Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2[⋆]

Brian Kempa[0000−0003−2239−4218], Pei Zhang[0000−0001−9560−2175], Phillip H. Jones[0000−0002−8220−7552], Joseph Zambreno[0000−0002−0566−5744], and Kristin Yvonne Rozier[0000−0002−6718−2828]

Iowa State University, Ames IA 50010, 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 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. We further define an optimization enabled by our new algorithms and implement the new version of R2U2. We encode specifications describing real-life faults occurring onboard Robonaut2 using Mission-time Linear Temporal Logic (MLTL) and detail our process of specification debugging, validation, and refinement. We deployed 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 · Temporal Logic specification · Steam-based Runtime Verification · MLTL · R2U2

1 Introduction

Safe integration of autonomous robotic systems necessitates embedding runtime checks into specialized, domain-specific platforms designed for utility and efficiency. Robonaut2 (R2) [8] 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 when deployed to complex,

[⋆] Supported by NASA ECF NNX16AR57G and NSF CAREER Award CNS-1552934.

dynamic environments like the ISS. In R2's case, position sensors within rotational joints can return faulty position data indistinguishable from high-torque data to the 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 a 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) [20] 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 [7]. ROSRV [11] and Declarative Robot Safety (DeRoS) [1] integrate RV into ROS by generating ROS nodes that monitor 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]. But, `EgMon` is a software implementation that would not work in R2's architecture: it reads previously-logged inputs, adds significant overhead, and allows a high level of false positives that would be unacceptable. Formal verification of autonomous robot systems is a burgeoning research area; see [16] for a survey.

R2 requires a hardware-based solution with consideration for resource constraints; Table 1 summarizes four options. IoTA considers some resource constraints in implementing RV, but for software [5]. RVS is the only other modern hardware RV implementation; its limited expression language only monitors the

Table 1. Comparison of Hardware Monitor Tools.

Tool	P2V[15]	BusMOP[18]	HW-CBMC [17]	R2U2 [21]
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

internal behavior of a real-time operating system and RVS requires resynthesis to change monitored properties [27]. The Realizable Responsive Unobtrusive Unit [22] (R2U2) is the only RV tool that starts an encoding with the resource constraints and then optimizes the verification configuration 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; these features proved useful in other case studies on real aerospace systems [9, 26, 24, 25]. However, an initial trial proved that even R2U2's most optimized configuration would not fit; *no currently existing RV tools were capable of on-board, real-time fault detection for R2's knee joint.* We would have to build a custom tool to bridge that gap.

Using R2U2 as a base, we designed and proved correct new observer-encoding algorithms suitable for R2 and developed an optimization until we were able to deploy RV on the real Robonaut2 knee joint successfully. A previously unnoticed fault syndrome prevented the simple original specification from operating correctly. Our revised specification set provided the required accuracy but utilized significantly more resources. The new specifications only fit on the FPGA because of the optimization enabled by our new encoding, resulting in successful fault disambiguation.

This paper contributes: (1) a significant revision of all asynchronous future-time MLTL monitor encodings of [21] with new proofs of correctness; (2) an optimization 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. section 3 gives the new monitoring encodings with correctness proofs, then implementions with optimization appear in section 4, along with experimental performance characterizations. Section 5 covers embedding of these observers on Robonaut2 and development of specifications for fault disambiguation. Finally, lessons learned and opportunities for future work appear in section 6.

2 Preliminaries

R2U2 uses Mission-time LTL (MLTL) for future-time temporal specification [21, 14]. MLTL is a bounded variant of MTL [2] with closed natural number interval bounds on each temporal operator.

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 [14]). Let π be a finite computation and let $|\pi|$ represent the length of π (where $|\pi| < +\infty$). Every position $\pi[i]$ (where $i \geq 0$) is an assignment over $2^{\mathcal{AP}}$; let π_i represent the suffix of π starting from position i (including i).

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:*

- $\pi, i \models p \in \mathcal{AP}$ *iff* $p \in \pi[0]$,
- $\pi, i \models \neg\varphi$ *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 every* $k < j$, $k \in [i + lb, i + ub]$, $\pi, k \models \varphi_1$.

