R2U2 Version 3.0: Re-imagining a Toolchain for Specification, Resource Estimation, and Optimized Observer Generation for Runtime Verification in Hardware and Software *

Chris Johannsen[1], Phillip Jones[1], Brian Kempa[1], Kristin Yvonne Rozier[1], Pei Zhang[2]

[1] Iowa State University
{cgjohann, phjones, bckempa, kyrozier}@iastate.edu
[2] Google
piz@google.com

Abstract. R2U2 is a modular runtime verification framework capable of monitoring sets of specifications in real time and in resource-constrained environments. Such environments demand that a runtime monitor be fast, easily integratable, accessible to domain experts, and have predictable resource requirements. Version 3.0 adds new features to R2U2 and its associated suite of tools that meet these needs including a new front-end compiler that accepts a custom specification language, a GUI for resource estimation, and improvements to R2U2's internal architecture.

1 Tool Overview

R2U2 (Realizable Responsive Unobtrusive Unit) is a modular framework for hardware (FPGA) and software (C and C++) real-time runtime verification (RV). R2U2 runs *online*, during system execution, with minimal overhead. (It also runs *offline*, over simulated data streams or recorded data logs.) R2U2 is *stream-based*; given a runtime requirement φ and an input computation π of sensor and software values at each timestamp i, R2U2 returns the verdict (`true` or `false`) for all i as to whether $\pi, i \models \varphi$. We call this output stream an *execution sequence* [34]; it is a stream of two-tuples $\langle verdict, time \rangle$ for every time i. R2U2 encodes specifications as *observers* (a set of which we call a *configuration*) via an optimized algorithm with published proofs of correctness, time, and space [34,20,18].

Fig. 1 depicts a standard R2U2 workflow. To integrate R2U2 into a target system, we first need a validated set of runtime requirements. Given the system's resource constraints, the Configuration Compiler for Property Organization (C2PO) creates an optimized encoding of the input set of requirements as an R2U2 configuration. Users can swap configurations monitored by R2U2 at runtime, during system execution, based on

* This work was funded by NSF CAREER Award CNS-1552934, NASA-ECF NNX16AR57G, NASA Cooperative Agreement Grant #80NSSC21M0121, and NSF:CPS Award 2038903. Thanks to the NASA Lunar Gateway Vehicle System Manager team for novel feature requests.

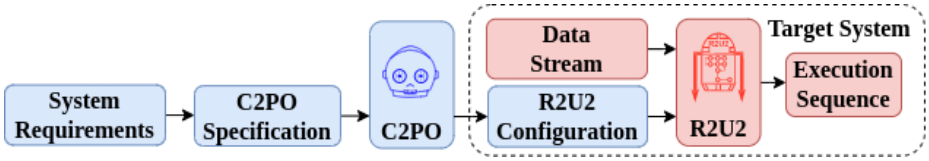

Fig. 1. Workflow for verifying a specification using R2U2. Red shaded boxes denote runtime components and blue shaded boxes denote design-time components. Note that for validation, the runtime components can run offline, e.g., by replacing the data stream with a log file of simulated data. Users formalize their system requirements as MLTL formulas within a C2PO specification, use C2PO to generate an R2U2 configuration, then monitor the verdicts R2U2 outputs based on the configuration and data stream.

system state, mission phase, or to upgrade the specification version – all without recompiling and redeploying the R2U2 engine, a key feature for systems that require onerous code change certifications, or e.g., systems that need to be launched into space and then dynamically updated as their hardware degrades.

R2U2 fills the unique gap in the RV community described by its name [39]:

REALIZABILITY R2U2 analyzes generic, re-usable specifications in Mission-Time Linear Temporal Logic (MLTL) [34,20], a variant of LTL with closed integer-bounded intervals on the temporal operators. MLTL excels at capturing requirements conceptualized as timelines, as is common in aerospace operational concepts, e.g., [1,11,45]. At its core, R2U2 specifications combine either a future-time or past-time MLTL formula with simple signal comparators [34]. New optional extensions provide additional features, such as simple set-level reasoning [5]. R2U2's hardware implementation, written in VHDL, avoids overburdening limited computing resources by utilizing Field Programmable Gate Arrays (FPGAs) to monitor in parallel with the system under absolute timing guarantees. R2U2's two software implementations avoid hardware integration and software instrumentation challenges at the cost of (minimal) compute resources on the host system and are designed to be suitable for different environments. The C version forgoes memory allocation and bounds checking to provide fast deterministic results for real-time controllers under stringent certifiability criteria; alternatively, the C++ version makes full use of dynamic memory, templates, and runtime checks for maximum flexibility without monitor tuning. Additionally, the implementations differ significantly in architecture to provide fault independence. The three monitor implementations enable on-board (embedded) and on-ground execution, integration with multiple human-machine interaction paradigms, cross-validation, or triple modular redundancy voting strategies to increase system trust.

