A Study on Run Time Assurance for Complex Cyber Physical Systems

Matthew Clark, Xenofon Koutsoukos, Ratnesh Kumar, Insup Lee,
George Pappas, Lee Pike, Joseph Porter, Oleg Sokolsky

April 18, 2013

Contents

1 Introduction

Matthew Clark

Cyber-physical systems are becoming more and more complex, thereby increasing the cost and time to ensure safety. They range from highly complex but finite state to infinite state, indeterminate systems. Current methods of performing software and system verification and validation requires exhaustive offline testing of every possible state space scenario; an impossible task for adaptive, non-deterministic, and near infinite state algorithms. This certification incapacity is creating a growing gap between state of the art software system capabilities and capabilities of systems which can be certified. Currently, we attempt to prove systems are correct via verification of every possible state PRIOR to fielding the system. However, if, through the use of run time architecture, we can provably bound systems behavior, then it may be possible to reduce the reliance on comprehensive off-line verification, shifting the analysis/test burden to the more provable run time assurance mechanism.

Consider autonomy as "the ability to reason and make decisions to reach given goals based on a systems current knowledge and its perception of the variable environment in which it evolves[1]." Autonomous, safety critical software that relies on the perception of its environment to make decisions quickly becomes a large near infinite state problem. To that end, Run Time Assurance (RTA) is the ability to ensure the safe operation of a system that contains functional components, which may not be sufficiently reliable, or sufficiently verified, according to current development or certification standards. There may be multiple reasons for having such components in a system: under the normal conditions, they can provide improved performance or operational efficiency for the system, or enhance the user experience. The core idea that enables the use of such components in a system is the presence of a safe fallback mechanism that:

1) Reliably detects potential problems

2) Invokes a recovery/switching mechanism that can ensure safe operation of the system, possibly with reduced capabilities and performance.

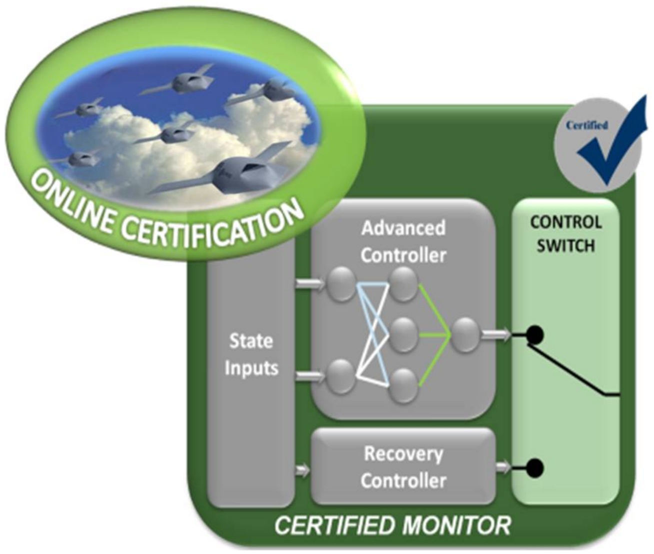


Figure 1: Run Time Assurance of Unverified Flight Critical Software

Within the flight domain, the following certification challenges were identified as only solvable at run time: unanticipated vehicle interactions, unanticipated external interactions, mission/battle management decisions with flight-critical consequences, untested system modes, and autonomous decision making control [1]. The desire is that unmanned aerial systems (UAS's) should be able to use the same infrastructure as manned systems, with minimized uniqueness. They also must be made to be responsive to dynamic missions, adapting in real time to changes in environment, mission, etc. This creates an unsolvable offline certification challenge but an opportunity for run time certification techniques.

Similarly, in the automotive sector, the Google autonomous car has successfully achieved over 100,000 miles of unattended driving in the streets of California [2]. Google has heralded the innovation as "safer than human driving while intoxicated." But up to this point, as required by California state law, a human driver must be present in the vehicle who has the ability to monitor and intervene the autonomous system in case of a failure. For the autonomous vehicle scenario to become reality, the human monitor must be replaced with a certified bounding algorithm that is capable of providing absolute guarantees on the vehicle's safety in the highly dynamic environment such as urban streets.

Within the power distribution industry, innovations in smart-grid technology consider decentralizing power distribution by creating stand-alone power units called "micro-grids." Users would have more control with these units and would have the option to add alternate power sources such as windmills or solar panels. Advantages of micro-grids include versatility, the ability to better align protection with the usage of alternative energy sources, and the relatively small vulnerability to security threats as compared to the current centralized power distribution system. However, the de-centralized "micro-grid" is an inherently unstable control system that needs to be closely monitored to prevent supply over-voltage or sagging [3] . To enable the combined use of the micro-grid, highly adaptive autonomous systems would be needed to carefully manage energy production and consumption and would require a boundary mechanism to assure safety of the system.

Finally, the NITRD report on 21st century medical devices cited the need for adaptive patient-specific algorithms. Current devices are designed to cover a large group of patients with similar medical conditions. Medical devices that adapt to a patient's specific medical condition would improve the quality of patient care. These medical devices may be more responsive to specific patient needs but must not broach a boundary that will cause harm [4] .

As early as 2002, Run Time Assurance was identified as a key technology to increase the reliance on online certification. Barron Associates and Lockheed Martin developed the original concept of "Run Time Assurance." They described RTA as a "mechanism for the implementation of advanced, complex flight control functions". The approach was similar to the Simplex architecture developed by the Software Engineering Institute at Carnegie Mellon University Both the RTA and Simplex concepts focused on verifying that an advanced controller performs within a pre-described boundary and predicts violations of the boundary in enough time to safely switch to a reversionary controller. In addition to work by Barron, Lockheed, and CMU, many government agencies, industrial partners, and academia have explored technologies applicable to run time verification and assurance.

However, a common, implementable framework has yet to be developed for the domain of safe and secure autonomous vehicles. The question arose, what will it take to create a run time assurance framework for the cyber physical systems in the autonomous vehicle space? To explore this question the following study investigates the key technologies available and needed to increase the reliance on run time assurance. To answer this question, four questions were provided to prominent researchers in the domains of Controls and Computer Science.

1) What algorithms can be used to guarantee safe bounds?

2) How do we create a run time version of the algorithm that enables safe switching?

3) How do we ensure timing constraints and worst case execution time is preserved?

4) How can model based design / simulation enable quicker realization of an end product?

The first question, addressed in section 2, explores algorithms and methods suitable for creating a run time assurance argument. For an autonomous system, certain assumptions about the known environment must be made given a set of known input and output states. Utilizing these assumptions to create a boundary for non-deterministic, adaptive systems, RTA aims to achieve advanced performance with the assurance of safety constraints and failsafe operability. The bounding algorithm entails mathematical techniques within the domains of both Control Science and Computer Science to model the environment and synthesize a guaranteed boundary. The constraints can then be verified offline using known Formal Methods approaches. The scope of the bounding algorithm must be large enough to gain the added benefit of allowing the adaptive controller to safely adapt. Secondly, the bounding must incorporate all possible failure modes of what is known. A predominant focus of the bounding algorithm research is on how to synthesize a provable boundary and then how to derive a complete set of properties to ensure safety of that boundary.

The second question, addressed in section 3, investigates the ability to create a run time version of the algorithms and methods described in section 2. Creating a mathematical boundary that accounts for all possible environmental scenarios becomes a highly computationally intensive problem. Such problems are difficult to calculate offline let alone provide assurance dynamically. Once the safety properties and switching conditions are identified, one needs to develop a monitor that will calculate the switching conditions and effect the switch. It is desirable to have a general mechanism in place, which can be instantiated for many different systems that rely on the same class of properties. The mechanism can then be verified once and reused. Development of this mechanism includes the design of a runtime monitoring and switching algorithm for the chosen property language, means of initiating a switch as well as means of determining the completion of the switch. Therefore, the second domain of expertise needed to formulate the RTA framework is the ability to perform the computations at run time. As new approximation methods are being investigated to reduce the required time needed to calculate the safe boundary, methods of run time verification must be employed to ensure the right data is being monitored, at the right time, with the appropriate mechanisms to switch to a safety controller if needed. These approximation methods do not account for the overhead associated with the instrumentation of a program needed to retrieve the right set of conditions and data at the right time.

The third question, addressed in section 4, seeks to ensure timing constraints and worst case execution time is preserved. As run time methods and monitoring software is added, impacts to existing hardware and software interaction will need to be considered. For example, any run time approach for flight critical systems will need to address interactions between triplex redundant control architectures. Technologies need to be considered from a hardware timing, synchronization, and parallel monitoring approach to ensure timing is considered within and external to the system. Multiple processors, cores, or interacting systems of systems rely on consistent timing constraints being followed. In this task, specifics of the system platform are used to implement the monitoring subsystem and deploy it on the platform. For example, if a partitioned architecture is used, the monitoring system may be deployed in a separate partition, which would require that inter-partition communication mechanisms have to be used for forwarding observations to the monitoring subsystem. The effect of this additional communication overhead on the system bus needs to be taken into account.

Finally, section 5 investigates how model based design / simulation enables quicker realization of RTA. Many formal verification and validation techniques emphasize correctness by construction and design for verification. These tag lines speak to the need to ensure the modeling and simulation environment is compatible with the current V&V techniques and formal methods, allowing an increase in validity in methods earlier in the design process. A modeling and simulation environment must be able to connect different abstractions of not only the run time implementation but the environment of which it is protecting. Run Time Assurance must consider such environments in order to accelerate framework production, simulation, verification, and validation.