2.1 Abstract Syntax Tree and Execution Sequence

As a reconfigurable monitor, R2U2 uses external specification data. This allows changes to the specification without recompilation or resynthesis of the R2U2 engine. R2U2 executes runtime reconfigurable specifications by constructing an *Abstract Syntax Tree* (AST) of logical observers wherein each node produces an *execution sequence* as output that can be used by other nodes in the tree.

Definition 3. *(Execution Sequence) (adapted from [21]) 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 element T_φ. 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 represents a non-empty set of verdicts: 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 $false$. Intuitively, if $T_\varphi^0 = (\mathsf{false}, 0)$ and $T_\varphi^1 = (\mathsf{true}, 5)$ then T_φ^1 represents that φ holds from $\tau = 0$ through $\tau = 1$.*

2.2 Propagation Delay

Each temporal operator in MLTL is accompanied by a closed natural integer bound, $I = [lb, ub]$. A node of the AST is decidable at a given time when sufficient information is known to determine the verdict at that time. As these observers 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 time between when a set of propositions $\pi[i]$ (i.e., input) arrives at φ, and when it is possible to know if $\pi, i \models \varphi$ (i.e., output). A node's* worst propagation delay *(**wpd**) is the maximum propagation delay it can experience, and the minimum value is the* best propagation delay *(**bpd**).*

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 φ be defined as follows:*

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

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

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

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

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 single-writer, single-reader buffers of the original operators [21]. The SCQ-backed operators enable a further implementation optimization, 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 [29].

3.1 Shared Connection Queues

A SCQ is a circular buffer of verdict tuples with one write pointer and one or more read pointers that buffers verdicts from a child subformula to be read by multiple parent expressions. These supplant the synchronization queues utilized

Algorithm 1: Function: write()

```
input: execution sequence tuple: $T_\varphi$
if $SCQ[wr\_ptr].v == T_\varphi.v$ then //Setup aggregation
    $wr\_ptr--$;
end
$SCQ[wr\_ptr] \leftarrow T_\varphi$;
$wr\_ptr++$;
```

Algorithm 2: Function: read()

```
input  : $rd\_ptr$, $\tau_e$
output: $T_\varphi$ or $Empty$
if $rd\_ptr == wr\_ptr$ then
    return $Empty$;
end
$T_\varphi \leftarrow SCQ[rd\_ptr]$;
while $T_\varphi.\tau < \tau_e$ do
    $rd\_ptr++$;
    if $rd\_ptr == wr\_ptr$ then
        return $Empty$;
    end
    $T_\varphi \leftarrow SCQ[rd\_ptr]$;
end
return $data$;
```

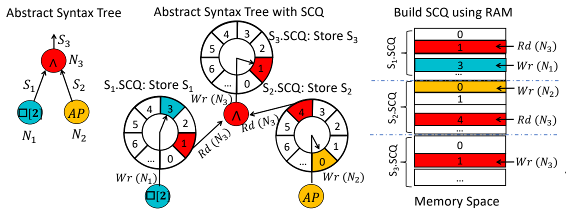


Fig. 1. Representative AST fragment showing a $\wedge$ operation (N_3) and it's children/inputs. The output of all three nodes are buffered with SCQs where N_3 holds read pointers to S_1 and S_2. The SCQs are arranged linearly in memory as shown.

in [21]. Shared Ring Buffers, which are similar structures from multi-threading software (e.g., [28]), 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* [21], wherein the latest tuple's timestamp is overwritten by subsequent timestamp values if their truth values (and therefore verdicts) 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 $\{(\mathsf{true}, 10), (\mathsf{false}, 16)\}$.

Reading from a non-empty SCQ returns verdicts with monotonically increasing time steps. This prevents reprocessing verdicts a reader has already observed. To enforce monotonic reads, the last timestamp seen by each reader is tracked in the variable τ_e. When reading, the first verdict found after the read pointer with a timestamp greater or equal to τ_e is returned; else, it returns empty. The circular structure 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.

