reveals an implicit assumption that encoder values freeze during a fault.

Revision With our new insignt on the fault behavior, we revise the specification strategy: *If there is a sudden, large jump in the encoder and an APS's position report, the APS that jumped is at fault.* The assumptions of our new strategy: (1) A sufficiently large discontinuity in the data is the fault signature, (2) In the fault case, only the faulty APS 'moves'. To compare the APS value before and after fault, we must identify the timestep of the fault – which is when the encoder goes out-of-range. To determine the "moving" APS, we can divide the joint range into sections and use the signal processing layer to get a Boolean a_n indicating APS1 in region n (and similarly with b_n and APS2). Now the temporal observers can check if the APS is in the same region before and after the encoder jump. The size of n dictates the maximum travel distance before triggering a region change. We select n such that the maximum travel is about

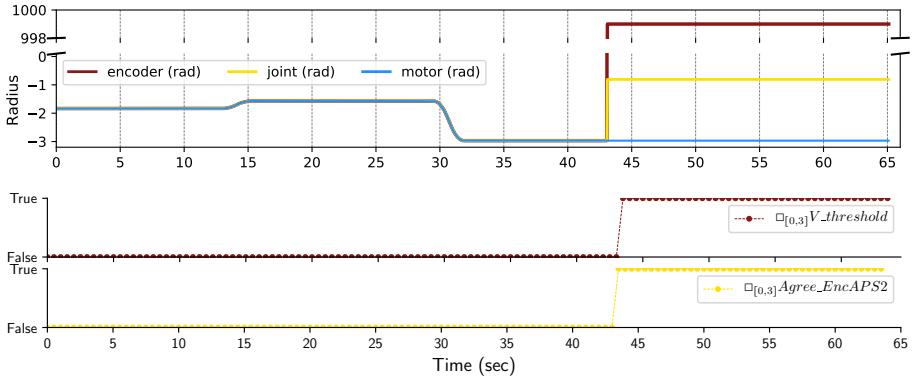


Fig. 9. Ground R2: APS Fault

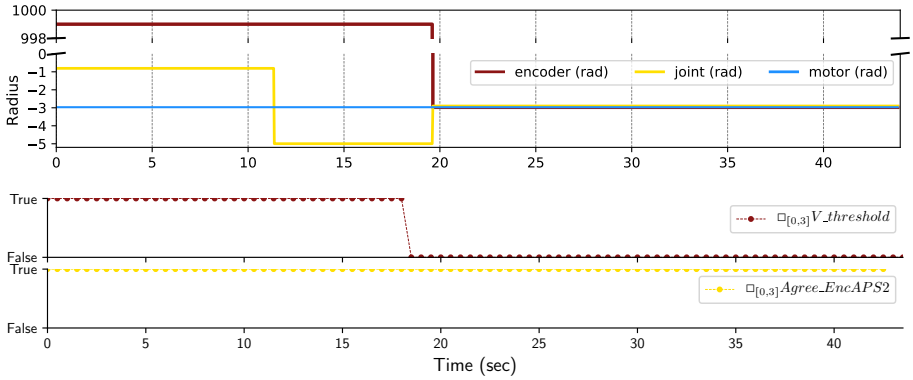


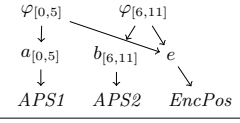
Fig. 10. Ground R2: Unsuccessful Recovery

half the fault discontinuity: ≈ 1 rad. The range of the APS is $[-\pi, \pi]$, requiring 6 regions, $(a_1, a_2, \ldots, a_6)$ and $(b_1, b_2, \ldots, b_6)$ to meet the target region size. The fault only occurs when arming a parked actuator so we are not concerned with travel during a fault. Also, nominal travel across a boundary is not accompanied with an encoder range error, further preventing false positives. Table 3 lists the MLTL and signal layer specification. The R2U2 configuration requires 154 instructions, 140 SCQs, and 196 queue slots with a max depth of 3 without CSE. With CSE this is reduced to 100 instructions, 86 SCQs, and 142 queue slots with a max depth of 3. Here, CSE has makes a huge difference requiring the number of SCQs be increased by one bit to 128 instead of 2 bits to 512 to accommodate the formula. The 33% reduction in instructions shows the efficiency of configurations with many repeated components.

Verification As a consistency check, an MLTL SMT solver proved the specifications were not tautologies, and were falsifiable [15]. Finally, recorded fault traces were played back through real hardware, successfully catching the fault with no false positives during nominal operation.