An exploration of the currently available technology necessary to design and implement a Run Time Assurance methodology that reduces the reliance on test is, by and large, the scope of the project. The remainder of this paper documents the results of further investigation into these four questions and the challenges that arise from each.

2 State of the Art in Run Time Assurance Boundary Methods

Dr. George Pappas and Dr. Ratnesh Kumar

Hybrid systems combine both digital and analog components, in a way that is useful for the analysis and design of distributed, embedded control systems. Hybrid systems have been used as mathematical models for many important applications, such as automated highway systems [5, 6, 7], air traffic management systems [8, 9, 10], embedded automotive controllers [11, 12], manufacturing systems [13], chemical processes [14], robotics [15, 16], real-time communication networks, and real-time circuits [17]. Over the past fifteen years, their wide applicability has inspired a great deal of research from both control theory and theoretical computer science.

Many of the above motivating applications are *safety critical*, and require guarantees of safe operation. Consequently, much research focuses on formal *analysis* and *design* of hybrid systems. Formal analysis of hybrid systems is concerned with verifying whether a hybrid system satisfies a desired specification, like avoiding an unsafe region of the state space. The process of formal design consists of synthesizing controllers for hybrid systems in order to meet a given specification.

2.1 Example: Aircraft conflict resolution

As a safety critical example we consider the verification of a conflict resolution maneuver for air traffic control similar to the one described in [10]. Consider the following conflict resolution maneuver for two aircraft:

1. Cruise until aircraft are α_1 miles apart;
2. Change heading by $\Delta\phi$; fly until lateral displacement of d miles achieved;
3. Change to original heading; fly until aircraft are α_2 miles apart;
4. Change heading by $-\Delta\phi$; fly until lateral displacement of $-d$ miles achieved;
5. Change to original heading.

The hybrid automaton modeling this maneuver has four discrete states $\{$*CRUISE, LEFT, STRAIGHT, RIGHT*$\}$ and is depicted in Figure 2. The continuous dynamics in each discrete state is the relative dynamics of the aircraft given a fixed velocity and heading, (v_i is the velocity and ϕ_i is the heading of aircraft i). The aircraft are considered to be at a safe distance if they are at least 5 miles apart. In the relative coordinate frame, the *unsafe* set is given by $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 25\}$. Aircraft 1 is assumed to fly at a fixed velocity v_1 and heading ϕ_1, while aircraft 2 can switch "modes" and rotate left or right a fixed angle of $\pm\Delta\phi$.

Major efforts have focused on developing algorithms that verify the safety of such hybrid system models (off-line analysis). For example, safety analysis of the hybrid system computes the region of the state space that will lead to a conflict. The result of such a computation of the minimal unsafe sets is shown in Figure 3. The set $\texttt{unsafeCruise} \setminus \texttt{unsafeLeft}$ contains the set of states which are made safe by the aircraft turning left, and the set $\texttt{unsafeCruise} \setminus \texttt{unsafeRight}$ contains the set of states which are made safe by the aircraft turning right. The set $\texttt{unsafeCruise} \setminus (\texttt{unsafeLeft} \cup \texttt{unsafeRight})$ contains the states which are made safe by turning either left or right, and the set $\texttt{unsafeCruise} \cap \texttt{unsafeLeft} \cap \texttt{unsafeRight}$ shown in Figure 3(d) is the set of states which is unsafe regardless of the action the aircraft takes.

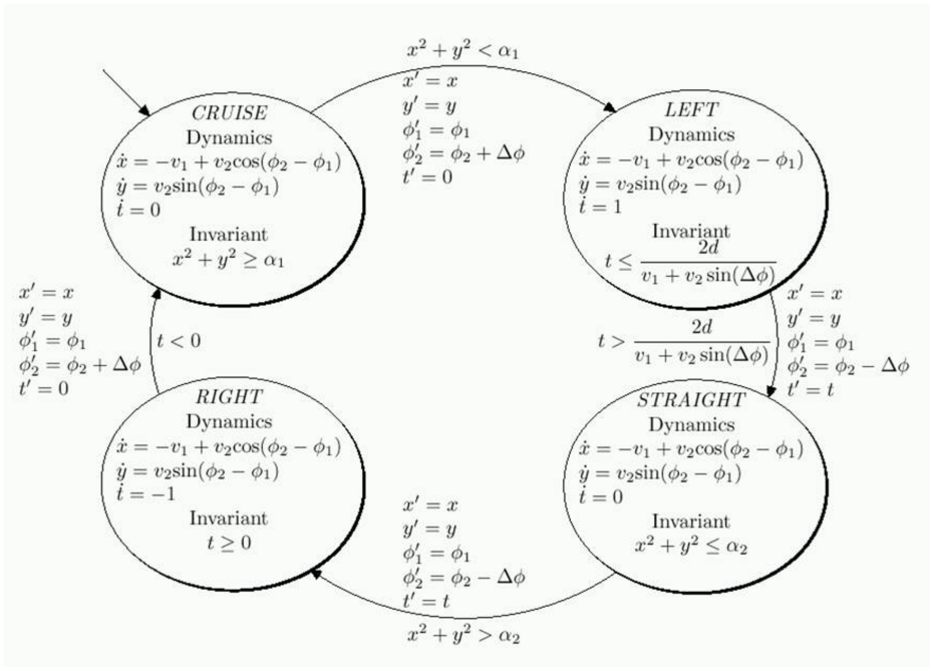
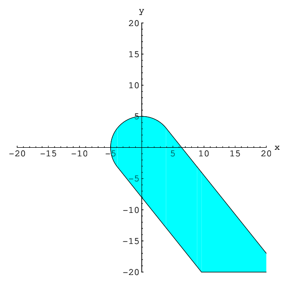
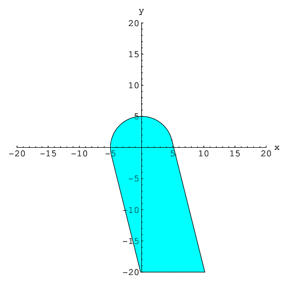

$x^2 + y^2 < \alpha_1$
$x' = x$
$y' = y$
$\phi_1' = \phi_1$
$\phi_2' = \phi_2 + \Delta\phi$
$t' = 0$
CRUISE
Dynamics
$\dot{x} = -v_1 + v_2\cos(\phi_2 - \phi_1)$
$\dot{y} = v_2\sin(\phi_2 - \phi_1)$
$\dot{t} = 0$
Invariant
$x^2 + y^2 \geq \alpha_1$
LEFT
Dynamics
$\dot{x} = -v_1 + v_2\cos(\phi_2 - \phi_1)$
$\dot{y} = v_2\sin(\phi_2 - \phi_1)$
$\dot{t} = 1$
Invariant
$t \leq \frac{2d}{v_1 + v_2\sin(\Delta\phi)}$
$x' = x$
$y' = y$
$\phi_1' = \phi_1$
$\phi_2' = \phi_2 + \Delta\phi$
$t' = 0$
$t < 0$
$t > \frac{2d}{v_1 + v_2\sin(\Delta\phi)}$
$x' = x$
$y' = y$
$\phi_1' = \phi_1$
$\phi_2' = \phi_2 - \Delta\phi$
$t' = t$
RIGHT
Dynamics
$\dot{x} = -v_1 + v_2\cos(\phi_2 - \phi_1)$
$\dot{y} = v_2\sin(\phi_2 - \phi_1)$
$\dot{t} = -1$
Invariant
$t \geq 0$
STRAIGHT
Dynamics
$\dot{x} = -v_1 + v_2\cos(\phi_2 - \phi_1)$
$\dot{y} = v_2\sin(\phi_2 - \phi_1)$
$\dot{t} = 0$
Invariant
$x^2 + y^2 \leq \alpha_2$
$x' = x$
$y' = y$
$\phi_1' = \phi_1$
$\phi_2' = \phi_2 - \Delta\phi$
$t' = t$
$x^2 + y^2 > \alpha_2$

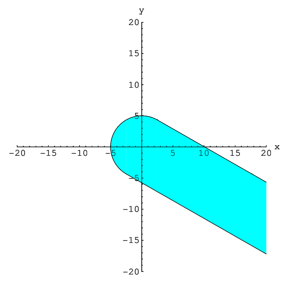
Figure 2: Hybrid system model of aircraft conflict resolution maneuver.



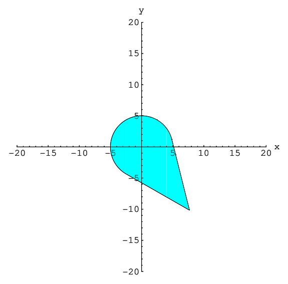
(a) `unsafeCruise`



(b) `unsafeLeft`



(c) `unsafeRight`



(d) `unsafeCruise` $\wedge$ `unsafeLeft` $\wedge$ `unsafeRight`

Figure 3: Showing minimal unsafe sets for each discrete state of maneuver automaton.

2.2 Computability boundary

In developing computational approaches for hybrid system verification, the first major obstacle has to do with computability. As hybrid control systems increase to have an infinite number of states, showing that a class of verification problems is computable was challenging. A natural way to show that a class of safety verification problems is computable, is the process of *abstraction*. Given a hybrid control system and some desired property, one extracts a finite, discrete system while preserving all properties of interest. This is achieved by constructing suitable, *finite* and *computable* partitions of the state space of the hybrid control system. By obtaining discrete abstractions which are finite, and preserve properties of interest, analysis can be equivalently performed on the finite system, which requires only a finite number of steps, guaranteeing termination. Checking the desired property on the abstracted system should be *equivalent to* checking the property on the original system.