Queue Sizing. The required buffer size for each observer is computed a priori by recursively sizing the SCQs in its MLTL AST based on the best and worst-case delays of their subexpressions. We call the maximum number of verdicts a SCQ can hold the depth, and the individual positions we call slots. We compute the size of the output queue for a node g with sibling nodes $\mathbb{S}_g$ that share a common parent with:

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

The minimum queue size is one because even with no delay the verdict must be passed between nodes. Sizing queues based on the worst-case delay guarantees that verdicts are consumed by the parent nodes before the write pointer recirculates, overwriting old data. This safely bounds the memory required to evaluate each node in the worst case. Software RV monitors can use these precomputed bounds to avoid dynamic allocation when desired. In hardware RV, we build SCQs using 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 [21] used one of two observers for $\Box_{[lb,ub]}$ depending on the bounds, this encoding only uses one observer. Negation (algorithm 3) returns all input verdicts after inverting

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
```

Algorithm 4: UNTIL Operation: $\varphi\mathcal{U}_{[lb,ub]}\psi$

```
Init: $\tau_{\downarrow\psi} = \tau_{prev\psi} = \tau_{out} = 0$, $\tau_{min} = -1$
if $T_\varphi.\tau > \tau_{min}$ and $T_\psi.\tau > \tau_{min}$ then
    $\tau_{min} = \min(T_\varphi.\tau, T_\psi.\tau)$;
    if ⌉⌊ of $T_\psi.v$ occurs then
        $\tau_{\downarrow\psi} = \tau_{prev\psi} + 1$;
    end
    $\tau_{prev\psi} = T_\psi.\tau$;
    if $T_\psi$ holds then
        $result = (\mathsf{true}, \tau_{min} - lb)$;
    else if $T_\varphi$ does not hold then
        $result = (\mathsf{false}, \tau_{min} - lb)$;
    else if $\tau_{min} \geq (ub - lb) + \tau_{\downarrow\psi}$ then
        $result = (\mathsf{false}, \tau_{min} - ub)$;
    end
    if $result.\tau \geq \tau_{out}$ then
        $\tau_{out} = result.\tau + 1$;
        return result;
    end
end
```

Algorithm 5: AND Operation: $\varphi \wedge \psi$

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

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

```
Init: $m_\uparrow = 0$, $\tau_{min} = -1$
if $T_\varphi.\tau > \tau_{min}$ then
    if ⌋⌈ of $T_\varphi$ occurs then
        $m_\uparrow = \tau_{min} + 1$;
    end
    $\tau_{min} = T_\varphi.\tau$;
    if $T_\varphi$ holds and $T_\varphi.\tau \geq \max((ub - lb) + m_\uparrow, ub)$
     then
        return $(\mathsf{true}, T_\varphi.\tau - ub)$;
    else if $T_\varphi.\tau \geq lb$ then
        return $(\mathsf{false}, T_\varphi.\tau - lb)$;
    end
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 $\mathcal{N}.iSCQ\_1$

their truth values. Until (algorithm 4) tracks the falling edges of ψ and the latest seen timestamp of φ. If ψ is true or φ is false, then the output is trivially true or false, respectively. Additionally, failure by elapsed time is detected from the time since the falling edge of ψ. And (algorithm 5) considers 4 cases to eagerly evaluate false verdicts. If both inputs are true, the output is true up to the smaller input timestamp. If both inputs are false, the output is false up to the largest observed timestamp; this is classic Boolean “short-circuiting” behavior. Otherwise, the verdict is false up to the timestamp of the false input. The Globally operator (algorithm 6) counts time stamps since the last rising edge. It outputs *true* when the length of the *true* signal meets or exceeds the duration of the interval. Operators with non-zero lower bounds can be treated as zero-bounded operators of equivalent duration by offsetting the returned timestamps. This shift equivalence (a separate operator in [21]) is directly embedded in our new encoding.