RESPONSIVENESS R2U2 provides two levels of responsiveness. At a system level, runtime reconfiguration of the monitor without a lengthy re-compilation (and re-certification) process keeps R2U2 responsive to the system's needs even as the mission, platform, or requirements evolve. At a specification level, R2U2's asynchronous (event-triggered) observers provably report both `true` and `false` verdicts (rather than only reporting property violations) in the first timestamp where there is sufficient information to evaluate $\pi, i \models \varphi$, thus monitoring integrity, safety, and security requirements in real-time. Since the monitor's response time is a function of the specification and known a priori, higher-level autonomous sys-

tem health and decision-making controllers can rely on R2U2 verdicts to provide a tight bound on mitigation triggering or other reactive behaviors.

UNOBTRUSIVENESS R2U2's multi-architecture, multi-platform design enables effective runtime verification while respecting crucial unobtrusiveness properties of embedded systems, including functionality (no change in behavior), certifiability (bounded time and memory under safety cases), timing (no interference with timing guarantees), and tolerances (respect constraints on size, weight, power, bandwidth, and overhead). R2U2 obeys unobtrusiveness constraints, provably fitting into tight resource limits and operational constraints frequently encountered in space missions. It can operate without code instrumentation or insight into black-box subcomponents such as ITAR, restricted, or closed-source modules [29].

User Base. After an extensive survey of all currently-available verification tools, NASA's Lunar Gateway Vehicle System Manager (VSM) team selected R2U2 for operational verification [8,9,10]; R2U2 is currently operating in the NASA core Flight System/core Flight Executive (cFS/cFE) [28] VSM environment. R2U2 is embedded in the space left over on the FPGA controlling NASA's Robonaut2's knee to provide real-time fault disambiguation [18], interfacing via the Robot Operating System (ROS) [31]. R2U2 is running on a UAS Traffic Management (UTM) system [5], where it recently detected a flight-plan timing fault. JAXA is running R2U2 on a 2021 autonomous satellite mission with a requirement for a provable memory bound of 200KB [30]. R2U2 recently verified a CubeSat communications system [24], an open-source UAS [16], a sounding rocket [15], and a high-altitude balloon [23]. The CySat-I satellite uses R2U2 for autonomous fault recovery [2]. In the recent past, R2U2 was used in NASA's Autonomy Operating System (AOS) for UAS [22] (where it flew on NASA's S1000 octocopter [21]), the NASA Swift UAS [43,34,13,36], and the NASA DragonEye UAS [44,41]. R2U2 aided in NASA embedded system battery prognostics [42] and a case study on small satellites and landers [35]. R2U2 has also proven useful for monitoring and diagnosis of security threats on-board NASA UAS like the DragonEye [40,27]. R2U2 was cataloged by the user community in a 2018 taxonomy of RV tools [12,39], and appeared in a 2020 Institute of Information Security (ETH Zürich, Switzerland) case study [33]. R2U2 is open-source, dual licensed under MIT[3] and Apache-2.0.[4] and is available at `https://r2u2.temporallogic.org/`.

2 Compiler and Specification Language

Specification is a notoriously difficult aspect of RV [37]; verification results are only meaningful if the input specifications are correct and complete with respect to the system requirements. An RV engine is only usable if system engineers can *validate* that it monitors its given requirements as they expect, so they can clearly explain when and why different RV verdicts occur. In consultation with outside groups using R2U2 on real systems [14,30,8], we developed a new specification language and an accompanying formula-set compiler. The language's and compiler's features make specifications easier to read and write, improving user productivity and easing validation to address the challenges of specification in RV.

[3] `https://choosealicense.com/licenses/mit/`

[4] `https://choosealicense.com/licenses/apache-2.0/`

Feature	Previous Syntax [39]	C2PO Syntax

Declare Signal	`Temp = 0;`
	Fix name to signal index
		`INPUT`
		`  Temp: float;`
		Declare name/type, signal index handled separately

Define Macro	N/A	`DEFINE`
		`  Temp_Limit = 97;`
		Improves readability and maintenance

Define Struct	N/A	`STRUCT`
		`  Alarm = { T: float; };`
		Enables data organization

Atomic Checker	`OVERTEMP = float(Temp) > 97;`
	In-lined constants, signal type determined by function name
		`ATOMIC`
		`  OVERTEMP = Temp > Temp_Limit;`
		All declared names available, uses known signal types

MLTL Formula	`G[0,3] !OVERTEMP;`	`FTSPEC`
		`  G[0,3] !OVERTEMP;`
		Requires temporal tense declared (`FTSPEC` or `PTSPEC`)

Table 1. Overview of changes to the R2U2 specification syntax for a basic temperature limit requirement, where $Temp$ is located at index 0 of the input signal vector. This is not an exhaustive comparison but covers directly equivalent features, while Fig. 2 and the remainder of Section 2 detail new capabilities.