Properties about the behavior of a system over time are naturally expressible in temporal logics, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL*). Preserving LTL properties leads to special partitions of the state space given by *language equivalence relations*, whereas CTL* properties are abstracted by *bisimulations*.

Unfortunately, there are severe obstacles due to undecidability. For example, in [18] it was shown that checking *reachability* (whether a certain region of the state space can be reached) is undecidable for a very simple class of hybrid systems, where the continuous dynamics involves only variables that proceed at two constant slopes. These results implied that more general classes of hybrid systems cannot have finite bisimulation or language equivalence quotients.

Therefore, exact discrete abstractions of hybrid systems is limited by this result. Given this limit, hybrid systems that can be abstracted fall into two classes. In the first class, the continuous behavior of the hybrid system must be restricted, as in the case of timed automata [19], multirate automata [20, 21], and rectangular automata [18, 22]. In the second class, the discrete behavior of the hybrid system must be restricted, as in the case of order-minimal hybrid systems [23, 24, 25].

2.3 Discrete abstractions

While the decidability frontier for hybrid system verification has been well understood for some time, in most practical cases when no equivalent abstraction can be found, one may be content with a *sufficient* abstraction or an approximation, where checking the desired property on the abstracted system is sufficient for checking the property on the original system.

The basic idea is to replace the actual system by a simpler, abstract system, in which model checking is easier to perform. The verification results about the abstract system, of course, can only be related back to verification results about the original concrete system under certain conditions on how the abstract and concrete system are related and whether the particular property in question survives this abstraction process.

The options for directly constructing discrete abstractions by finite quotients and for which subclasses they work have been examined by Henzinger [26, 27]. Because of the limited scope of discrete abstractions, more general predicate abstractions [28, 29, 30, 31, 32, 33] and abstraction-refinement techniques like Counterexample-Guided Abstraction-Refinement (CEGAR) have been developed subsequently [34, 29, 35, 36]. Those directions have again worked successfully in discrete and, to some extent, real-time systems.

The use of discrete abstractions for continuous dynamics has become standard in hybrid systems verification (see e.g. [37, 38]). The main advantage of this approach is that it offers the possibility to leverage controller synthesis techniques developed in the areas of supervisory control of discrete-event systems. We briefly describe the approach presented in [39] for computing discrete abstractions for a class of switched control systems:

$$\Sigma: \; \dot{\mathbf{x}}(t) = f_{\mathbf{p}(t)}(\mathbf{x}(t)), \; \mathbf{x}(t) \in \mathbb{R}^n, \; \mathbf{p}(t) \in P$$

where P is a finite set of modes. Given a parameter $\tau > 0$, we define a transition system $T_\tau(\Sigma)$ that describes trajectories of duration τ of Σ. This can be seen as a time sampling process. This is natural when the switching in Σ is determined by a time-triggered controller with period τ.

The computation of a discrete abstraction of $T_\tau(\Sigma)$ can be done by the following simple approach. We start by approximating the set of states $X_1 = \mathbb{R}^n$ by a lattice:

$$[\mathbb{R}^n]_\eta = \left\{ x \in \mathbb{R}^n \;\middle|\; x_i = k_i \frac{2\eta}{\sqrt{n}}, \; k_i \in \mathbb{Z}, \; i = 1, ..., n \right\},$$

where x_i is the i-th coordinate of x and $\eta > 0$ is a state space discretization parameter.

We can then define the abstraction of $T_\tau(\Sigma)$ as the transition system $T_{\tau,\eta}(\Sigma) = (X_2, U, \mathcal{S}_2, X_2^0, Y, \mathcal{O}_2)$, where the set of states is $X_2 = [\mathbb{R}^n]_\eta$; the set of labels remains the same $U = P$; the transition relation is essentially obtained by rounding the transition relation of $T_\tau(\Sigma)$ on the lattice $[\mathbb{R}^n]_\eta$ (see Figure 4):

$$x_2' = \mathcal{S}_2(x_2, p) \iff x_2' = \arg \min_{x' \in [\mathbb{R}^n]_\eta} \|x' - \mathcal{S}_1(x_2, p)\|;$$

the set of outputs remains the same $Y = \mathbb{R}^n$; the observation map $\mathcal{O}_2$ is given by $\mathcal{O}_2(q) = q$, the set of initial states is $X_2^0 = [\mathbb{R}^n]_\eta$. Note that the transition system $T_{\tau,\eta}(\Sigma)$ is discrete since its sets of states and actions are respectively countable and finite.

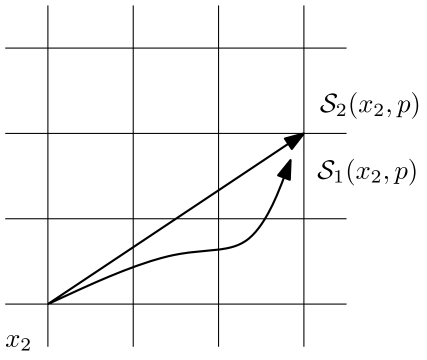


Figure 4: Principle for the computation of a discrete abstraction of a switched system.

The approximate bisimilarity of $T_\tau(\Sigma)$ and $T_{\tau,\eta}(\Sigma)$ is related to the notion of *incremental stability* [40]. Intuitively, incremental global uniform asymptotic stability (δ-GUAS) of a switched system means that all the trajectories associated with the same switching signal converge to the same reference trajectory independently of their initial condition. Incremental stability of a switched system can be characterized using Lyapunov functions.

Definition 1 *A smooth function $\mathcal{V} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+$ is a common δ-GUAS Lyapunov function for Σ if there exist $\mathcal{K}_\infty$ functions*[1] *$\underline{\alpha}$, $\overline{\alpha}$ and $\kappa > 0$ such that for all $x, y \in \mathbb{R}^n$, for all $p \in P$:*

$$\underline{\alpha}(\|x - y\|) \leq \mathcal{V}(x, y) \leq \overline{\alpha}(\|x - y\|);$$
$$\frac{\partial \mathcal{V}}{\partial x}(x, y) \cdot f_p(x) + \frac{\partial \mathcal{V}}{\partial y}(x, y) \cdot f_p(y) \leq -\kappa \mathcal{V}(x, y).$$

[1]A continuous function $\gamma : \mathbb{R}^+ \to \mathbb{R}^+$ is said to belong to class $\mathcal{K}_\infty$ if it is strictly increasing, $\gamma(0) = 0$ and $\gamma(r) \to \infty$ when $r \to \infty$.

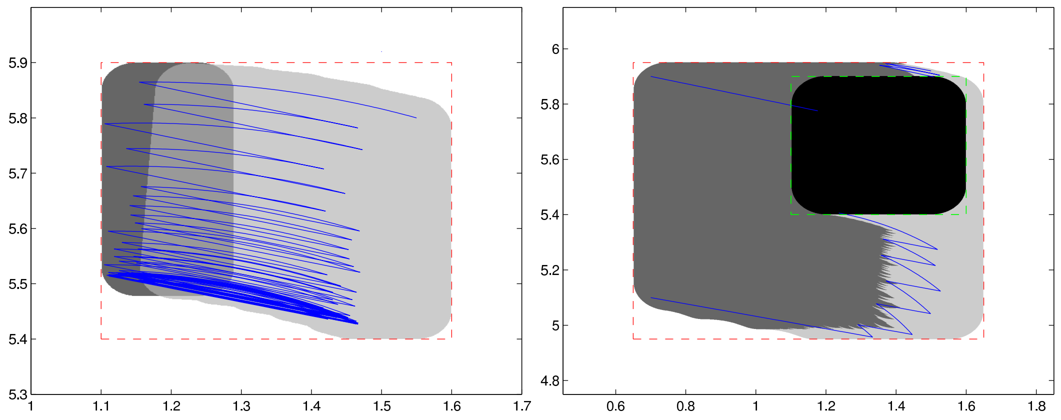


Figure 5: Example of switching safety (left) and reachability (right) controllers synthesized using approximately bisimilar discrete abstractions of a model of a DC-DC converter [41, 42].

The existence of a common δ-GUAS Lyapunov function ensures that the switched system Σ is incrementally stable. We need to make the supplementary assumption on the δ-GUAS Lyapunov function that there exists a $\mathcal{K}_\infty$ function γ such that

$$\forall x, y, z \in \mathbb{R}^n,\ |\mathcal{V}(x,y) - \mathcal{V}(x,z)| \leq \gamma(\|y - z\|). \tag{1}$$

Theorem 1 *Consider a switched system Σ, time and state space sampling parameters $\tau, \eta > 0$ and a desired precision $\varepsilon > 0$. If there exists a common δ-GUAS Lyapunov function $\mathcal{V}$ for Σ such that equation (1) holds and*

$$\eta \leq \min\left\{\gamma^{-1}\left((1 - e^{-\kappa\tau})\underline{\alpha}(\varepsilon)\right), \overline{\alpha}^{-1}\left(\underline{\alpha}(\varepsilon)\right)\right\} \tag{2}$$

then

$$\mathcal{R}_\varepsilon = \{(x_1, x_2) \in X_1 \times X_2 |\ \mathcal{V}(x_1, x_2) \leq \underline{\alpha}(\varepsilon)\}$$

is an ε-approximate bisimulation relation between $T_\tau(\Sigma)$ and $T_{\tau,\eta}(\Sigma)$. Moreover, $T_\tau(\Sigma) \sim_\varepsilon T_{\tau,\eta}(\Sigma)$.