3.3 Correctness of New MLTL Observers

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

Theorem 1 (Correctness of the $\Box$-operator). *Let execution sequence $\langle T_\varphi \rangle$ be the output of Algorithm 6 with interval $[lb, ub]$ over computation π. Algorithm 6 correctly implements $\Box_{[lb,ub]}\varphi$, that is $\forall i\ T_\varphi = (i, \mathsf{true}) \Leftrightarrow \pi, i \models \Box_{[lb,ub]}\varphi$.*

Proof (Proof of Theorem 1). In [21] the following equivalence with the globally operator is developed: $\forall i : (i - lb \in [\tau, \tau + ub - lb] \rightarrow \pi, i \models \varphi) \Leftrightarrow \pi, \tau \models \Box_{[lb,ub]}\varphi$

Since $(ub - lb) \not< 0$, $\Box_{lb,ub}\varphi$ holds at τ iff $\pi, i \models \varphi$ where $(i - ub) \leq \tau \leq (i - lb)$. From these conditions, we see that $\Box_{[lb,ub]}\varphi$ is equivalent to the verdicts $\Box_{[0,ub-lb]}\varphi$ shifted back by lb, i.e., φ must hold for $ub - lb$ or longer.

$\Leftarrow$: In Algorithm 6, a rising edge of φ is detected by line 2-5 which account for aggregation. If φ has held for at least $ub - lb$, then line 7 returns a true verdict, shifted by ub. Otherwise, a false verdict is returned (line 9) which eagerly fails all time steps unable to meet the condition $\pi, i \models \varphi$ for $(i - lb) \geq \tau$. The check on line 8 prevents premature output of false verdicts on initialization.

$\Rightarrow$: True verdicts are only returned from line 7, which requires φ to have held for at least $ub - lb$ per the check in line 6. False verdicts are only returned from line 9, which requires φ has not sufficiently held (line 9) but sufficient information is available (line 8).

Verdicts are returned iff they satisfy the original equivalence. $\square$

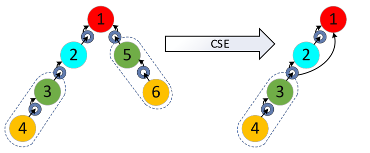
Due to size, proofs for Algorithm 4 and Algorithm 5 are available online.[1]

4 Optimization and Experimental Performance Analysis

In a set of MLTL formulas, repeated sub-expressions can generate redundant observer instructions, needlessly increasing required queue space and execution time. Compilers use *common subexpression elimination* (CSE) [6] to share the output of repeated expressions. Figure 3 demonstrates the application of CSE to MLTL ASTs. CSE is not possible with single-reader buffers and requires SCQs with multiple readers. Algorithm 7 removes duplicate branches of a formula's AST. Sub-expressions are eliminated both within and between formulas.

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

$load\ \mathcal{AP}(a0)$
$f_{\Box[0,3]}(S0)$
$f_{\neg}(S1)$
$load\ \mathcal{AP}(a0)$
$f_{\Box[0,3]}(S3)$
$f_{\mathcal{U}[0,5]}(S2, S4)$



$$\Big(\overbrace{\neg}^{2}\overbrace{\Box_{[0,3]}}^{3}\overbrace{a_0}^{4}\Big)\overbrace{\mathcal{U}_{[0,5]}}^{1}\Big(\overbrace{\Box_{[0,3]}}^{5}\overbrace{a_0}^{6}\Big)$$

$load\ \mathcal{AP}(a0)$
$f_{\Box[0,3]}(S0)$
$f_{\neg}(S1)$
$f_{\mathcal{U}[0,5]}(S2, S1)$

Fig. 3. Example of CSE on an MLTL formula where nodes 3 and 5 have identical children. On the left is the AST and resulting R2U2 institution representing the above formula. The AST and instructions on the right are produced by applying CSE. 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, we tested the 10,000 random MLTL benchmark formulas used in [14] by converting them to observer trees and queue configurations with and without CSE enabled. The benchmark set formulas vary in length, number of variables, and probability of choosing the $\mathcal{U}$-operator.

The R2U2 configuration compiler is a single-threaded Python application and was run in parallel (12 instances at a time) on a 2019 MacBook Pro with a i9-9880H Intel CPU and 32 GB system RAM. 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. In total, the 10,000 runs across 12 parallel processes completed in under 15 seconds wall clock time.