Table 3. Fault disambiguation specification – revision 2

<table>
<tr><td colspan="2">R2U2 Configuration</td></tr>
<tr><td>Bus Values</td><td>Temporal Formulas</td></tr>
<tr><td>APS1: Position [rad]
APS2: Position [rad]
EncPos: Position [rad]</td><td>$\varphi_n = (a_n \wedge \neg e) \wedge \diamond_{[1,2]}(\neg a_n \wedge e) \rightarrow \text{APS1}_{\text{Fault}} \quad \forall n \in [0,5]$
$\varphi_{m+6} = (b_m \wedge \neg e) \wedge \diamond_{[1,2]}(\neg a_m \wedge e) \rightarrow \text{APS2}_{\text{Fault}} \; \forall m \in [0,5]$</td></tr>
<tr><td>Signal Processing</td><td rowspan="2">Observer Tree</td></tr>
<tr><td>$e = \text{EncPos} > 100$
$a_n = \pi(\frac{n}{6} - 1) < \text{APS1} < \pi(\frac{n+1}{6} - 1) \forall n \in [0,5]$
$b_n = \pi(\frac{n}{6} - 1) < \text{APS2} < \pi(\frac{n+1}{6} - 1) \forall n \in [0,5]$</td></tr>
</table>



6 Conclusion

We have successfully designed, proven, and implemented a new set of future time temporal observers for runtime verification with R2U2. We successfully applied these to fault disambiguation on real Robonaut2 hardware on loan from NASA – a real, demanding, and constrained environment. Importantly, the techniques presented in sections 3 and 4 are not exclusive to this application or R2U2.

Working with FPGA limitations provided important lessons on the relation between specification complexity and hardware resources. In fig. 6 LUTs required is linear with time-stamp length, transistor count (and therefore chip space and power) is exponential with LUT size. Also, the discontinuities in Fig. 7 are due to BRAM width alignment. Since both LUT type and BRAM width are properties of the FPGA, the target hardware can drastically change the maximum size of specification even with the same amount of logic and BRAM free. For a hardware R2U2 deployment, BRAM will probably be the limiting resource. This may not be true for other monitors, but it's the price of reconfigurability which allows RV to be embedded, certified, and deployed flexibly and was a requirement of the R2 team.

Future Work With the core observers implemented, we can re-encode the extended operator set of MLTL which features operators like "release" that are currently accepted by the R2U2 formula compiler but encoded via the equivalence relation in section 2. These additional encodings would reduce the number of negations in the AST and therefor the amount of SCQ space required. Additional design-time optimizations to the AST are also under investigation.

On the application side, we are working toward distributing specifications across RV monitors on multiple FPGAs. This extension has potential to increase the set of specifications that can be monitored, both by utilizing more of the leftover fabric on the platform, and by allowing observers to reason over proprieties that cannot by observed from a single location.

References

1. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Towards rule-based dynamic safety monitoring for mobile robots. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots. pp. 207–218. Springer (2014)