Let us remark that the δ-GUAS Lyapunov function $\mathcal{V}$ essentially plays the role of an approximate bisimulation function here. Particularly, it should be noted that given a time sampling parameter $\tau > 0$ and a desired precision $\varepsilon > 0$, it is always possible to choose $\eta > 0$ such that equation (2) holds. This essentially means that approximately bisimilar discrete abstractions of arbitrary precision can be computed for $T_\tau(\Sigma)$.

For simplicity, we only presented results for switched systems that have a common δ-GUAS Lyapunov function. However, let us remark that it is possible to extend these results for systems with multiple Lyapunov functions by imposing a dwell time (see [39] for more details). The approach described above has been used in [41, 42] for synthesizing safety and reachability switching controllers for a model of a DC-DC converter (see Figure 5). Similar approaches can be used to compute approximately bisimilar abstractions for incrementally stable nonlinear systems [43] and time-delay nonlinear systems [44]. An approach relaxing the incremental stability assumptions for non-linear systems can be found [45]; this approach is made currently available in the tool PESSOA [46]. The development of discrete abstractions defined on non-uniform grids has been treated in [47, 48]. Finally, an approach for computing discrete abstractions of linear systems without griding the state-space can be found in [49].

In [30] Kumar et al. presented an approach for discretizing a hybrid system to a purely discrete transition system that preserves its behaviors at all switching instances and further preserves the structure of the underlying discrete transitions graph (transition structure of the hybrid automaton). This guarantees that the abstracted discrete transition system preserves the associated reachability/safety properties and so those can be verified using model-checking and counter-example guided abstraction refinement approaches as demonstrated in the work of Jiang [35]. This safety verification approach was applied to synchronization of distributed local clocks of the nodes on a CAN bus by Jiang et al. [36]. The class of discrete transition systems obtained by abstracting hybrid systems can themselves possess finite bisimilar abstractions, and such classes have been reported by Zhou et al. in [31, 32]. The method of abstracting hybrid systems into safety preserving discrete abstractions has further been employed by Zhou et al. [50] and Li et al. [33] for abstracting Simulink/Stateflow diagrams into discrete transition systems.

2.4 Reachability computations

The basic idea behind explicit-state reachability analysis is to perform a set-valued simulation of the system. Simulating the execution of a system from a single starting state up to a prespecified number of transitions is easy. It is especially efficient when the system is deterministic, because there are no choices in the simulation then.

Reachability analysis generalizes the simulation of a single initial point to simulating how a set of initial states evolves under a prespecified number of transitions of the system. Like simulations, reachability analysis directly follows the transition semantics of hybrid automata but considers appropriate sets of states instead of single states.

Reachability analysis cannot work with arbitrary sets. But suppose the initial set of states is represented by a set P of a specific family of shapes $\mathcal{S}$ (e.g., all polygons, or all rectangles, and so on). Suppose there is a way to compute the image of a set of states of shape $\mathcal{S}$ under one transition of the system and that the resulting set of states can be overapproximated again by a set of shape $\mathcal{S}$. Then, for any prespecified number of transitions, say 10, the image of an initial set P of shape $\mathcal{S}$ under the system dynamics can be approximated by computing the one-step transition of P and then the one-step transition of the resulting set of shape $\mathcal{S}$ and so on until 10 transitions have been made. Since the resulting set will again have an overapproximation of shape $\mathcal{S}$, the resulting procedure can be used for (approximate) bounded model checking.

The effectiveness and efficiency of this reachability analysis hinges on the computational properties of the shapes $\mathcal{S}$ and whether the dynamics of a particular class of systems can be approximated well by sets of shape $\mathcal{S}$. The standard method to compute the reachable states is to iterate the following *one-step successor* operators for discrete and continuous transitions respectively. We now present computing such successor for two different classes of continuous dynamics.

2.4.1 Zonotope methods for linear control systems

Zonotopes are a compact representation for a special form of polygons that have been used successfully for reachability analysis [51, 52, 53, 54], because their special form simplifies several aspects of reachability computation. A *zonotope* $P \subseteq \mathbb{R}^n$ is defined by a center $c \in \mathbb{R}^n$ and a finite number of generators $v_1, \ldots, v_k \in \mathbb{R}^n$ that span the polytope as bounded linear combinations from the center:

$$P = \left\{ c + \sum_{i=1}^{k} \alpha_i v_i \mid \alpha_i \in [-1, 1] \right\}$$

A common denotation for this zonotope is $(c, \langle v_1, \ldots, v_k \rangle)$. Zonotopes are central-symmetric convex polytopes with computationally attractive features. Linear transformations, which are one of the basic operations in reachability of linear systems, can be computed efficiently for zonotopes. For a matrix $A \in \mathbb{R}^{n \times n}$, the image of the linear transformation $x \mapsto Ax$ of zonotope $P = (c, \langle v_1, \ldots, v_k \rangle)$ can simply be computed component-wise:

$$AP = (Ac, \langle Av_1, \ldots, Av_k \rangle)$$

The Minkowski sum of zonotopes $P = (c, \langle v_1, \ldots, v_k \rangle)$ and $Q = (d, \langle w_1, \ldots, w_m \rangle)$ can also be computed efficiently by a single vector addition and a single list concatenation:

$$P \oplus Q = (c + d, \langle v_1, \ldots, v_k, w_1, \ldots, w_m \rangle).$$

It is also computationally efficient to find box-hull overapproximations of zonotopes.

The intersection of zonotopes with guards of hybrid automata and the regions resulting from the resets are unfortunately not generally zonotopes, which still poses challenges for using zonotope reachability for hybrid systems. The technique is extremely scalable for continuous systems with an LTI (linear time-invariant) dynamics [52, 53, 54]. Just like when working with general polytopes, direct uses of zonotope-based reachability computation suffer from the wrapping effect [53]. The *wrapping effect* is the effect that approximation errors accumulate during the computation, making results at higher time-steps increasingly useless. Each of these image computations may cause approximation errors. Other versions of zonotope-based reachability computation avoid wrapping effects [52], even for LTI with input, by basing all subsequent image computations on the original zonotope. This way, approximation errors may still accumulate at hybrid switching surfaces, but no longer at each intermediate time-step of a continuous transition. Extensions of zonotope-based reachability analysis to nonlinear systems and hybrid systems can be found at [55, 56]. Another pleasant property of zonotope computations is that they turn out to be numerically more stable than general polytope operations. Zonotope-based reachability computation has recently been extended to reachability computations based on support functions [57], which are essentially a generalization of zonotopes to (subclasses of) generators in infinitely many directions.

2.4.2 Hamilton-Jacobi methods for nonlinear control systems

Level set methods are a class of algorithms designed for approximating the solution of the Hamilton-Jacobi partial differential equation (PDE) [58], which arises in many fields including optimal control, differential games, and dynamic implicit surfaces. In particular, dynamic implicit surfaces can be used to compute backward reachable sets and tubes for nonlinear, nondeterministic, continuous dynamic systems with control and/or disturbance inputs; in other words, inputs and parameters to the model can be treated in a worst case and/or best case fashion. The strengths and weaknesses of the Hamilton-Jacobi PDE formulation of reachability are very similar to those of viability theory: it can treat very general dynamics with adversarial inputs and can represent very general sets, but the known computational algorithms require resources that grow exponentially with the state space dimension; for example, in ToolboxLS[2] the level set algorithms run on a Cartesian grid of the state space. The ToolboxLS algorithms also do not guarantee the sign of computational errors, but they deliver higher accuracy for a given resolution than that available from typical sound alternatives.

Because ToolboxLS [59] is designed for dynamic implicit surfaces rather than specifically for reachability, it does not include a specialized verification interface; however, it has a 140 page user manual documenting the software and over twenty complete examples including three reachable

[2] `http://www.cs.ubc.ca/~mitchell/ToolboxLS/`

set computations. It has been used primarily for reachability of systems with two to four continuous dimensions, including collision avoidance, quadrotor flips, aerial refueling, automated landing, and glidepath recapture.

The above reachability computation approach was also used by Jin et al. in [60] to demonstrate how backward reachability methods can be used to compute the stability regions and how those can then be used for switching control designs for achieving power system stability under discrete line or bus faults.

2.5 Barrier certificate methods

The basic idea behind barrier certificates is to find a barrier separating reachable states and bad states that we can easily show to be impenetrable by the continuous system dynamics. Barrier certificates were proposed for safety verification in [61] and later refined in [62]. Barrier methods are different from explicit reachability computational approaches as they do not require computation of reachable sets, but instead relies on a deductive inference.

