A Formal Verification Framework for Runtime Assurance

J Tanner Slagel[1], Lauren M. White[1], Aaron Dutle[1], César A. Muñoz[1], and Nicolas Crespo[1]

NASA Langley Research Center, Hampton, VA 23666, USA
j.tanner.slagel@nasa.gov

Abstract. The simplex architecture is an instance of Runtime Assurance (RTA) where a trusted component takes control of a safety-critical system when an untrusted component violates a safety property. This paper presents a formalization of the simplex RTA framework in the language of hybrid programs. A feature of this formal verification framework is that, for a given system, a specific instantiation can be created and its safety properties are guaranteed by construction. Instantiations may be kept at varying levels of generality that allow for black box components, such as ML/AI-based controllers, to be modeled. The framework is written in the Prototype Verification System (PVS) using Plaidypvs, an embedding of differential dynamic logic in PVS. As a proof of concept, the framework is illustrated on an automatic vehicle braking system.

Keywords: Runtime Assurance · Hybrid Programs · Plaidypvs · PVS

1 Introduction

Runtime Assurance (RTA) is a design-time architecture for safety-critical systems where an internal monitor takes action upon detecting a violation of a property [2]. The simplex architecture is an instance of RTA where control of the overall system is handed to a trusted controller when an untrusted one violates a safety property [9]. Simplex RTA is emerging as a method for allowing AI/ML and other unverified software to be integrated into safety-critical applications.

This paper presents a formalization of a simplex RTA framework in the Prototype Verification System (PVS) [7] using an embedding of differential dynamic logic (dL) called Plaidypvs [10]. A novel feature of this framework is that it can be instantiated at different levels of abstraction. This feature allows for the formal verification of a system with an untrusted black box component, such as an AI/ML controller.

1.1 Runtime Assurance

Runtime *verification* is the use of a *monitor* to check safety properties of a system at runtime [4,6]. If a property is violated, the monitor may send a signal

to perform some action or to alert a user. Runtime *Assurance* is the design-time integration of runtime verification into a system to provide some guarantee on the overall system.

Arguably, the most common application of Runtime Assurance is in the *simplex* architecture [9]. In this architecture, a system has an *advanced* controller (AC) and a *reversionary* controller (RC). The system is allowed to operate with the AC until a runtime monitor detects that some property has been violated and then the RC takes over. Assuming that the monitor can detect improper functioning with enough time for the RC to correct the impending problem, and that the RC is trusted, this use of RTA allows for the integration of untrusted — but possibly more performant — controllers in a safe way. This is of particular interest with the rise of AI/ML technology and the desire to integrate it in safety-critical systems like aircraft. To this end, ASTM and NASA have each published guidelines on the use of RTA in such systems [1,2]. Due to the ubiquity of the simplex RTA framework, the term RTA will be used to represent a simplex architecture in the remainder of the paper.

This paper does not address the many difficulties in deploying RTA to an industrial-level system [3]. Instead, the focus is on the formal verification of the simplex RTA *framework* in the language of hybrid programs. Employing the Plaidypvs formalization in PVS allows for the verification of the general framework and then, by specializing some components of the hybrid program, to verify instances of the framework while keeping the untrusted component essentially a black box.

1.2 Plaidypvs

Plaidypvs (**P**roper**l**y **A**ssured **I**mplementation of **D**ifferential Dynamic Logic for H**y**brid **P**rogram **V**erification and **S**pecification) [10][1] is a formal embedding of Differential Dynamic Logic [8] that allows for the formal specification of, and reasoning about, hybrid programs within the PVS interactive theorem prover. Hybrid programs are used to model hybrid systems, i.e., systems with both continuous and discrete behavior, which often arise in safety- and mission-critical applications [5].

In Plaidypvs, reasoning can be done on the executions of a hybrid program using universal and existential quantifiers denoted by *allruns* $[\cdot]$ and *someruns* $\langle\cdot\rangle$ respectively. For a given hybrid program α and a Boolean expression P on environments, $[\alpha]P$ (respectively, $\langle\alpha\rangle P$) asserts that every (respectively, some) run of the hybrid program α ends at a value that satisfies P.