2. Alur, R., Henzinger, T.: A really temporal logic. J. ACM **41**(1), 181–204 (1994)
3. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. Information and Computation **104**(1), 35–77 (1993)
4. Badger, J., Hulse, A., Taylor, R., Curtis, A., Gooding, D., Thackston, A.: Model-based robotic dynamic motion control for the robonaut 2 humanoid robot. In: 2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids). pp. 62–67 (Oct 2013). https://doi.org/10.1109/HUMANOIDS.2013.7029956
5. Badger, J., Gooding, D., Ensley, K., Hambuchen, K., Thackston, A.: ROS in Space: A Case Study on Robonaut 2, pp. 343–373. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-26054-9_13, https://doi.org/10.1007/978-3-319-26054-9_13
6. Clemens, J., Pal, R., Sherrell, B.: Runtime state verification on resource-constrained platforms. In: MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM). pp. 1–6. IEEE (2018)
7. Cooper, K., Eckhardt, J., Kennedy, K.: Redundancy elimination revisited. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 12–21. ACM (2008)
8. Cowley, A., Taylor, C.J.: Towards language-based verification of robot behaviors. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 4776–4782. IEEE (2011)
9. Diftler, M.A., Mehling, J.S., Abdallah, M.E., Radford, N.A., Bridgwater, L.B., Sanders, A.M., Askew, R.S., Linn, D.M., Yamokoski, J.D., Permenter, F.A., Hargrave, B.K., Platt, R., Savely, R.T., Ambrose, R.O.: Robonaut 2 - the first humanoid robot in space. In: 2011 IEEE International Conference on Robotics and Automation. pp. 2178–2183 (May 2011). https://doi.org/10.1109/ICRA.2011.5979830
10. Halder, R., Proença, J., Macedo, N., Santos, A.: Formal verification of ros-based robotic applications using timed-automata. In: 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE). pp. 44–50. IEEE (2017)
11. Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., Rosu, G.: Rosrv: Runtime verification for robots. In: International Conference on Runtime Verification. pp. 247–254. Springer (2014)
12. J.M.Badger, A.M.Hulse, A.Thackston: Advancing safe human-robot interactions with robonaut 2. In: Proceedings of the 12th International Symposium on Artificial Intelligence, Robotics and Automation in Space (2014)
13. Kane, A., Chowdhury, O., Datta, A., Koopman, P.: A case study on runtime monitoring of an autonomous research vehicle (arv) system. In: Runtime Verification. pp. 102–117. Springer (2015)
14. Lee, P.P., Bu, T., Chandranmenon, G.: A lock-free, cache-efficient shared ring buffer for multi-core architectures. In: Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. pp. 78–79. ACM (2009)
15. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for mission-time ltl. In: Proceedings of 31st International Conference on Computer Aided Verification (CAV 2010). LNCS, vol. TBD, p. TBD. Springer, New York, NY, USA (July 2019). https://doi.org/TBD
16. Lu, H., Forin, A.: The design and implementation of p2v, an architecture for zero-overhead online verification of software programs. Tech. Rep. MSR-TR-2007-99, Microsoft Research (August 2007)

17. Luckcuck, M., Farrell, M., Dennis, L., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: a survey. arXiv preprint arXiv:1807.00048 (2018)
18. Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: Proceedings of the 54th Annual Design Automation Conference 2017. p. 35. ACM (2017)
19. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable cots-based real-time embedded systems. In: 2008 Real-Time Systems Symposium. pp. 481–491 (Nov 2008)
20. Pratt, G.A., Williamson, M.M.: Series elastic actuators. In: Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots. vol. 1, pp. 399–406 vol.1 (Aug 1995). https://doi.org/10.1109/IROS.1995.525827
21. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
22. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science (LNCS), vol. 8413, pp. 357–372. Springer-Verlag (April 2014)
23. Rozier, K.Y., Schumann, J.: R2u2: Tool overview. In: Proceedings of International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES). vol. 3, pp. 138–156. Kalpa Publications, Seattle, WA, USA (September 2017). https://doi.org/TBD, https://easychair.org/publications/paper/Vncw
24. Solet, D., Béchennec, J.L., Briday, M., Faucou, S., Pillement, S.: Hardware runtime verification of a rtos kernel: Evaluation using fault injection. In: 2018 14th European Dependable Computing Conference (EDCC). pp. 25–32. IEEE (2018)
25. Wong, L., Arora, N.S., Gao, L., Hoang, T., Wu, J.: Oracle streams: A high performance implementation for near real time asynchronous replication. In: 2009 IEEE 25th International Conference on Data Engineering. pp. 1363–1374. IEEE (2009)
26. Zhang, P., Li, J., Kempa, B., Zambreno, J., Jones, P.H., Rozier, K.: Formal specification encoding under platform resource constraints ((Under Submission))
27. Zhang, P., Zambreno, J., Jones, P.H., Rozier, K.: Model predictive runtime verification for embedded platforms with real-time deadlines ((Under Submission))

A Proofs of New MLTL Observer Correctness

A.1 Negation Operator: ¬

Theorem 2 (Correctness of the NEGATION Operator: ¬). *The observer in Algorithm 3 correctly implements $\pi, i \models \neg\varphi$ for any $\langle T_\varphi \rangle$.*

Algorithm 3: NEGATION Operator: $\neg\varphi$

Init: $\tau_{min} = -1$
if $T_\varphi.\tau > \tau_{min}$ **then**
  $\tau_{min} = T_\varphi.\tau$;
  return $(!T_\varphi.v, T_\varphi.\tau)$;
end

Proof (Proof of Theorem 2). For every input tuple, an output is produced with an identical time and negated verdict. The theorem follows from the SCQ read and write semantics, Alg. 2 and Alg. 1.