For a continuous control system, a barrier certicate is a function of state satisfying a set of inequalities on both the function itself and its Lie derivative along the flow of the system. In the state space, the zero level set of a barrier certicate separates an unsafe region from all system trajectories starting from a set of possible initial states. Therefore, the existence of such a function provides an exact certicate/proof of system safety. Similar to the Lyapunov approach for proving stability, the main idea here is to study properties of the system without the need to compute the reachable set explicitly. Although an overapproximation of the reachable set may also be used as a proof for safety, a barrier certicate may be easier to compute when the system is nonlinear and uncertain. Moreover, a barrier certicate can be easily used to verify safety in innite time horizon. With this methodology, it is possible to treat a large class of hybrid systems, including those with nonlinear continuous dynamics, uncertainty, and constraints. Consider a control system described by ordinary differential equations

$$\dot{x}(t) = f(x(t), d(t)), \tag{3}$$

where $x(t) \in \mathbb{R}^n$ is the state of the system, $d(t) \in \mathcal{D} \subseteq \mathbb{R}^m$ is a collection of uncertain disturbance inputs, and $f \in C(\mathbb{R}^{n+m}, \mathbb{R}^n)$. We will be mostly dealing with a bounded disturbance set $\mathcal{D}$. In the safety verification problem, we will be interested only in segments of system trajectories that are contained in a given set $\mathcal{X} \subseteq \mathbb{R}^n$. Now suppose also that a set of possible initial states $\mathcal{X}_0 \subseteq \mathcal{X}$ and a set of unsafe states $\mathcal{X}_u \subseteq \mathcal{X}$ are given. Our objective is to prove that the system is safe in the following sense.

Definition 2 (Safety) *Given a system (3) and the sets $\mathcal{X}$, $\mathcal{D}$, $\mathcal{X}_0$ and $\mathcal{X}_u$, we say that the system safety property holds if there does not exist a time instant $T \geq 0$, a bounded and piecewise continuous disturbance input $d : [0, T] \to \mathcal{D}$, and a corresponding trajectory $x : [0, T] \to \mathbb{R}^n$ such that $x(0) \in \mathcal{X}_0$, $x(T) \in \mathcal{X}_u$, and $x(t) \in \mathcal{X}$ $\forall t \in [0, T]$.*

The safety of the system (3) can be shown by the existence of a barrier certificate [63]. A barrier certificate is a function of state satisfying some Lyapunov-like conditions on both the function itself and its time derivative along the flow of the system, stated in Proposition 1 below. The main idea is to ask that the value of the function at the initial set $\mathcal{X}_0$ to be non-positive, the time derivative of the function to be non-positive on $\mathcal{X}$, and the value of the function at the unsafe set $\mathcal{X}_u$ to be strictly positive. If a function satisfying such a property can found, then we can conclude that no trajectory of the system starting from $\mathcal{X}_0$ can reach $\mathcal{X}_u$.

Proposition 1 ([63]) *Let the system (3) and the sets $\mathcal{X} \subseteq \mathbb{R}^n$, $\mathcal{D} \subseteq \mathbb{R}^m$, $\mathcal{X}_0 \subseteq \mathcal{X}$ and $\mathcal{X}_u \subseteq \mathcal{X}$ be given, with $f \in C(\mathbb{R}^{n+m}, \mathbb{R}^n)$. Suppose there exists a function $B \in C^1(\mathbb{R}^n)$ that satisfies the following conditions:*

$$B(x) \leq 0 \quad \forall x \in \mathcal{X}_0, \tag{4}$$

$$B(x) > 0 \quad \forall x \in \mathcal{X}_u, \tag{5}$$

$$\frac{\partial B}{\partial x}(x) f(x,d) \leq 0 \quad \forall (x,d) \in \mathcal{X} \times \mathcal{D}, \tag{6}$$

then the safety of the system (3) in the sense of Definition 2 is guaranteed.

The above method is analogous to the Lyapunov method for stability analysis. Contrary to stability analysis, however, no notion of equilibrium, stability, or convergence is required in safety verification. For example, the system does not even need to have an equilibrium.

The conditions in Proposition 1 define a convex set of barrier certificates $\{B(x)\}$. This is a very beneficial property, as a barrier certificate inside this set can be searched using convex optimization. For example, when the vector field $f(x,d)$ is polynomial and the sets $\mathcal{X}$, $\mathcal{D}$, $\mathcal{X}_0$, $\mathcal{X}_u$ are semialgebraic, i.e., defined by polynomial inequalities and equalities, a computational framework called *sum of squares optimization* [64] that is based on semidefinite programming can be utilized to search for a polynomial barrier certificate. In particular, the software SOSTOOLS [64] is available for this computation.

The method can also be extended to handle safety verification of hybrid systems, systems with disturbances and to switching diffusion systems [62].

2.6 Verification using robust simulation

Verification consists in analyzing the behavior of a control system Σ against some specification: given a property Φ defined on trajectories (e.g. "a trajectory reaches the set Y_F"), we would like to be able to prove that it is satisfied by all trajectories of the control system. For discrete software/hardware systems, the problem has attracted a lot of attention in computer science, resulting in the success story of Model Checking [65] with the associated set of techniques widely used in the industry. Verification of continuous control and hybrid systems is in general much more challenging since these generally have an uncountable number of trajectories. There has also been a lot of work on the subject using set-based reachability computations, abstraction and or deductive techniques (see [66] for a recent survey).

In the following, we briefly describe an approach based on simulation of individual trajectories of a system together with the use of bisimulation function [67]. Let us consider the following dynamical system:

$$\Sigma : \left\{ \begin{array}{lcl} \dot{\mathbf{x}}(t) & = & f(\mathbf{x}(t)),\ \mathbf{x}(t) \in \mathbb{R}^n,\ \mathbf{x}(0) \in I^0 \\ \mathbf{y}(t) & = & g(\mathbf{x}(t)),\ \mathbf{y}(t) \in \mathbb{R}^k. \end{array} \right.$$

Let Φ be a property defined on the output trajectories $\mathbf{y}$, formulated for instance in some temporal logic; $\mathbf{y} \models \Phi$ means that trajectory $\mathbf{y}$ satisfies property Φ. Let us assume that we are given a measure $\rho(\mathbf{y}, \Phi)$ estimating how robustly property Φ is satisfied by Φ:

$$\big(\forall t \geq 0,\ \|\mathbf{y} - \mathbf{y}'\| \leq \rho(\mathbf{y}, \Phi)\big) \implies \mathbf{y}' \models \Phi.$$

We refer the reader to [67] for a precise definition of such a measure and algorithms for its computation. The last ingredient of the verification approach is a bisimulation function between Σ and itself:

Definition 3 *Let $\mathcal{V} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+$ be a smooth function, $\mathcal{V}$ is an (auto)bisimulation function of Σ if for all $(x_1, x_2) \in \mathbb{R}^n \times \mathbb{R}^n$,*

$$\mathcal{V}(x_1, x_2) \geq \|g(x_2) - g(x_1)\|$$

$$\frac{\partial \mathcal{V}(x_1, x_2)}{\partial x_1} \cdot f(x_1) + \frac{\partial \mathcal{V}(x_1, x_2)}{\partial x_2} \cdot f(x_2) \leq 0$$

Using robustness measures for property satisfaction and a bisimulation function makes it possible to verify that the property holds for an infinite number of trajectories by simulating only one trajectory:

Theorem 2 *Let $x^0 \in I^0$ be an initial condition of Σ and let $\mathbf{y}$ be the associated output trajectory. Then, for all $x'^0 \in I^0$, with associated output trajectories $\mathbf{y}'$*

$$\left(\mathcal{V}(x^0, x'^0) \leq \rho(\mathbf{y}, \Phi)\right) \implies \mathbf{y}' \models \Phi$$

The previous result allows us to verify that the property Φ holds for all the trajectories of Σ by computing only a finite number of them. Let $\{x_1^0, \ldots, x_n^0\} \subseteq I^0$ be a finite subset of initial conditions and $\{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ the associated output trajectories of Σ such that for all $x^0 \in I^0$, there exists x_i^0 such that $\mathcal{V}(x^0, x_i^0) \leq \rho(\mathbf{y}_i, \Phi)$ then all the trajectories of Σ satisfy Φ. An algorithm to construct iteratively the sample of initial states can be found in [67]. In the case we cannot cover the whole set of initial states, the algorithm identifies a subset of initial states for which the property holds.

Similar approaches have been developed for the verification of non-deterministic metric transition systems [68, 69] or hybrid systems [70, 71]. The same kind of ideas can be used for controller synthesis by defining control laws for a finite number of trajectories [72].

2.7 Computational tools

Here is a current list of state of the art tools for hybrid system verification and synthesis.

- **SpaceEx** The SpaceEx tool platform (spaceex.imag.fr) developed at VERIMAG [73, 74] is a tool for computing reachability of hybrid systems with complex, high-dimensional dynamics. It can handle hybrid automata whose continuous and jump dynamics are piecewise affine with nondeterministic inputs. Nondeterministic inputs are particularly useful for modeling the approximation error when nonlinear systems are brought to piecewise affine form. SpaceEx comes with a web-based graphical user interface and a graphical model editor. Its input language facilitates the construction of complex models from automata components that can be combined to networks and parameterized to construct new components.

  The analysis engine of SpaceEx combines explicit set representations (polyhedra), implicit set representations (support functions) and linear programming [75] to achieve a high scalability while maintaining high accuracy. It constructs an overapproximation of the reachable states in the form of template polyhedra. Template polyhedra are polyhedra whose faces are oriented according to a user-provided set of directions (template directions). A cover of the continuous trajectories is obtained by time-discretization with an adaptive time-step algorithm. The algorithm ensures that the approximation error in each template direction remains below a given value. Empirical measurements indicate that the complexity of the image computations is linear in the number of variables, quadratic in the number of template directions, and linear in the number of time-discretization steps.