2.1 New Specification Language

Previous versions of R2U2 used a specification language derived from the implementation of the hardware runtime engine. While sufficiently expressive for the creation of R2U2 configurations, it utilized a restricted syntax that supported only basic MLTL operators and single-operator expressions over non-Boolean data types. Writing specifications that are transparent and easy to validate could be difficult without in-depth knowledge of R2U2's architecture [17].

The new SMV-inspired [26] specification language allows users the option to write specifications more naturally with support for compound expressions over complex data types including sets and C-like structs as well as sections for defining structs, variables, macros, and MLTL formulas. C2PO supports Boolean, struct, and parametric set types with configurable integer and floating point types. To run R2U2 in software, users select a C standard type for each of the integer and float types e.g., an unsigned 16-bit integer (`uint16_t`) and double-precision floating point (`double`). If targeting hardware (FPGA implementation), users can configure integer and float types to a bit-width supported by the target system. Table 1 presents a comparison between the old [39] and new syntaxes and Fig. 2 presents a sample file for monitoring a request-handling system.

```
1  STRUCT
2    Request: { state: int; time_active: float; };
3    Arbiter: { ReqSet: set<Request>; };
4
5  INPUT
6    st0, st1, st2, st3: int;
7    ta0, ta1, ta2, ta3: float;
8
9  DEFINE
10   WT := 0; GR := 1; RJ := 2; -- WAIT, GRANT, REJECT
11
12   rq0 := Request(st0, ta0); rq1 := Request(st1, ta1);
13   rq2 := Request(st2, ta2); rq3 := Request(st3, ta3);
14
15   Arb0 := Arbiter({rq0, rq1}); Arb1 := Arbiter({rq2, rq3});
16   ArbSet := {Arb0, Arb1};
17
18 FTSPEC
19   (rq0.time_active - rq1.time_active) < 10.0 &&
20   (rq1.time_active - rq0.time_active) < 10.0;
21
22   foreach(arb: ArbSet)(
23    foreach(rq: arb.ReqSet)(
24      (rq.state == WT) U[0,5] (rq.state == GR || rq.state == RJ)
25    )
26   );
```

Fig. 2. Sample C2PO specification file using structs (lines $2-3$, $12-13$), sets (lines 3, $15-16$), and set aggregation operators (lines $22-23$). The specification on lines $19-20$ captures the English requirement, "The active times for rq_0 and rq_1 shall differ by no more than 10.0 seconds," and the specification on lines $22-26$ captures the English requirement, "For each request r of each arbiter in $ArbSet$, r's status shall be GRANT or REJECT within the next 5 seconds and until then shall be WAITING."

To create an R2U2 configuration, C2PO generates an Abstract Syntax Tree (AST) representation of the input, performs type checking, applies optimizations and rewriting rules, then outputs the corresponding R2U2 configuration. R2U2 does not use automata to encode temporal logic observers (as reported erroneously elsewhere [12]); instead C2PO traverses the AST to produce assembly-like imperative evaluation instructions for the R2U2 monitor to executed at runtime.

In order to meet the demands of a wide range of systems, R2U2 Version 3.0 includes many optional features that are specific to one of the three implementations that can be enabled during system integration. For example, the Booleanizer module computes arbitrary non-Boolean expressions in the C implementation of R2U2, but this feature is not an option in the C++ or hardware implementations. C2PO allows users to enable or disable such features according to the capabilities of their target systems and chosen R2U2 implementation.

2.2 Assume-Guarantee Contract Support

Assume Guarantee Contracts (AGCs) provide a template for structuring and validating complex requirements in aerospace operational concepts [3]. AGCs feature a guard or trigger clause called the "assumption" and a system invariant called the "guarantee;" they have been used to structure both English and formal (e.g., temporal logic) requirements by projects including the NASA Lunar Gateway Vehicle System Man-

ager [10]. R2U2 V3.0 now directly supports AGCs with an input syntax for expressing AGCs in C2PO and an output format for R2U2 that provides granular interpretation of verdicts, as presented in [17]. The input syntax for declaring an AGC is `assumption => guarantee` where the semantics for this logical implication provides three distinct cases: the AGC is "inactive" if the assumption is false, "true" if both the assumption and guarantee are true, and "false" otherwise. When the optional AGC feature is enabled, R2U2 produces three-valued verdicts to represent the state of the AGCs in a clear format; otherwise R2U2 interprets logical implications in the standard way (where `false` $\rightarrow$ `true` results in the verdict `true` rather than `inactive`).

2.3 Set Aggregation

A common pattern in real-world specifications applies an identical formula to various input signals, such as testing all temperature sensors for an overheat condition. A naive encoding of these specifications in MLTL can be excessively large to the point of obscuring intent while providing ample opportunity for copy-paste errors, typos, or incomplete updates to variables – all of which are difficult for humans to spot during validation. C2PO mitigates this issue by supporting set aggregation operators that compactly encode these expressions as sets of streams with a predicate applied to each element [14].