A.2 And Operator: $\wedge$

Theorem 3 (Correctness of the AND Operator: $\wedge$). *The observer in Algorithm 4 correctly implements $\pi, i \models \varphi \wedge \psi$ for any $\langle T_\varphi \rangle$ and $\langle T_\psi \rangle$.*

Algorithm 4: AND Operation: $\varphi \wedge \psi$

Init: $\tau_{min} = -1$
if $T_\varphi.\tau > \tau_{min}$ ***or*** $T_\psi.\tau > \tau_{min}$ **then**
  if T_φ *holds* ***and*** T_ψ *holds* **then**
    $\tau_{min} = \min(T_\varphi.\tau, T_\psi.\tau)$;
    return $(\mathsf{true}, \min(T_\varphi.\tau, T_\psi.\tau))$;
  else if T_φ *does not hold* ***and*** T_ψ *does not hold* **then**
    $\tau_{min} = \max(T_\varphi.\tau, T_\psi.\tau)$;
    return $(\mathsf{false}, \max(T_\varphi.\tau, T_\psi.\tau))$;
  else if T_φ *does not hold* **then**
    $\tau_{min} = T_\varphi.\tau$;
    return $(\mathsf{false}, T_\varphi.\tau)$;
  else if T_ψ *does not hold* **then**
    $\tau_{min} = T_\psi.\tau$;
    return $(\mathsf{false}, T_\psi.\tau)$;
  end
end

Proof (Proof of Theorem 3). Input may be aggregated and therefore may represent an interval of like-valued verdicts. Results returned in 4 cases:

1. Both true: all inputs true for all time steps where both inputs are known to be true by definition.
2. Both false: All time steps with a known false value of either input is false, therefore output is false up through maximum known time stamp.
3. Only first input is false. This is sufficient to declare the output false up to the final time-step of the false interval.
4. Only second input is false, same construction as previous point.

A.3 Global Operator: $\Box$

Theorem 4 (Correctness of the GLOBAL Operator: $\Box$). *The observer in Algorithm 5 correctly implements $\pi, i \models \Box_J \varphi$ for any $\langle T_\varphi \rangle$.*

Algorithm 5: GLOBAL Operation: $\Box_{[lb,ub]}\varphi$

```
   Init: m_↑ = 0, τ_min = −1
 1 if T_φ.τ > τ_min then
 2 |   if ⌐¬ of T_φ occurs then
 3 |   |   m_↑ = τ_min + 1;
 4 |   end
 5 |   τ_min = T_φ.τ;
 6 |   if T_φ holds and T_φ.τ ≥ max((ub − lb) + m_↑, ub) then
 7 |   |   return (true, T_φ.τ − ub);
 8 |   else if T_φ.τ ≥ lb then
 9 |   |   return (false, T_φ.τ − lb);
10 |   end
11 end
```

Proof (Proof of Theorem 4). In [22] it is shown that:

$$\forall i : (i - lb \in [n, n + ub - lb] \rightarrow \pi, i \models \varphi) \Leftrightarrow \Box_{lb,ub}\varphi$$

Therefore the verdict of $\Box_{lb,ub}\varphi$ at time n is only dependent on $\pi, i \models \varphi$ for values of i that satisfy:

- $(i - lb) \geq n$: Since $(ub - lb) \not< 0$ and $n \in \mathbb{N}_0$ then $i - lb \geq 0$ which is upheld by the check on line 11 of Algorithm 5 and returns false time steps where $\pi, i \models \neg\varphi$ while $i - lb \geq 0$. Intuitively, this suppresses false verdicts unless the lower bound has been met.
- $(i - ub) \leq n$: By the same logic $(i - ub) \leq 0$ which is upheld by the second check on line 8 does not allow. Intuitively, this suppresses true verdicts unless the full duration of the interval has been observed.

Thus, a rising edge of φ (captured by lines 4-5) must be seen at a time $\leq (i - lb)$ and no falling condition can be seen before a time $> (i - ub)$. The first check on line 8 ensures φ has held for at least the duration of J, satisfying this condition. With output occurring iff the original equivalence relation is satisfied, the theorem follows.