  The accuracy of the overapproximation can be increased arbitrarily by choosing smaller time steps and adding more template directions. To attain a given approximation error (in the

Hausdorff sense), the number of template directions is worst-case exponential. In case studies, the developers of SpaceEx observe that a linear number of user-specified directions, possibly augmented by a small set of 'critical' directions, often suffices. The prime goal of SpaceEx being scalability, it uses floating-point computations that do not formally guarantee soundness. SpaceEx has been used to verify continuous and hybrid systems with more than 100 continuous variables.

- **S-TALIRO** S-TaLiRo (https://sites.google.com/a/asu.edu/s-taliro/s-taliro) is a Matlab toolbox developed at Arizona State University that searches for trajectories of minimal robustness in Simulink / Stateflow. It can analyze arbitrary Simulink models or user defined functions that model the system. Among the advantages of the toolbox is the seamless integration inside the Matlab environment, which is widely used in the industry for model-based development of control software.

- **PESSOA** Pessoa (https://sites.google.com/a/cyphylab.ee.ucla.edu/pessoa/) is a software toolbox, developed at UCLA's CyPhyLab, for the synthesis of correct-by-design embedded control software. It is based on the recent notion of approximate bisimulation that allows one to replace differential equations, describing a physical system, by an equivalent finite-state machine. Controller design problems can then be solved by using efficient synthesis algorithms operating over the equivalent finite-state machine models. The resulting controllers are also finite-state, are guaranteed to enforce the control specifications on the original physical system, and can be readily transformed into bug-free code for any desired digital platform.

- **TuLiP** (http://sourceforge.net/projects/tulip-control/) developed at Cal Tech is a Python-based software toolbox for the synthesis of embedded control software that is provably correct with respect to an expressive subset of linear temporal logic (LTL) specifications. TuLiP combines routines for (1) finite state abstraction of control systems, (2) digital design synthesis from LTL specifications, and (3) receding horizon control. The underlying digital design synthesis routine treats the environment as adversary; hence, the resulting controller is guaranteed to be correct for any admissible environment profile. TuLiP applies the receding horizon framework, allowing the synthesis problem to be broken into a set of smaller problems, and consequently alleviating the computational complexity of the synthesis procedure, while preserving the correctness guarantee.

- **LTLMOP** (http://ltlmop.github.com/), developed at Cornell University, stands for Linear Temporal Logic MissiOn Planning and is a collection of Python applications for designing, testing, and implementing hybrid controllers generated automatically from task specifications written in Structured English or Temporal Logic.

- **Toolbox for Level Set methods** The toolbox of level set methods developed at UC Berkeley and the University of British Columbia (http://www.cs.ubc.ca/ mitchell/ToolboxLS/index.html) for solving time-dependent Hamilton-Jacobi partial differential equations (PDEs) in the Matlab programming environment. Hamilton-Jacobi and related PDEs arise in computing reach-avoid operators for nonlinear control systems. The algorithms in the toolbox can be used in any number of dimensions, although computational cost makes dimensions four and higher a challenge.

- **HSolver** (http://hsolver.sourceforge.net/) [76] is an open-source software package for the formal verification of safety properties of continuous-time hybrid systems. It allows hybrid systems with non-linear ordinary differential equations and non-linear jumps assuming a global compact domain restriction on all variables. Even though HSolver is based on fast machine-precision floating point arithmetic, it uses sound rounding, and hence the correctness of its results cannot be hampered by round-off errors. HSolver does not only verify (unbounded horizon) reachability properties of hybrid systems, but—in addition—it also computes abstractions of the input system. So, even for input systems that are unsafe, or for which exhaustive formal verification is too difficult, it will compute abstractions that can be used by other tools. For example, the abstractions could be used for guiding search for error trajectories of unsafe systems.

- **KeYmaera** (http://symbolaris.com/info/KeYmaera.html) [77, 78, 79, 80, 81] is a hybrid verification tool for hybrid systems that combines deductive, real algebraic, and computer algebraic prover technologies. It is an automated and interactive theorem prover for a natural specification and verification logic for hybrid systems. With this, the verification principle behind this tool is fundamentally different and quite complementary to existing tools. It supports differential dynamic logic [82, 77, 79, 80], which is a real-valued first-order dynamic logic for hybrid programs [82, 77, 79, 80], a program notation for hybrid systems.

2.8 Run-Time Assurance Framework

Run-time assurance framework provides methodologies (algorithms and tools) to ensure safe operation of a given system while maximizing the use of its certain advanced subsystem that offers complex performance capabilities but that is not fully verified. The advanced subsystem is accompanied by a substitute baseline subsystem that has lower complexity and has been verified to be fail-safe. As a result, it is possible to revert control from the advanced subsystem to the substitute baseline subsystem whenever it is deemed necessary during the run-time operation, thereby assuring safety of the overall system in a run-time fashion while maximizing the use of the advanced subsystem.

A possible architecture for run-time assurance framework is shown in Figure 6 that is inspired from the works of Barron Associates and CMU [83, 84]. Here the system under control (plant),

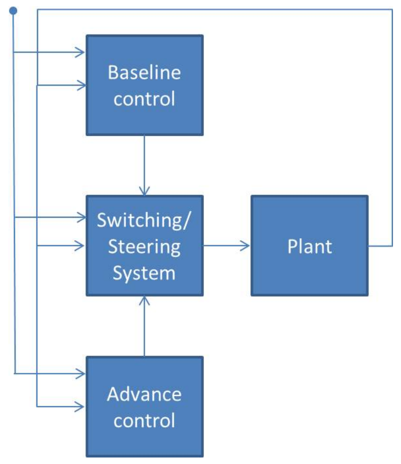


Figure 6: Run-time assurance architecture

a verified fail-safe baseline control and an unverified advanced control are already designed and made available. The objective is to design a switching/steering system so as to maximize the use of the advance control without sacrificing the safety of the overall system. The task of the switching/steering system is to monitor, in an online fashion, the operation and performance of the advanced control and, if deemed necessary, switch to the fail-safe baseline control.

The design of a switching/steering system can be approached by modeling certain regions of the state-space as depicted in Figure 7. The figure depicts the unsafe part of the state-space, the complement of which is deemed safe. A certain subset of the safe states is depicted to be the target states; these are the states where the system is considered fail-safe (e.g., for a group of vehicles it is the set of states where the inter-vehicle distance satisfies a certain safe separation bound and also all the vehicles are stable).

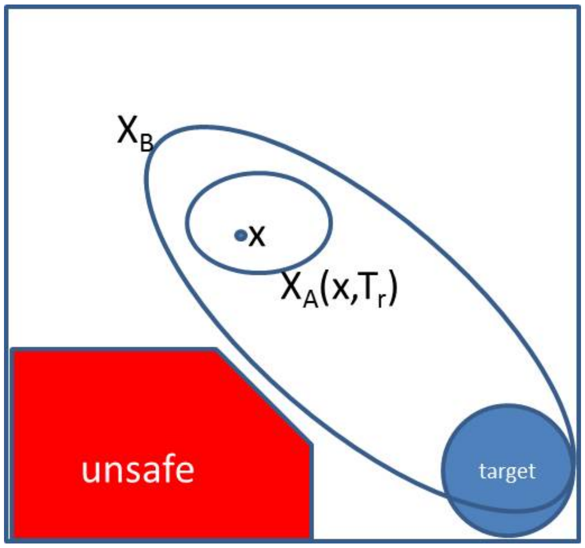


Figure 7: Safety region X_B vs. active safety margin $X_A(x,T)$

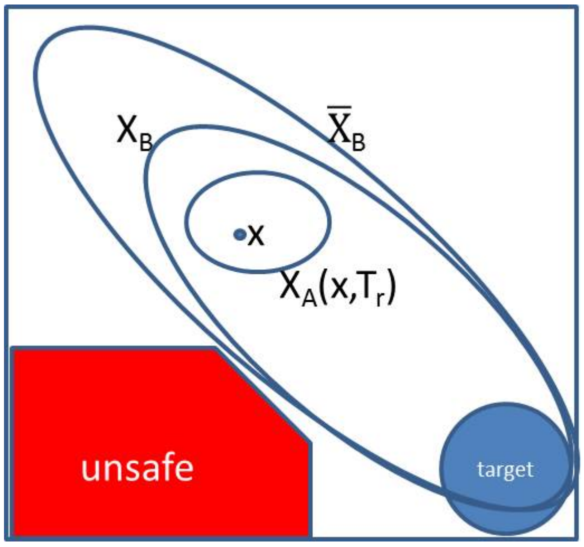


Figure 8: Enhanced safety region $\overline{X}_B$ vs. active safety margin $X_A(x,T)$

X_B denotes the *safety region* of the baseline control, and corresponds to the set of states from where the baseline controller can steer the system to the target states while always avoiding the unsafe states, and once the target states are reached, those can be maintained forever, i.e., can be made to remain invariant. It is clear that under the operation of the advanced control, the current state x must lie within X_B (i.e., $x \in X_B$) so as to be able to avail the safety of the fail-safe baseline control by being able to safely switch to it.

Associated with any cyberphysical system, such as shown in Figure 6, is a *reaction time* T, which is the extra time needed to perform a switch in control from the instance of recognizing that such a switch is needed to the instance of actual execution of the switch. The reaction time corresponds to the computation and communication delays associated with a given cyberphysical system and needs to be computed/estimated. It then turns out that, not only the current state x must lie within X_B, but an entire "ball" of states around x that can be reached under the operation of the advanced control within the reaction time T must lie within X_B. Thus associated with the current state x and the reaction time T, exists an *active region of safety margin*, denoted $X_A(x,T)$, that corresponds to the set of states reachable from x within time T under the advanced control. Thus for ensuring safety, it must hold at each current state x, reached under the advanced control, that $X_A(x,T) \subset X_B$, and otherwise a switch to the baseline control must be made. Thus another task for the switching/steering block is to perform monitoring to the estimate the current system state and bound the set of forward reachable states $X_A(x,T)$.