To illustrate, consider the specification in Fig. 2. The direct encoding of this specification without the "foreach" operator is

```
(rq0.status == W) U[0,5] (rq0.status == G || rq0.status == R) &&
(rq1.status == W) U[0,5] (rq1.status == G || rq1.status == R) &&
(rq2.status == W) U[0,5] (rq2.status == G || rq2.status == R) &&
(rq3.status == W) U[0,5] (rq3.status == G || rq3.status == R)
```

Contrast this with the more compact encoding using the "foreach" operator on lines 22−26 in Fig. 2. The latter retains the intent of the English-level requirement while being semantically equivalent to the direct encoding. This concise representation both eases validation by improving readability and reduces the potential for errors by avoiding replicated values that require simultaneous updates.

2.4 Common Subexpression Elimination

C2PO uses an AST as the intermediate representation of its input and can therefore use optimization techniques common in compiler design such as Common Subexpression Elimination (CSE) [6]. Similarly to applying the isomorphism elimination rule for Binary Decision Trees [4], Common Subexpression Elimination (CSE) prunes all but one instance of any identical AST subtrees, reusing the result from that subtree for monitoring multiple requirements without wasting memory and execution time by representing it redundantly. Analysis of CSE on randomly-generated MLTL requirements resulted in a speed-up of 37% and required 4.3% less memory [18]. We expect larger savings in human-authored requirement specifications, however, due to reuse of both common specification patterns and structures in the underlying system. For example, a non-trivial subexpression might represent a system's confidence in its navigational fix and many specifications might depend on the navigation state, thus re-using this subexpression.

3 Resource Estimation GUI

As R2U2's user base expands, so does the variance in the domain expertise of these specification authors; R2U2 V3.0 therefore enables resource-aware requirements specification by users without experience with the performance trade-offs of syntactically

different but semantically equivalent temporal logic encodings. The R2U2 Configuration Explorer is a web application that provides visual feedback from C2PO about the resource costs of specifications, e.g., in the form of MLTL formulas; see Fig. 3. With a short feedback loop on critical parameters like execution time, memory, and relative formula size, all a user needs to understand is what resources are available on their target system (not R2U2 itself) to write performant specifications that fit the available resources.

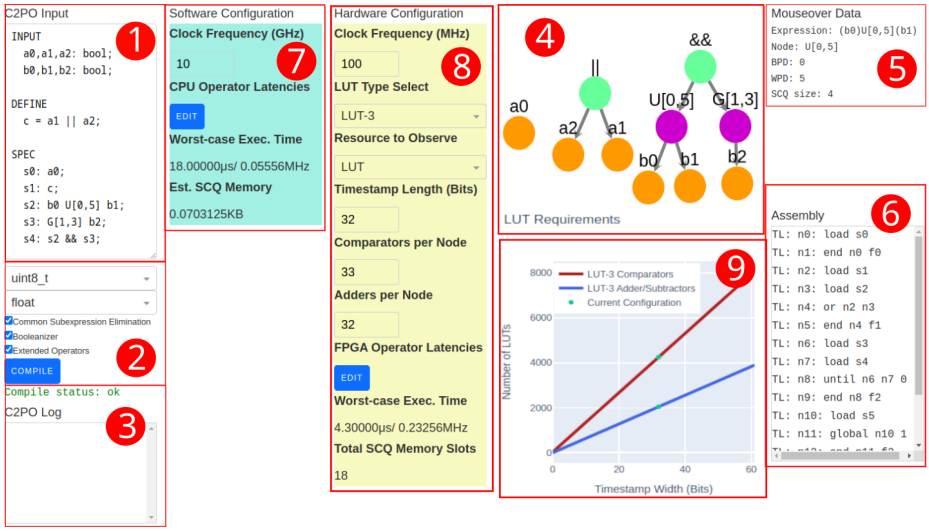

Fig. 3. R2U2 Configuration Explorer web application: 1) C2PO specification input; 2) C2PO options; 3) C2PO output; 4) AST visualization; 5) AST node data; 6) R2U2 instruction; 7) C engine speed and memory calculator; 8) FPGA speed and size calculator; 9) FPGA design size vs maximum timestamp value.

3.1 C2PO Feedback

Feedback from C2PO (elements $1-6$ in Fig. 3) allows users to visualize the intermediate representation of a given input specification as well as the effects of optimizations and options on their final R2U2 configurations. Properties such as the memory required to represent specifications with differently-sized temporal intervals, or syntactically different but functionally similar checks, can be unintuitive for users to compute on the fly. The AST visualization provides transparency into this process for users unfamiliar with R2U2's implementation via an interactive web-based interface suited to experimentation with different variations of a possible specification.

3.2 Software Resource Calculator

The software resource calculator (element 7 in Fig. 3) provides users of the R2U2 software implementations with an estimate of the time and memory required to evaluate one time step of a specification in the worst case.