A.4 Until Operator: $\mathcal{U}$

Theorem 5 (Correctness of the $\mathcal{U}$-operator). *The observer stated in Algorithm 6 correctly implements the MTLT $\mathcal{U}$-operator. That is, for any two execution sequence $\langle T_\varphi \rangle$ and $\langle T_\psi \rangle$ it produces the same verdicts as algorithm* **??** *which is proven to implement $e^n \models \varphi \mathcal{U}_J \psi$ in [22].*

Algorithm 6: UNTIL Operation: $\varphi \mathcal{U}_{[lb,ub]} \psi$

```
   Init: τ_↓ψ = τ_prevψ = τ_out = 0, τ_min = −1
 1 if T_φ.τ > τ_min and T_ψ.τ > τ_min then
 2 |   τ_min = min(T_φ.τ, T_ψ.τ);
 3 |   if ⌜⌟ of T_ψ.v occurs then
 4 |   |   τ_↓ψ = τ_prevψ + 1;
 5 |   end
 6 |   τ_prevψ = T_ψ.τ;
 7 |   if T_ψ holds then
 8 |   |   result = (true, τ_min − lb);
 9 |   else if T_φ does not hold then
10 |   |   result = (false, τ_min − lb);
11 |   else if τ_min ≥ (ub − lb) + τ_↓ψ then
12 |   |   result = (false, τ_min − ub);
13 |   end
14 |   if result.τ ≥ τ_out then
15 |   |   τ_out = result.τ + 1;
16 |   |   return result;
17 |   end
18 end
```

Proof (Proof of Theorem 5). Suppose that for a given input stream there exists two MLTL $\mathcal{U}$-operator observers monitoring $\varphi \mathcal{U}_{[lb,ub]} \psi$: ξ using Algorithm **??** and ξ' Algorithm 6, that return different truth values. There are only three locations in Algorithm 6 where a result is decided (lines 11, 13, and 15) and therefore one of these cases in ξ' must not match the behavior of ξ.

Line 11 Requires ψ to be true, the trivially case of $\mathcal{U}$. The timestamp accounts for the delay between the verdict and the interval of interest, as indicated by the lower bound.

Line 13 Requires both φ and ψ to be false, the trivially false case of $\mathcal{U}$. The timestamp justification is the same as the previous case.

Line 15 Requires φ to hold, and ψ to be false, also data past the length of the interval from the last falling edge of ψ is available. This is the case when the interval elapses without ψ becoming true. The timestamp for this verdict is the time of the data, offset by the length of the interval.

Since ξ' cannot return a verdict that differs from ξ, then if ξ' will always eventually return a verdict then Theorem 5 holds.

B Proofs of New MLTL Observer Complexity

Theorem 6 (Space Complexity of Asynchronous Observers). *The respective asynchronous observer for a given MLTL specification φ implemented with SCQs as per algorithms 3-6 has the same space complexity, in terms of memory bits, as the observers in [22] of $(2 + \lceil \log_2(n) \rceil) \cdot (2 \cdot m \cdot p)$, where m is the number of binary operators, p is the worst-case delay of a single predecessor chain in $\mathrm{AST}(\varphi)$, and $n \in \mathbb{N}_0$ is the time stamp it is executed.*

Proof (Proof of Theorem 6). Only three modifications to the method presented in [22] have been made:

1. CSE: In the worst case, no elimination occurs and the AST is unmodified, thus no queues are eliminated.
2. SCQ: The shared connection queues require no additional space over the synchronization queues used in [22].
3. $\mathcal{U}-$operator: Algorithm 6 does not improve queue size over algorithm **??** in the worst case.

In the worst case, all modifications to the method in [22] have identical space complexity, therefore the bound remains unchanged.

Theorem 7 (Time Complexity of Asynchronous Observers). *The asynchronous observer for a given MLTL specification φ implemented with SCQs as per algorithms 3-6 has the same time complexity as the observers in [22] of $\mathcal{O}\Big(\log_2 \log_2 \max(p, n) \cdot d\Big)$, where p is the maximum worst-case-delay of any observer in $\mathrm{AST}(\varphi)$, d the depth of AST(φ), and $n \in \mathbb{N}_0$ the time stamp it is executed.*

Proof (Proof of Theorem 7). Only three modifications to the method presented in [22] have been made:

1. CSE: In the worst case, no elimination occurs therefore no instructions are eliminated.
2. SCQ: By the nature of the streaming input data, SCQ usage will be indistinguishable from the synchronization queues used in the original.
3. Until: In the worst case, the incoming data is already in lockstep such that no benefit from the interval computation of algorithm 6 is seen and complexity is equivalent to algorithm **??**.

In the worst case, all modifications to the method in [22] have identical time complexity, therefore the bound remains unchanged.