Algorithm 7: `CSE`(T, S)

```
Input  : AST Tree: $T$, Set: $S = \{(label, node)\}$
Output: optimized AST: T
// Recuse through $T$ in post-order
Let $N = \text{root}(T)$
if leftChild$(N) \neq \emptyset$ then CSE(leftChild$(N), S)$
if rightChild$(N) \neq \emptyset$ 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 $(N.\text{label}, \bullet) \notin S$ then
    // Unique subtree, store reference
    $S = S \cup (N.\text{label}, N)$
else
    // Common subtree, link existing
    Let $M \in T$ such that $(N.\text{label}, M) \in S$
    $T = T \cup (\text{parent}(N), M)$
    $T = T - (\text{parent}(N), N)$
end
```

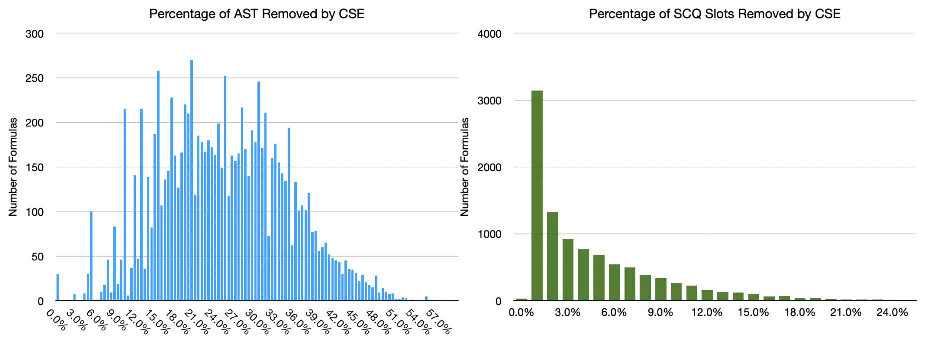


Fig. 4. Reduction in AST nodes (left) and SCQ slots (right) as percentage of unoptimized size. The y-axis indicates the number of formulas from MLTL benchmark set [14].

Over the whole set, the number of R2U2 observer nodes dropped 27.06% from $788,095$ to $574,822$ and the total queue slots required decreased 4.28% from $42,300,361$ to $40,491,507$. Adding CSE to the R2U2 configuration compiler increased the configuration time 10.25% from 57.66 to 63.57 total 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. The 24% AST node reduction over random formulas gives hope for similar or greater reductions in encoding real specifications due to the greater repetition in human-written specifications. The queue space reduction saved a median of 100 slots per formula, which is important as BRAMs are less plentiful on FPGAs. The benchmark formulas use large operator intervals, which limit our SCQ reductions by requiring sufficient space for their worst-case propagation delays. We expect to see increased queue space savings on formulas with shorter intervals; we will explore this in future work.

5 Theory into practice: Robonaut2

Robonaut2's legs are comprised of series-elastic actuators with torsional springs, causing external force to register on the internal position sensors [19]. Precise measurements of the spring displacement cap applied force, affording near-human dexterity while remaining safe in confined spaces with astronauts [3]. After deployment, NASA observed that the *Absolute Position Sensors* (APSs) sporadically initialize incorrectly by ≈ 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 R2 cannot distinguish it from sensor drift. To increase availability and resilience, Robonaut2 must be able to automatically trigger corrective action without compromising existing safety guarantees.

Constraints. 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, the left-over space on the FPGA is limited and additional runtime verification logic must not impact the response time of the existing controller. Additionally, the system's remote deployment limits available debug information. We derived our initial specification from a plain-language description of the fault mechanics by subject-matter experts while awaiting a real trace.

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 [22]; 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 [4]. Since the FPGA is the nexus of the actuator's sensors, all required data can be accessed on-chip.