Software Worst-case Execution Time The highly optimized nature of R2U2's software implementations makes runtime performance highly dependent on the target platform's architecture, C/C++ compiler version, and `make` environment factors; e.g., the length of the current working directory name can impact cache alignment. We use a

simplified computing model to provide an estimation of the computing speed based on the number of CPU cycles required for each operation on the target platform. Users can edit these clock cycle values in the GUI, e.g., to test for platform-specific latencies. The estimated worst-case execution time (WCET) in software W_{sw} of an AST node g is:

$$W_{sw}(g) = \sum_{c \in \mathbb{C}_g} (W_{sw}(c)) + Cycles(g.type) \tag{1}$$

where $\mathbb{C}_g$ are the children nodes of g and $Cycles$ is a dictionary mapping AST node types to a corresponding number of clock cycles. For instance, $Cycles(\wedge) = 10$ cycles by default.

Software Memory Requirements R2U2 uses Shared Connection Queues (SCQs) to store verdict-timestamp pairs for each node in the AST. SCQs are single-writer, many-reader circular buffers that buffer the results of dependent temporal expressions that might not be evaluated at the same timestamp. The total SCQ size for a specification is the total number of SCQ slots required by the specification multiplied by the size of one slot. The required number of SCQ slots for a node g is:

$$size(g.Queue) = max(max\{s.wpd \mid \forall s \in \mathbb{S}_g\} - g.bpd, 0) + 1 \tag{2}$$

where $g.Queue$ is the output SCQ of g, $s.wpd$ is the worst-case propagation delay of node s, $s.bpd$ is the best-case propagation delay of node s, and $\mathbb{S}_g$ is the set of sibling nodes of g. The propagation delays of a node represent the minimum and maximum number of time steps needed to evaluate the node and are defined recursively in Definition 4 of [18]. Intuitively, a node requires enough memory such that its results will not be overwritten before they are consumed by a parent node. The total SCQ memory of an AST is the sum of the sizes of SCQs of all nodes in the AST.

SCQ memory is an estimation of the actual total memory usage, but is typically the largest and most constraining memory type, e.g., as compared to instruction or pointer memory. The R2U2 C implementation statically fixes all memory sizes in advance to avoid dynamic allocation, so the SCQ sizing feedback is useful for: (1) selecting an initial size based on expected usage and; (2) verifying a configuration will fit on a deployed monitor with a fixed SCQ limit.

3.3 Hardware Resource Calculator

The hardware resource calculator (elements $8 - 9$ in Fig. 3) provides estimations for hardware WCET (W_{hw}), total SCQ memory slots, and a graph for visualizing estimated FPGA resource requirements - Look-Up Tables (LUT) and Block RAMs (BRAM). Required resources depend on the type of FPGA architecture. The GUI accepts clock rate, LUT-type, timestamp length, and node sizing as parameters to better match the estimate to a target platform. This approach was validated on Virtex-5 and Zynq7000 FPGA platforms as well as the ACTEL ProASIC3L used for Robonaut2 in [18].

Hardware Worst-case Execution Time The GUI computes the estimated W_{hw} using a more precise method than in Section 3.2 by taking into account SCQ usage during execution. The R2U2 hardware implementation's estimated worst-case execution time (W_{hw}) of an AST node g is:

$$W_{hw}(g) = \sum_{c \in \mathbb{C}_g} (W_{hw}(c)) + Latency_{init}(g.type) \\ + Latency_{eval}(g.type) * \sum_{c \in \mathbb{C}_g} (size(c.Queue)) \tag{3}$$

where $Latency_{init}$, $Latency_{eval}$ are dictionaries mapping AST node types to microsecond latencies corresponding to the initial and evaluation times of the node respectively. The multiplication accounts for evaluation of each buffered input from the child node, up to the queue size in the worst case.

Hardware Memory Requirements The hardware resource calculator provides the explicit number of SCQ slots required for the collection of specifications in the specification set (aka *configuration*) using Formula 2 and summing sizes required for all AST nodes.

FPGAs use BRAMs to implement an R2U2 monitor's SCQ memory, where the size and number of ports of the BRAMs limit the queue depth of the BRAMs. To compute the required number of BRAMs, let d be the total SCQ size, w be the bit width of each verdict-timestamp pair, w_{max} be the widest bit width the BRAM can accommodate, and $D(w)$ be the maximum queue depth of a BRAM with verdict-timestamp pair bit width w. The required number of cascaded BRAMs is:

$$N_{BRAM}(w, d) = \lceil \frac{d}{D(w_{max})} \rceil * mod(w, w_{max}) + \lceil \frac{d}{D(rem(w, w_{max}))} \rceil \tag{4}$$

Hardware LUT Requirements Each R2U2 operator requires a constant number of comparator and adder/subtractor LUTs, configured by the user in the GUI. The GUI accounts for scaling based on the LUT type and uses the bit width of each verdict-timestamp pair w to estimate total LUT usage. The total number of required comparator LUTs (N_{cmp}) and adder/subtractor LUTs (N_{add}) are:

$$N_{cmp}(w) = \begin{cases} 4 * w & \text{if LUT-3} \\ 2 * w & \text{if LUT-4} \\ w & \text{if LUT-6} \end{cases} \qquad N_{add}(w) = \begin{cases} 2 * w & \text{if LUT-3 or LUT-4} \\ w & \text{if LUT-6} \end{cases}$$

4 Runtime Engine Improvements

To better serve mission-critical systems that must satisfy strict flight certification requirements (such as NASA's VSM [8,9,10]), we have made a number of improvements to the internal architecture of the C version of R2U2 that provide memory assurances and flexibility as well as extended computational abilities. Fig. 4 depicts this updated architecture.