Further, since the advanced control is unverified, it may contain faults. So further online monitoring must be performed within the switching/steering block for detecting any fault in the advanced control, and a switch to the baseline control must be made once any such fault is detected. For fault-detection purposes, monitoring can be performed at multiple levels: At the advance control level to directly detect any of its faults, and also at the higher system levels to catch any advanced control faults that may manifest as failures at the higher system levels. Since not all failures at higher levels may correspond to faults in the advanced control, fault-detection must be accompanied with fault-identification. Thus a switch must be performed whenever:

$$[X_A(x,T) \subsetneq X_B] \vee [\text{advanced control fault detected/identified}].$$

In some cases, it may be possible to enlarge the safety region X_B to a larger set $\overline{X}_B$ by designing a steering control that can safely steer any state in $\overline{X}_B$ to a state in X_B and retain there for at least the duration of reaction time so upon reaching X_B, the steering control may be switched to the baseline control. Thus the rule for switching from the advanced to the steering control (and next switching to the baseline control) becomes:

$$[X_A(x,T) \subsetneq \overline{X}_B] \vee [\text{advanced control fault detected/identified}],$$

where the corresponding enlarged safety region $\overline{X}_B$ is depicted in Figure 8.

In summary, the design of the switching/steering block involves the following off-line and online computations:

Offline:

- Compute safety region X_B under baseline control
- Compute enhanced safety region $\overline{X}_B$ under steering control
- Compute control switch reaction time T
- Verify switching/steering system

Runtime:

- Estimate current state x under the operation of advanced control
- Estimate/bound active safety margin $X_A(x,T)$ associated with advanced control at current state x within reaction time T
- Monitor input/output behaviors at various levels of hierarchy to detect any violations of the specifications for those levels
- Perform detection and identification of any advanced control fault

The following models will be needed to be able to perform the above offline/runtime computations:

- Plant model
- Baseline control model to compute/bound X_B
- Real-time analysis models of cyberphysical infrastructure to estimate/bound T
- Advanced control model or its abstraction to compute/bound $X_A(x,T)$
- Input/output correctness properties of advanced controller and its controlled plant
- Models of noise distribution

3 State of the Art for Runtime Monitoring and Switching

Dr. Insup Lee and Dr. Oleg Sokolsky

This section provides a brief overview of the current research on runtime monitoring and switching. Section 3.1 introduces runtime verification from a history perspective, and it identifies various components required for runtime monitoring and switching. Section 3.2 elaborates prior work within each component.

3.1 Overview of runtime monitoring and switching

Runtime verification (RV) is a relatively new area of study that is concerned with dynamic monitoring, analysis and recovery of system executions with respect to precisely specified properties. A rigorous approach to proving the correctness of programs at run time was first presented in a paradigm called program checking by Blum and Kannan [85]. The papers on program checking also demonstrated that runtime monitoring was feasible in many instances where static, design-time verification does not appear to be. This was the primary initial motivation for this field at its inception about a decade ago. That is, despite the best verification efforts, problems can occur at run time, and dynamic analysis of the running system can improve confidence in its evolving behavior.

Since then, researchers came to realize that precisely specified monitoring and checking can be used for many purposes, such as program understanding, systems usage understanding, security or safety policy monitoring, debugging, testing, verification and validation, fault protection, behavior modification (e.g., recovery), etc. At the most abstract level, a running system can be regarded as a generator of execution traces, i.e., sequences of relevant states or events. Traces can be processed in various ways, e.g., checked against formalized specifications [86], used to drive the simulation of behavioral models, analyzed with special algorithms, visualized [87], etc. Statistical analysis [88] and machine learning of system properties can be performed over the traces.

The scope of RV research covers three conceptually separate aspects. The first aspect concerns checking of traces, such as property specification, and algorithms for the checking of such property specifications over a given trace. The second aspect has to do with the generation of traces; that is, how observations are made and recorded. Most RV systems rely on some form of instrumentation to extract observations. Finally, the third aspect is about feedback and recovery; that is, actions that are carried out when a safety violation occurs, e.g., raising an alarm or switching to a safe controller.

The three aspects of RV are not independent, however. A trace generation method has to ensure that all observations necessary for checking a property are recorded. A missed observation is likely to result in an incorrect checking outcome. On the other hand, generating irrelevant observations increases checking overhead and should be avoided. Recording exactly the right set of observations for a given property is the subject of trace generation research. This dependency between the checking and trace generation aspects may also be turned around: if the trace generation method is fixed, one can pose the question, which property specification language would be the most appropriate. For example, if the trace records state changes — that is, differences between two successive states — rather than values of state variables, checking a state-based property would require an additional step of reconstructing states. Likewise, what feedback and recovery actions are required and when they must be taken depend on the checking result. A false positive checking outcome may trigger unnecessary or undesirable switching actions, whereas a false negative one prevents recovery actions from being carried out when needed, which potentially leads the system to failure.

We distinguish three broad categories of RV approaches to property specification. In the first category, properties depend on execution history and their evaluation depends on a sequence of states in a trace. Such properties are often expressed in a variant of temporal logic; however, other specification languages are also used, such as regular expressions and state machines. The second category relies on the use of contracts or assume-guarantee interfaces between modules in the system. Properties in this category typically describe a single state or changes between two consecutive states in the trace. It may be considered a special case of the previous category. However, logics use for property specification as well as checking algorithms tend to be different here. Finally, the third category, sometimes referred to as specification-less monitoring, develops checking algorithms that detect violations of specific common properties. These properties typically relate to concurrent executions, such as freedom from race conditions [89], atomicity [90], serializability [91], etc.

An important research issue in runtime verification, which cuts across all RV aspects mentioned above, is the management of spatial and temporal overhead. An inefficient implementation of the monitoring and switching algorithm can easily be disruptive to system performance and limit the use of RV techniques. Such an implementation may also delay the system from switching to a safe state on time. A significant source of overhead is instrumentation that is necessary to extract observations from a running system. Reducing the number of instrumentation points reduces instrumentation overhead but may lead to missing observations and incorrect checking results, which may in turn result in system failure due to wrong or no recovery actions.

Last but not least, an RV framework has to provide the right feedback to the users. In many situations, it is not enough to signal that a violation has occurred. If checking is applied to a running system, as opposed to a recorded trace, feedback from the checker can help the system recover from a problem. Understanding the right feedback to produce, and reasoning about its effects on the system overall behavior is another important direction in RV research and especially critical in our RTA problem setting.

3.2 Categories of runtime verification

3.2.1 Monitoring and checking of temporal specifications

This category of RV tools have been developed as a direct extension of the static verification concept of model checking. Model checking algorithms [92] verify a model of the system against properties specified in a temporal logic or similar formalism. The original RV research question was whether an execution of the system can be verified at run time against the same set of properties. From the theoretical perspective, the problem is to deal with the fact that the trace is evolving as checking is being performed. Repeating the process from the beginning of the trace every time a new observation arrives is wasteful. On-line or incremental variants of the algorithms are necessary to make RV efficient. Another problem with respect to temporal properties comes from the finiteness of observed traces. Standard temporal logics, which are defined over infinite traces, and their corresponding checking algorithms need to be extended towards incomplete behaviors of finite traces.

Efficient on-line algorithms for checking a trace with respect to formulas in past- and future-time temporal logics, as well as regular expressions, have been available for about a decade [93, 94, 95, 86, 96] and are incorporated in a variety of tools [97, 98, 99]. One of the most extensive toolsets for runtime verification of formal specifications is the MOP framework [100]. It supports a variety of specification formalisms, such as temporal logics and finite state machines, and several monitoring targets, such as Java programs and bus snooping.

Comparison of RV algorithms for different specification formalisms reveals that they have much in common. All of the algorithms maintain a checker state that is updated when new observation from a target system arrives. For example, in the case of past-time linear temporal logic (ptLTL), the checker state contains the valuations of all subformulas of a given formula in the current state of the trace, which is updated using a dynamic programming approach [95].

This similarity between algorithms for seemingly different formalisms has led to the research on special monitoring logics that can be used as the common underlying formalism for specifying monitors. Eagle [101] is a logic with explicit fixpoint operators that is capable of implementing future- and past-time temporal logics, interval logics, extended regular expressions, and other formalisms that may be useful in the monitoring context. An execution engine for Eagle specifications makes it a flexible platform for implementing checkers. RuleR [102] takes this approach one step further by replacing explicit fixed point operators with rules that activate each other in response to observations. RuleR is lower-level logic that requires more effort to encode high-level semantics of a commonly used logic. However, it allows much finer management of the checker state, leading to more efficient checkers, and yields a system that is easier to implement and maintain.