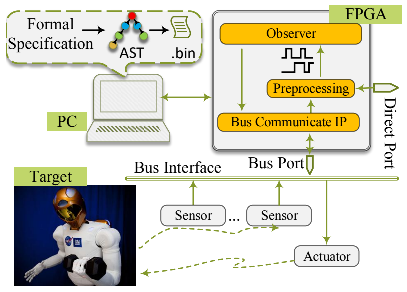


Fig. 5. R2U2 observers, encoded on the FPGA, monitor internal sensor values passed over the R2 control bus.

5.1 Embedding Runtime Verification

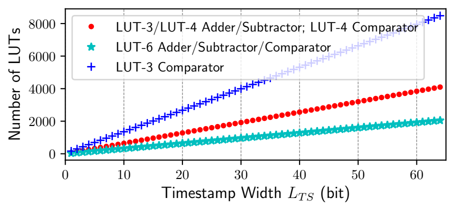


Fig. 6. LUT resource usage for timestamp length L_{TS}. Growth is linear, but the rate is dependent on FPGA process type.

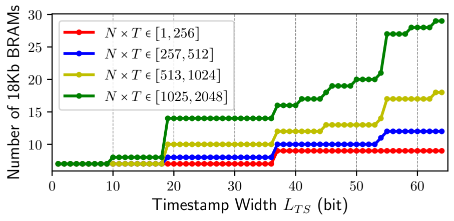


Fig. 7. BRAM resource usage for timestamp length L_{TS} where $N \times T$ is the number of binary operators N, times $T = \max n.wcd \ \forall n \in N$ i.e. total queue space.

R2U2 allows runtime configuration of the observer specifications, while the size and duration limits of these specifications are 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 [22]. Memory requirements are driven by queue depth and timestamp length. We can compute the minimum required resources of a given specification, or the maximum parameters that fit within a given design. Figures 6 and 7 show the scaling of FPGA look-up-tables (LUTs) and BRAM required as timestamp width is increased, respectively. We selected a max queue depth of 20 and timestamp length of 16-bits 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 out 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.

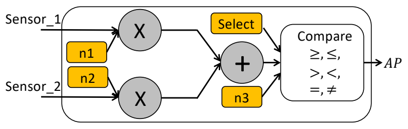


Fig. 8. R2U2 atomic checker. Orange blocks are configurable online.

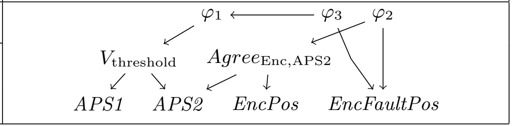
Boolean Checker Construction. R2U2's runtime-configurable Atomic Checkers, shown in Fig. 8, convert the native sensor format to Boolean variables used in specification. For example, the *EncPos* sensors value indicates the rotation degree of the motor. Robonaut2's native encoder format is a 19-bit integer, 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 while **sensor_2** always returns 1. In this configuration, **n1** is the scale factor, and **n2** is the configurable offset. The final *AP* output is the Boolean result of 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 due to sensor outliers, we ensure states hold for at least three timesteps before reporting a fault. Robonaut2's existing logic sets an "encoder fault position" signal when the encoder and APS1 disagree. Our MLTL specifications reason over the APS1 position, APS2 position, encoder position, and encoder fault position sensor inputs; see Table 2. The corresponding R2U2 configuration for this set of specifications 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.