Hybrid programs are syntactically defined as a datatype $\mathcal{H}$ in Plaidypvs according to the grammar

$$\alpha ::= \mathbf{x} := \ell \mid \mathbf{x}' = \ell \,\&\, P \mid ?P \mid x := * \mid \alpha_1; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^*$$

[1] Plaidypvs is available as part of the NASA PVS library at https://github.com/nasa/pvslib/tree/master/dL.

where $\mathbf{x} := \ell$ is a discrete assignment, $\mathbf{x}' = \ell$ is a differential system symbolizing a continuous evolution, and P is a Boolean expression. The expression $?P$ represents a check of the Boolean expression P, $x := *$ is an arbitrary assignment of the variable x, the expression $\alpha_1; \alpha_2$ represents sequential execution of two subprograms, $\alpha_1 \cup \alpha_2$ symbolizes a nondeterministic choice between two subprograms, and finally α_1^* represents a fixed but unknown number of repetitions of a hybrid program.

Plaidypvs uses a predicate called the dL-sequent denoted $\Gamma \vdash \Delta$, where Γ and Δ are lists of Boolean expressions. This predicate is defined by the Boolean formula

$$\bigwedge_i \Gamma_i(e) \rightarrow \bigvee_j \Delta_j(e),$$

where $\Gamma_i(e)$ and $\Delta_j(e)$ represent the i-th and j-th Boolean expressions of Γ and Δ, respectively, evaluated in the environment e.

2 RTA Framework in Plaidypvs

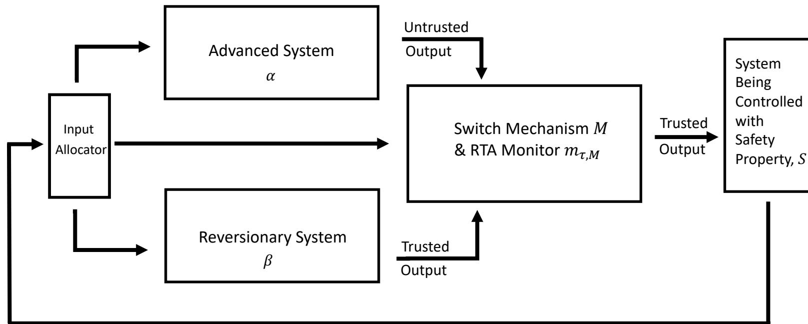

Fig. 1. The simplex RTA framework. In this work the advanced and reversionary systems are denoted by hybrid programs α and β respectively.

This section presents a general framework for RTA in Plaidypvs where the entire system, including trusted and untrusted components, are modeled as hybrid programs. In this architecture, it is assumed the monitor does not instantaneously detect when the switch condition is violated, but rather samples at least every $\tau \in \mathbb{R}_{\geq 0}$ amount of time. This assumption models real-world systems where the monitor is checked with discrete samples.

To model this sampling, the notion of a monitored hybrid program is introduced. This monitored hybrid program can be defined as a function $m_{\tau,M}$, where τ is the maximum allowed amount of time between samples and M is the switch condition. This function takes a hybrid program α and produces a hybrid

program that has the same dynamics as α but is restricted to the runs where M has been true within τ units of time of the final state. For a hybrid program α the associated monitored hybrid program is defined as:

$$m_{\tau,M}(\alpha) = \begin{cases} (?M; t := 0; (\mathbf{x}' = \ell, t' = 1 \,\&\, P \wedge t \leq \tau))^* & \text{if } \alpha = (\mathbf{x}' = \ell \,\&\, P), \\ m_{\tau,M}(\alpha_1); m_{\tau,M}(\alpha_2) & \text{if } \alpha = \alpha_1; \alpha_2, \\ m_{\tau,M}(\alpha_1) \cup m_{\tau,M}(\alpha_2) & \text{if } \alpha = \alpha_1 \cup \alpha_2, \\ (m_{\tau,M}(\alpha_1))^* & \text{if } \alpha = \alpha_1^* \\ \alpha & \text{otherwise.} \end{cases}$$

Here, it is required that the variable t does not appear in the hybrid program α.

Figure 1 shows the general RTA framework, which has been specified and verified in Plaidypvs.[2] Let the advanced and reversionary components be modeled by hybrid programs α and β, respectively, and let S be a Boolean expression describing the safety property that must be always satisfied by the RTA system. The function $m_{\tau,M}$ enforces that the hybrid program does not evolve for more than $\tau \in \mathbb{R}_{\geq 0}$ units of time without the switch condition property M being checked. In this system, the RTA framework can be written as the hybrid program:

$$((?M; m_{\tau,M}(\alpha)) \cup (?\neg M; \beta))^* . \tag{1}$$

This RTA structure enforces the switch to β when it is detected that the switch condition property M is not satisfied. Note that β is allowed to run for as long as it wants regardless of the value of M. The switch back to the advanced system α is not specified in this paper but can be defined within Plaidypvs. For instance, one enforcement of a switchback is to replace β with a monitored reversionary system $m_{\tau,\neg N}(\beta)$ checking for a switchback condition N. With the assumption that N is an invariant of β, N implies M, and the existence of a run of $m_{\tau,\neg N}(\beta)$ where N is true, it can be shown a switchback to the advanced system occurs.