Static Memory Arenas The R2U2 V3.0 C version uses only statically-allocated memory. This avoids the many pitfalls of allocating memory (slow allocator calls, fragmentation, leaks, out-of-memory errors, etc.) and guarantees the amount of memory required for the entire execution of R2U2 up front. Additionally, many mission-critical systems

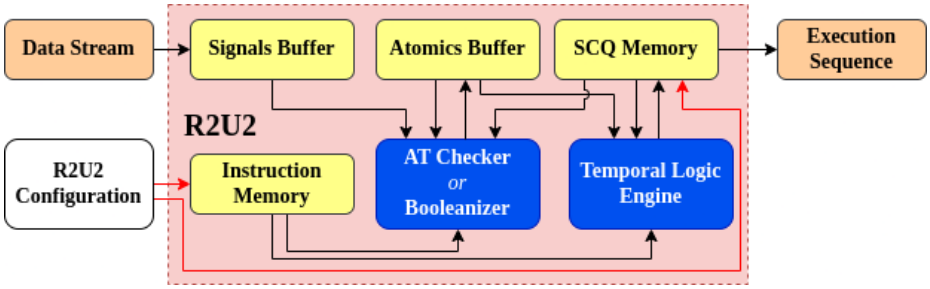

Fig. 4. Internal architecture of an R2U2 monitor. Orange boxes are streams of data, yellow boxes are memory arenas, and blue boxes are modules. Arrows entering and exiting blue boxes denote read and write relationships respectively. The red arrows denote relationships that are only active upon startup i.e., when R2U2 populates instruction memory and configures SCQ memory.

either do not have or do not permit dynamic memory allocation, e.g., to satisfy requirements for flight certification [32]. R2U2 now runs unmodified on these platforms as well as traditional systems.

Each type of memory (yellow boxes of Fig. 4) has a predefined "arena" with a maximum size set during integration of the monitor with the target platform. When a user loads an R2U2 configuration, R2U2 fills the slots of these arenas in sequence until the arena is full.

Monitor Type Parameterization Complimentary to the switch to static memory, the internals of the reasoning engine are now fully parameterized. A single header file allows users to adjust maximum values, bit widths, and even internal types. Proper tuning has performance benefits, but crucially allows users to fit R2U2 to use the exact amounts of resources available on a target system. For example, limiting the size of the gaps between timestamps, e.g., in cases where the specification will be either reset frequently or evaluated infrequently, allows more SCQs to fit in the same amount of memory permitting larger formula sets with functionally similar behavior.

Arbitrary Data Flow R2U2 initially worked as a stack of engines, at each timestamp passing results from the Atomic Checker (AT) to the Temporal Logic engine (TL), then passing the TL verdicts through the Bayesian Network (BN) layer to produce that timestamp's verdict [34]. Now, R2U2 can connect these engines in any order. This simplifies configuration generation from the perspective of C2PO, enabling arbitrary ordering of instructions. Atomic checker properties can now accept results of temporal logic formulas as input, for example, without adding a confusing step delay in the verdict stream.

AT Checker Extended Mode The C version of the atomic checker has an extended mode allowing for additional comparisons and filters beyond the standard hardware-compatible set. In extended mode, the atomic checker produces Boolean "atomics" from conditionals, where each conditional compares the result of a filter to either a constant or another input signal. Filters are predefined functions such as simple data type casts (bool, int, float, etc.) or mathematical functions like rate, moving average, or absolute angle difference. For example:

• `a5 := abs_diff_angle(s3,105) < 50;` checks if the absolute difference between the data of signal 3 and the value 105 when treated as angles is below 50.

• `a43 := int(s32) == s33;` checks that the values of signals 32 and 33 are in agreement when treated as integers.

Booleanizer The R2U2 V3.0 C implementation includes a new general-purpose computing module that uses a three-address code representation [7] called the Booleanizer that can take the place of the AT checker. This module enables arbitrary expressions over non-Boolean data types using arithmetic, bitwise, and relational operators as well as extended set aggregation operators such as "forexactlyn" or "foratmostn" operators.

5 Discussion

R2U2's toolchain now provides an effective means by which to formalize, validate, and verify system requirements in real time, giving users control and transparency of the memory and feature set of their target-specific monitors. We have combined the collection of capabilities from previously-published R2U2 case studies into one modular, centralized implementation that we have rigorously evaluated for correctness (e.g., using [38,19]).