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

Table 2. Fault disambiguation specification – revision 1

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

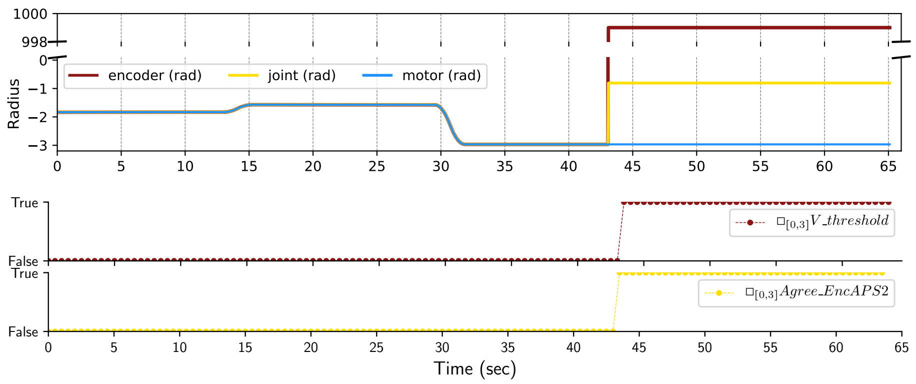


Fig. 9. Ground R2: APS Fault

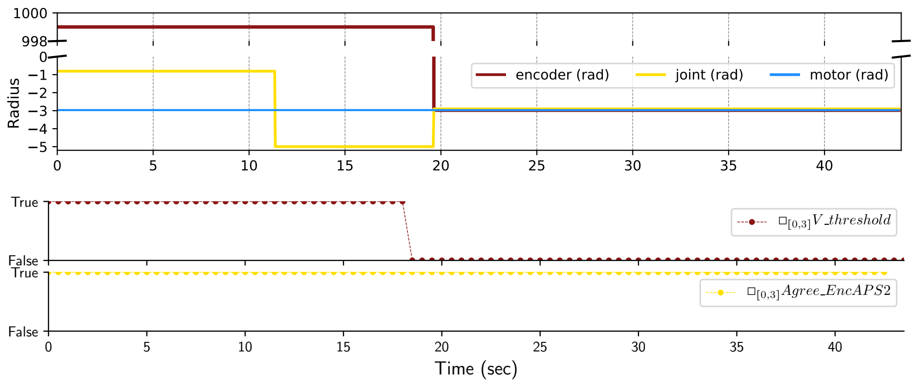


Fig. 10. Ground R2: Unsuccessful Recovery

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 emergency stop. Additionally, the Boolean $V_{\text{threshold}}$ correctly detects that we are not in the failure mode of interest after time 19. This data reveals an implicit assumption that encoder values freeze during a fault.

Revision. With our new insight 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 are: (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 a 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 the signal from APS1 is in region n (and similarly with b_n and APS2). Now the temporal observers can check if each APS is in the same region before and after the encoder jump. The size of n dictates the maximum rotation distance before triggering a region change. We select n such that the maximum rotation is about 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 rotation during a fault. Also, encoder range errors do not register in the *EncPos* signal stream when an actuator experiences nominal joint rotation across a boundary, further preventing false positives. Table 3 lists the MLTL and signal layer specification. The final R2U2 configuration requires 154 instructions, 140 SCQs, and 196 SCQ slots with a max depth of 3 without CSE. CSE reduces this to 100 instructions, 86 SCQs, and 142 SCQ slots with a max depth of 3. The 33% reduction in instructions shows the impact of CSE optimization on human-written specifications that necessarily contain repeated references to important subsystems. While CSE results in a 38% reduction in SCQ quantity, the significance of this reduction is that the number of SCQs in the unoptimized R2U2 configuration crosses a power-of-two boundary, requiring 8 bits to encode but the same specification requires only 7 bits with CSE enabled. When embedding hardware monitors, bus size increases account for multiplicative jumps in resource requirements (as in Fig. 7). Our RV specification would not fit in Robonaut2's knee's available FPGA space without SCQ-based encodings reduced by CSE.

Verification. Following the best-practices for specification debugging established in [23], we checked each specification, its negation, and the conjunction of all specifications for satisfiability. We utilized the MLTL SMT solver from [14] to prove the specifications were both satisfiable and falsifiable. Finally, we played