Given an RTA system, a primary goal is to know that the safety property S is always satisfied, written in Plaidypvs as:

$$\left[((?M; m_{\tau,M}(\alpha)) \cup (?\neg M; \beta))^*\right] S.$$

To prove this invariant property, a general rule was specified and proven in Plaidypvs that relates the safety of the overall system to safety of its individual components:

$$\frac{\Gamma \vdash S \wedge (M \vee G) \quad S \vdash [m_{\tau,M}(\alpha)](S \wedge (G \vee M)) \quad G \vdash [\beta^*]S}{\Gamma \vdash [((?M; m_{\tau,M}(\alpha)) \cup (?\neg M; \beta))^*]S} \; (\mathbf{RTA}),$$

where $G \in \mathcal{B}$ is a user-instantiated condition that represents a property that carries over when switching between the advanced system to the reversionary

[2] The formal development presented in this paper, including examples, is available at https://github.com/nasa/pvslib/tree/master/dL/dL_RTA.

system. This switch property G is used to capture conditions that will allow the reversionary controller to satisfy S after detecting the monitor is violated.

The rule **RTA** takes the RTA system in Formula (1) and generates three subgoals. The first subgoal $\Gamma \vdash S \wedge (M \vee G)$ corresponds to the initial state of the system. The safety property S must be true to start, and either the monitoring condition M or the switch property G must hold. The second subgoal $S \vdash [m_{\tau,M}\,(\alpha)](S \wedge (G \vee M))$ is the proof condition that if the system is in a safe initial point, every monitored run of the advanced system will satisfy S (recall that a *monitored* run of α is terminated within τ of the M failing to hold), and have the property that if the monitoring condition M is not true, then the switch condition G holds— since $G \vee M \iff (\neg M \rightarrow G)$. The third subgoal $G \vdash [\beta^*]S$ requires proving that when starting from the switch condition being true, the reversionary system may run any finite number of times and the safety property S is satisfied.

3 A Simple Example

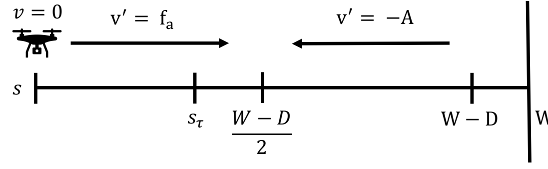

Fig. 2. A simple hysteresis controller set-up, which is an instantiation of the general RTA framework.

Consider the one-dimensional braking example in Figure 2, where a vehicle whose position and velocity are given by the variables s and v, respectively, is governed by an RTA system. The advanced controller is given by

$$\alpha := (s' := v, v' := f_a), \tag{2}$$

where $f_a \in \mathcal{R}$ is any positive acceleration function bounded by $A \in \mathbb{R}_{>0}$. Note that f_a is not specifically defined, but is a black box component with only the requirement that it is bounded by A. The reversionary braking system is given by

$$\beta := (s' := v, v' := -A). \tag{3}$$

The switching condition M is a check on the position and velocity. Given a sampling rate τ, the full RTA system is a variant of the hysteresis controller, having the form of Formula (1).

Let the vehicle start with $s = 0$ and $v = 0$, and as a safety property, require that the vehicle does not go within a given distance $D \in \mathbb{R}_{>0}$ to a wall $W \in \mathbb{R}$

(assuming that $W - D > 0$). The RTA property is then

$$s = 0,\, v = 0 \rightarrow \left[(?M; m_{\tau,M}\,(\alpha) \cup ?\neg M; \beta)^* \right] (s \leq W - D). \tag{4}$$

Using the values

$$\tau = \sqrt{\frac{W - D}{A}} \tag{5}$$

$$M = s \leq s_\tau \,\wedge\, v \leq \sqrt{A(W - D)} \tag{6}$$

$$s_\tau = \frac{(\sqrt{A(W - D)} - \tau)^2}{2}, \tag{7}$$

allows Formula (4) to be proven in Plaidypvs. In fact, the proof process itself was crucial in discovering the requirement on the maximum time between samples, τ, to guarantee the safety property for the system. Furthermore, it illuminates an important aspect of runtime verification, that the amount of drift in the system between samples must be accounted for to ensure correctness. This is shown in the term s_τ corresponding the position component of the switch condition M. This term is specified so that even in the worst case, where the sampling rate is such that the position is sampled right before M is violated and the next sample occurs τ time after the previous, the switch condition allows the reversionary controller to take over with enough time so that a distance D away from the wall is maintained.