C2PO and its new specification language enable higher-level abstractions for users that make the specification development process faster, more transparent, and less reliant on a deep understanding of R2U2's underlying algorithms. The new GUI front-end allows up-front specification design and resource usage estimation by system designers so that users can rapidly prototype specifications before downloading and using R2U2. These improvements make specifying, validating, and monitoring system requirements easier and more accessible to the systems that stand to benefit most from RV. Since specification is the biggest bottleneck to formal methods and autonomy [37], this is an important feature for an RV engine.

It is now much easier to integrate R2U2 into production environments, like NASA cFS/cFE[25,28] or ROS[31], due to the unified front end compiler, expanded engine capabilities, and better user tooling. Recently R2U2 has launched on several real-life, full-scale air and space missions, largely enabled by these advancements. This major upgrade lays a solid foundation for expanded RV capabilities and integration into a wider array of missions and embedded architectures.

References

1. J.C.Ryan, M.L.Cummings, N.Roy, A Banerjee, A.Schulte. "Designing an Interactive Local and Global Decision Support System for Aircraft Carrier Deck Scheduling." AIAA Infotech, 2011.
2. Aurandt, A., Jones, P., Rozier, K.Y.: Runtime Verification Triggers Real-time, Autonomous Fault Recovery on the CySat-I. In: Proceedings of the 14th NASA Formal Methods Symposium (NFM 2022). Lecture Notes in Computer Science (LNCS), vol. 13260. Springer, Cham, Caltech, California, USA (May 2022). https://doi.org/10.1007/978-3-031-06773-0_45
3. Badger, J.M., Strawser, P., Claunch, C.: A distributed hierarchical framework for autonomous spacecraft control. In: 2019 IEEE Aerospace Conference. pp. 1–8. IEEE (2019)
4. Bryant, R.: Graph-based algorithms for Boolean-function manipulation. IEEE TC **C-35**(8), 677–691 (1986)
5. Cauwels, M., Hammer, A., Hertz, B., Jones, P., Rozier, K.Y.: Integrating Runtime Verification into an Automated UAS Traffic Management System. In: Proceedings of DETECT: international workshop on moDeling, vErification and Testing of dEpendable CriTical systems. Communications in Computer and Information Science (CCIS), Springer, L'Aquila, Italy (September 2020). https://doi.org/10.1007/978-3-030-59155-7_26
6. Cooper, K., Eckhardt, J., Kennedy, K.: Redundancy elimination revisited. In: 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 12–21 (2008)
7. Cooper, K.D., Torczon, L.: Engineering a compiler. Elsevier (2011)
8. Dabney, J.B., Badger, J.M., Rajagopal, P.: Adding a verification view for an autonomous real-time system architecture. In: Proceedings of SciTech Forum. p. Online. 2021-0566, AIAA (January 2021). https://doi.org/https://doi.org/10.2514/6.2021-0566
9. Dabney, J.B.: Using assume-guarantee contracts in autonomous spacecraft. Flight Software Workshop (FSW) Online: `https://www.youtube.com/watch?v=zrtyiyNf674` (February 2021)
10. Dabney, J.B., Rajagopal, P., Badger, J.M.: Using assume-guarantee contracts for developmental verification of autonomous spacecraft. Flight Software Workshop (FSW) Online: `https://www.youtube.com/watch?v=HFnn6TzblPg` (February 2022)
11. Erzberger, H., Heere, K.: Algorithm and operational concept for resolving short-range conflicts. Proc. IMechE G J. Aerosp. Eng. **224**(2), 225–243 (2010). https://doi.org/10.1243/09544100JAERO546, `http://pig.sagepub.com/content/224/2/225.abstract`
12. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: International Conference on Runtime Verification. pp. 241–262. Springer (2018)
13. 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)
14. Hammer, A., Cauwels, M., Hertz, B., Jones, P., Rozier, K.Y.: Integrating runtime verification into an automated uas traffic management system (July 2021). https://doi.org/10.1007/s11334-021-00407-5
15. Hertz, B., Luppen, Z., Rozier, K.Y.: Integrating runtime verification into a sounding rocket control system. In: Proceedings of the 13th NASA Formal Methods Symposium (NFM 2021) (May 2021), available online at `http://temporallogic.org/research/NFM21/`
16. Johannsen, C., Anderson, M., Burken, W., Diersen, E., Edgren, J., Glick, C., Jou, S., Kumar, A., Levandowski, J., Moyer, E., Roquet, T., VandeLoo, A., Rozier, K.Y.: OpenUAS Version 1.0. IEEE, Athens, Greece (Virtual) (June 2021)