Table 3. Fault disambiguation specification – revision 2

R2U2 Configuration	
Bus Values	Temporal Formulas
APS1: Position [rad]	$\varphi_n = (a_n \wedge \neg e) \wedge \diamond_{[1,2]}(\neg a_n \wedge e) \rightarrow \mathrm{APS1}_{\mathrm{Fault}} \;\; \forall n \in [0,5]$
APS2: Position [rad]	$\varphi_{m+6} = (b_m \wedge \neg e) \wedge \diamond_{[1,2]}(\neg a_m \wedge e) \rightarrow \mathrm{APS2}_{\mathrm{Fault}} \; \forall m \in [0,5]$
EncPos: Position [rad]	Observer Tree
Signal Processing	$$\begin{tikzcd} & & \varphi_{[0,5]} \arrow[dll] \arrow[d] & \varphi_{[6,11]} \arrow[dll] \arrow[dl] \\ a_{[0,5]} \arrow[d] & b_{[6,11]} \arrow[d] & e \arrow[d] & \\ \mathit{APS1} & \mathit{APS2} & \mathit{EncPos} & \end{tikzcd}$$
$e = \mathrm{EncPos} > 100$	
$a_n = \pi(\frac{n}{6} - 1) < \mathrm{APS1} < \pi(\frac{n+1}{6} - 1) \forall n \in [0,5]$	
$b_n = \pi(\frac{n}{6} - 1) < \mathrm{APS2} < \pi(\frac{n+1}{6} - 1) \forall n \in [0,5]$	

back all available recorded traces of both faulty and nominal operation through the real hardware, with our final R2U2 configuration running, to check that we successfully catch the fault with no false positives during nominal operation.

6 Conclusion

We have successfully embedded R2U2 to provide trusted fault-disambiguation for automatic mitigation. Our new encodings enabled CSE optimization, a crucial step in meeting the resource limitation challenges of the R2 platform. Importantly, the techniques presented in sections 3 and 4 are not exclusive to this application or to R2U2, but could be ported to other RV algorithms, tools, and application domains.

Working with FPGA limitations provided important lessons on the relation between specification complexity and hardware resources. In Fig. 6 LUT requirements scale linearly with timestamp length; however, transistor count (and therefore chip space and power) scales 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 a specification's encoding, even with the same amount of LUTs and BRAM free. For a hardware R2U2 deployment, BRAM will probably be the limiting resource. This may not be true for other RV engines, but it's the price of reconfigurability, which allows RV to be embedded, certified, and deployed flexibly, and which was a requirement of the R2 team.

Future Work. In the current implementation, we utilize the equivalence relations in section 2 to represent full MLTL semantics; next we plan to implement direct encodings, e.g., for the $\mathcal{R}$ operator. Encoding every operator directly would reduce the number of negations in the AST and therefore, the amount of SCQ space required. We will then investigate additional design-time optimizations to the AST.

On the application side, we are working toward distributing specifications across RV monitors on multiple FPGAs. This extension has the potential to increase the number of specifications we can monitor on a given platform, 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.: Real-time logics: Complexity and expressiveness. Information and Computation **104**(1), 35–77 (1993)
3. 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
4. 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
5. 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)
6. 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)
7. 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)
8. 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
9. Geist, J., Rozier, K.Y., Schumann, J.: Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In: Proceedings of the 14th International Conference on Runtime Verification (RV14). vol. 8734, pp. 215–230. Springer-Verlag (September 2014)
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. 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). LNCS, vol. 11562, pp. 3–22. Springer, New York, NY, USA (July 2019). https://doi.org/https://doi.org/10.1007/978-3-030-25543-5_1

15. 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)
16. 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)
17. 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)
18. 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)
19. 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
20. 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)
21. 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)
22. 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
23. Rozier, K., Vardi, M.: LTL satisfiability checking. International Journal on Software Tools for Technology Transfer (STTT) **12**(2), 123 – 137 (March 2010). https://doi.org/DOI 10.1007/s10009-010-0140-3, http://dx.doi.org/10.1007/s10009-010-0140-3
24. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In: Proceedings of the 15th International Conference on Runtime Verification (RV15). Springer-Verlag, Vienna, Austria (September 2015)
25. Schumann, J., Moosbrugger, P., Rozier, K.Y.: Runtime Analysis with R2U2: A Tool Exhibition Report. In: Proceedings of the 16th International Conference on Runtime Verification (RV15). Springer-Verlag, Madrid, Spain (September 2016)
26. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. International Journal of Prognostics and Health Management (IJPHM) **6**(1), 1–27 (June 2015)
27. 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)

28. 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)
29. Zhang, P., Zambreno, J., Jones, P.H., Rozier, K.: Model predictive runtime verification for embedded platforms with real-time deadlines (Under submission, 2020)