Proving the property is done by applying the **RTA** rule with

$$G = s \leq \frac{W - D}{2} \wedge v \leq \sqrt{A(W - D)},$$

which yields the three subgoals

$$s = 0, v = 0 \vdash s \leq W - D \wedge (M \vee G) \tag{8}$$

$$s \leq W - D \vdash [m_{\tau,M}\,(\alpha)] s \leq W - D \wedge (M \vee G) \tag{9}$$

$$G \vdash [(\beta)^*]\,(s \leq W - D) \tag{10}$$

each of which are then proven within Plaidypvs.

4 Conclusion

This paper presents a general framework for RTA, which has been formalized in Plaidypvs. The formalization allows the designer of a safety-critical system to prove safety properties of the entire RTA system based on properties of its individual components. A simple braking example illustrates the use of this emerging idea. Particularly, this framework extracts and reveals requirements of the underlying system being modeled. Future work will apply this RTA framework to more complex examples in the aerospace domain, to extract safety requirements, including the delicate interplay between sample rates of sensors and monitoring

specifications. Using the temporal extension of Plaidypvs that includes the trace semantics of hybrid programs [11], examples will be developed where temporal properties are shown for the system. Additionally, the trace semantics of hybrid program will allow a rigorous connection to be made between a hybrid program and its analogous monitored hybrid program. Plaidypvs allows more complicated RTA structures to be modeled at a generic level. This could include multiple components such as secondary reversionary controller or even a system made of several simplex RTA structures, which creates the need for modeling concurrency in Plaidypvs.

References

1. ASTM International: Standard practice for methods to safely bound behavior of aircraft systems containing complex functions using run-time assurance, ASTM F3269-21. (2021). https://doi.org/10.1520/F3269-21
2. Brat, G., Pai, G.: Runtime assurance of aeronautical products: Preliminary recommendations. Technical Memorandum (2023), https://ntrs.nasa.gov/citations/20220015734
3. Goodloe, A.: Challenges in high-assurance runtime verification. In: International Symposium on Leveraging Applications of Formal Methods. ISoLA. Springer (2016). https://doi.org/10.1007/978-3-319-47166-2_31
4. Havelund, K.: Using runtime analysis to guide model checking of java programs. In: International Symposium on Model Checking Software. SPIN. Springer (2000). https://doi.org/10.1007/10722468_15
5. Jeannin, J., Ghorbal, K., Kouskoulas, Y., Schmidt, A.C., Gardner, R.W., Mitsch, S., Platzer, A.: A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. International Journal on Software Tools for Technology Transfer **19**(6) (2017). https://doi.org/10.1007/978-3-662-46681-0_2
6. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: Euromicro Conference on Real-Time Systems. Euromicro RTS. IEEE (1999). https://doi.org/10.1109/EMRTS.1999.777457
7. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: International Conference on Automated Deduction. CADE. Springer (1992). https://doi.org/10.1007/3-540-55602-8_217
8. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41**(2) (2008). https://doi.org/10.1007/s10817-008-9103-8
9. Seto, D., Krogh, B., Sha, L., Chutinan, A.: The simplex architecture for safe online control system upgrades. In: Proceedings of the 1998 American Control Conference. ACC. vol. 6, pp. 3504–3508 (1998). https://doi.org/10.1109/ACC.1998.703255
10. Slagel, J.T., Moscato, M.M., White, L., Muñoz, C., Balachandran, S., Dutle, A.: Embedding differential dynamic logic in PVS. In: International Conference on Logical and Semantic Frameworks, with Applications. LSFA (2023), https://ntrs.nasa.gov/citations/20220019093
11. White, L., Titolo, L., Slagel, J.T., Muñoz, C.: A temporal differential dynamic logic formal embedding. In: ACM SIGPLAN International Conference on Certified Programs and Proofs. CPP (2024). https://doi.org/10.1145/3636501.3636943