17. Kempa, B., Johannsen, C., Rozier, K.Y.: Improving usability and trust in real-time verification of a large-scale complex safety-critical system. Ada User Journal (2022)
18. Kempa, B., Zhang, P., Jones, P.H., Zambreno, J., Rozier, K.Y.: Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2. In: Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). pp. 196–214. Lecture Notes in Computer Science (LNCS), Springer, Vienna, Austria (September 2020), http://research.temporallogic.org/papers/KZJZR20.pdf
19. Li, J., Rozier, K.Y.: MLTL Benchmark Generation via Formula Progression. In: Proceedings of the 18th International Conference on Runtime Verification (RV18). pp. 426–433. Springer-Verlag, Limassol, Cyprus (November 2018)
20. 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, Springer, New York, NY, USA (July 2019)
21. Lowry, M., Bajwa, A., Quach, P., Karsai, G., Rozier, K., Rayadurgam, S.: Autonomy Operating System for UAVs. Online: https://nari.arc.nasa.gov/sites/default/files/attachments/15%29%20Mike%20Lowry%20SAEApril19-2017.Final_.pdf (April 2017)
22. Lowry, M., Bajwa, A.: Autonomy Operating System (AOS) for UAVs. Proposal Presentation, NASA Ames Research Center, Moffett Field, California (June 2015)
23. Luppen, Z., Jacks, M., Baughman, N., Hertz, B., Cutler, J., Lee, D.Y., Rozier, K.Y.: Elucidation and Analysis of Specification Patterns in Aerospace System Telemetry. In: Proceedings of the 14th NASA Formal Methods Symposium (NFM 2022). Lecture Notes in Computer Science (LNCS), vol. 13260. Springer, Cham, Caltech, California, USA (May 2022). https://doi.org/10.1007/978-3-031-06773-0_28
24. Luppen, Z.A., Lee, D.Y., Rozier, K.Y.: A Case Study in Formal Specification and Runtime Verification of a CubeSat Communications System. In: SciTech. AIAA, Nashville, TN, USA (January 2021)
25. McComas, D.: NASA/GSFC's Flight Software Core Flight System. In: Flight Software Workshop. Southwest Research Institute, San Antonio, Texas (November 2012)
26. McMillan, K.L.: The SMV Language. Cadence Berkeley Labs pp. 1–49 (1999)
27. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems pp. 1–31 (April 2017). https://doi.org/10.1007/s10703-017-0275-x
28. NASA: core Flight System (cFS) Background and Overview. Online: https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf (2014)
29. NASA: NASA Export Control Program Operations Manual. Online: https://nodis3.gsfc.nasa.gov/NPR_attachments/N_AII_2190_0001.pdf (2015)
30. Okubo, N.: Using R2U2 in JAXA program. Electronic correspondence (November–December 2020), series of emails and zoom call from JAXA to PI with technical questions about embedding R2U2 into an autonomous satellite mission with a provable memory bound of 200KB
31. Open Robotics: Robot Operating System (ROS). Online: https://www.ros.org/ (2021)
32. Radio Technical Commission for Aeronautics: DO-333 – formal methods supplement to DO-178C and DO-278A (2011), https://www.rtca.org/content/standards-guidance-materials
33. Raszyk, M., Basin, D., Traytel, D.: Multi-head monitoring of metric dynamic logic. In: International Symposium on Automated Technology for Verification and Analysis. pp. 233–250. Springer (2020)

34. 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)
35. Rozier, K.Y.: R2U2 in Space: System and Software Health Management for Small Satellites. In: Spacecraft Flight Software Workshop (FSW) (December 2016), https://www.youtube.com/watch?v=OAgQFuEGSi8, https://www.youtube.com/watch?v=OAgQFuEGSi8
36. Rozier, K.Y., Schumann, J., Ippolito, C.: Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS. Technical Memorandum NASA/TM-2015-218817, NASA, NASA Ames Research Center, Moffett Field, CA 94035, USA (May 2015)
37. Rozier, K.Y.: Specification: The biggest bottleneck in formal methods and autonomy. In: Proceedings of 8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016). LNCS, vol. 9971, pp. 1–19. Springer-Verlag, Toronto, ON, Canada (July 2016). https://doi.org/10.1007/978-3-319-48869-1_2
38. Rozier, K.Y.: On the evaluation and comparison of runtime verification tools for hardware and cyber-physical systems. In: Proceedings of International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES). vol. 3, pp. 123–137. Kalpa Publications, Seattle, WA, USA (September 2017), https://easychair.org/publications/paper/877G
39. 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://easychair.org/publications/paper/Vncw
40. 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)
41. 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 (RV16). Springer-Verlag, Madrid, Spain (September 2016)
42. Schumann, J., Roychoudhury, I., Kulkarni, C.: Diagnostic reasoning using prognostic information for unmanned aerial systems. Proceedings of the 2015 Annual Conference of the Prognostics and Health Management Society (PHM2015) (2015)
43. 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. In: Proceedings of the 2013 Annual Conference of the Prognostics and Health Management Society (PHM2013). pp. 381–401 (October 2013)
44. 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)
45. Zhao, Y., Rozier, K.Y.: Formal specification and verification of a coordination protocol for an automated air traffic control system. Science of Computer Programming Journal **96**(3), 337–353 (December 2014)