The above temporal specification languages and corresponding checking algorithms can potentially be adapted for various layers of the system, especially at the execution platform and the control layers. At the layer of execution platforms, the above temporal approaches can be applied to check for timing behaviors and other performance metrics of the tasks executing on the platform, as well as to check for correctness behavior of the scheduling and communication protocols. At the control layer, the main challenge is to derive a mapping between the specification languages specified in the control-theoretic analysis of switching (Task 1) and the logics used by the runtime monitoring algorithm. Abstractions of the state variables specified in the control-theoretic properties will be required to derive a set of observable variables (directly or indirectly) in the monitoring specification languages such that the corresponding properties can be checked efficiently. For example, if stability of the control system is expressed as an invariant of the Lyapunov functions over the control state vector; the mapping in this case provides an equivalent “invariant” in the monitoring logics on the corresponding variables observed by the monitor.

One issue in monitoring control dynamical systems is how to efficiently and accurately monitor and check their continuous state values. Along this line, temporal logics for dense-time real-values signals and corresponding automatic generation of property monitors have been explored (see e.g., [103, 104]). Temporal properties in such logics specify the required behaviors of the dynamics of purely continuous or switched control systems, such as those modeled by sets of differential equations or hybrid automata. Although the proposed monitor generation algorithm proposed in [103] works only in offline setting, extensions of such logics (e.g., to incorporate events related to signal changes) as well as adaptions of the monitor algorithm to online setting are potentially useful in the context of switching in UAVs.

From the perspective of control analysis, many properties desired for system safety such as causality and stability can be expressed in simple non-temporal invariants over frequency domain. It would be interesting to explore lightweight transformation algorithms that derive spectral values from the observed traces to leverage on the spectral properties, which will in turn enable more efficient and effective checking algorithms. This transformation approach differs from the mapping approach described above in that the checkers compute the property based on the values of the transformed control-theoretic variables instead of the directly observed values.

3.2.2 Design-by-contract monitoring

Design by contract, pioneered in the Eiffel programming language [105], relies on interface specifications between components in the system. These specifications take the form of pre-conditions, post-conditions, and invariants. Contracts are typically state predicates (that is, expressions over the values of system variables in the current state). Contracts also typically allow us to relate “old” and “new” values of variables. Like any other correctness properties, contracts can be verified statically (within the inherent limitations of verification tools), but they often are also checked at run time for unexpected violations.

Unlike temporal specifications discussed earlier, contract checking is typically integrated more tightly with the system execution, and can conceptually be viewed as part of the system. Semantics of contracts determine, at which points the contracts should be checked. For example, a pre-condition for a method in an object-oriented program is checked when the method is called, and a post-condition is checked just before the method returns. Because of this tighter integration, instrumentation is typically not an issue.

Some languages, such as Eiffel and Spec# [106], provide special syntax for contracts. Other language frameworks support embedding of contract as code, usually providing a library of contract primitives. For example, the *Code Contracts* project [107] provides contracts for .NET languages as calls to methods of a Contract class. The TraceContract project [108] supports writing temporal specifications in Scala as calls to methods of a `Monitor` class. There are several comment-based contract frameworks for Java, such as JML [109] and Jass [110]. Both are based on writing specifications as special recognizable comments, which can then be extracted by tools. Some contract checking systems, Jass among them, extend standard contracts with *trace assertions* [111], which can be used to specify restrictions on sequences of invocations of methods of the class. With this extension, the distinction between contracts and temporal properties begins to disappear.

Due to the restrictive expressive power of the contract languages, design-by-contract monitoring is applicable to only simple properties. On the other hand, instrumentation and checking in this category of RV are much simpler compared to the monitoring and checking of temporal properties. Hence, this approach is potentially useful for checking component-level properties such as functional correctness, interface compliance properties (e.g., service requirements), or network communication and security protocols at the execution platform layer.

Since system-level properties are often concerned with the overall behavior of the entire system and over execution paths, they are typically not expressible by contract languages. A hybrid of the two monitoring design-by-contract and (event-triggered) temporal monitoring can help leverage the strengths of both techniques. It would be interesting to explore hierarchical monitoring, where violations at child components detected by the design-by-contract monitoring method are used as monitored events to the monitors of the parent components. An alternative hybrid technique is found in high-order temporal contracts [112]. Here, the contract systems are extended with temporal logics to express and enforce temporal properties. This integrated approach is particularly interesting as it leverages and combines techniques from both higher-order design-by-contract systems and the above first-order temporal systems within the same language.

Since contract checking is tightly integrated with the system execution, specific contract systems need to be designed for the implementation languages, which again imply language specific challanges. The tight integration of contract checking and the system execution also makes isolation of concerns between the system being monitored and the monitor itself difficult to achieve. Further, since assertions are hard to formulate for many kinds of system of interest – in particular, for adaptive systems – therefore, it may not be suitable for our setting.

3.2.3 Monitoring with uncertainty and system faults

The majority of RV tools assume full observability of system states. In complex real-time embedded systems, this may not always hold true, e.g., due to the high cost in sensing all variables reliably. Uncertainty in the system behaviors also arises due to the unreliability of hardware such as due to degradation over time or possible hardware defects. It is thus useful to perform runtime verification in presence of uncertainty in the system state information. RV techniques also need to account for the impacts of hardware and software faults on the quality of the observations.

One approach to incorporate uncertainty of the system behaviors is to use probabilistic models. In this context, the system behavior with respect to a given safety specification can be computed as the likelihood that the specification is being satisfied. A number of runtime monitoring algorithms for probabilistic properties based on statistical methods have been proposed, for instance, [113, 114, 115, 116]. Wang, *et al.* [117] extends the technique in [113] with Mont Carlo simulation for runtime verification of mixed analog and digital signal designs. Software runtime monitoring algorithms have also been extended to account for hardware behaviors [118].

To allow unobservable states in runtime monitoring, stochastic models with hidden states have been applied to the context of RV. For example, Wilcox, *et al.* [119] proposes a technique that relies on the probabilistic hierarchical constraint automata model [120] to capture probabilistic and faulty behaviors within both software and hardware. This technique integrates runtime monitoring techniques for temporal properties with HMM-based stochastic state estimation to detect safety violations during runtime. State estimation method has also been used in [121] to estimate the confidence of the verification results when sampling-based monitoring technique is used.

While existing RV tools in this category remain relatively few, uncertainty is a critical issue in complex systems such as UAVs. Integration of model-based diagnosis and other fault-tolerance techniques with runtime monitoring is a promising direction in tackling this issue. One challenge to be addressed is how to reliably derive the faulty model of the system. Since violations of the system model may violate the correctness of the monitoring algorithm itself, appropriate methods for checking at runtime the correctness of the model used are inherently necessary.

3.3 Instrumentation

In order to extract observations necessary for checking properties, RV tools rely on instrumentation. Most tools use active instrumentation, that is, insertion of code probes into the running system. One of the few exceptions is BusMOP [122], an instantiation of the MOP framework [100] for the monitoring of PCI buses. BusMOP uses passive instrumentation, directly observing traffic on the bus. In order to avoid missing relevant observations and allow RV to scale to large systems, instrumentation should be automatic. Automatic instrumentation requires us to perform analysis of the target system in order to identify where events of interest occur.

An increasingly popular way of implementing instrumentation of source code is through the use of aspect-oriented programming [123]. Indeed, multiple probes are added to the code that collectively supply observations to the monitor. Thus, instrumentation satisfies the definition of a cross-cutting concern, which underlies aspect-oriented programming. The use of aspect-oriented techniques in RV research was pioneered in the tools J-Lo [97] and Hawk [124]. An even closer connection between RV and aspect-oriented instrumentation was made in the AspectBench Compiler using the notion of *tracematches* [125]. A tracematch is an extension of the AspectJ language, which captures a regular pattern of events. In other words, it is similar to other temporal specifications discussed in Section 3.2.1. However, tracematches are given aspect semantics so that they can be automatically weaved by the compiler. Since then, several RV tools, including MOP [100] and Larva [126], rely on AspectJ for instrumentation. Aspect-oriented instrumentation for other programming languages is less common. The InterAspect system [127], which performs instrumentation based on the intermediate representation of the GCC compiler, thus providing aspect-oriented instrumentation for languages with GCC front-ends, in particular, C and C++.

Aspect-oriented programming provides a powerful way for instrumentation. Most existing aspect-oriented instrumentation systems, however, are developed for Java, which may not optimal for performance on embedded platforms. New aspect-oriented instrumentation techniques for embedded programming languages are hence necessary. Here, it would be interesting to explore the use of InterAspect instrumentation system [127] for GCC-based languages within RV tools.

Since monitoring code is added directly to the monitored program, one challenge when using aspect-oriented instrumentation approach is how to guarantee that the added code does not adversely change the behaviors of the original program in undesirable or unpredictable manner. In safety-critical systems, re-certification of the software after instrumentation will be necessary to confirm to certification standards. To address this, time-aware instrumentation techniques [128] have been proposed to preserve timing properties of the instrumented program. The idea is to transform the execution time distribution of the system to maximize the coverage of the trace while meeting the time constraints.

Based on the mode of interaction between the instrumented system and the checker, we distinguish between synchronous and asynchronous monitoring. In synchronous monitoring, whenever a relevant observation is produced by the system, further execution is stopped until the checker confirms that no violation has occurred. Synchronous monitoring may deliver a higher degree of assurance than the asynchronous one, because it can block a dangerous action. However, this comes typically with a higher instrumentation overhead. In some applications, where the system can tolerate a slower response, but effects of a property violation are dramatic [126], this additional overhead is justified.

3.4 Overhead reduction

Runtime checking of complicated properties that involve many system variables imposes high overhead on the system performance. There has been much work on reducing the checking overhead by combining static analysis techniques with subsequent runtime checking. We overview some of these approaches here; the relationship between runtime verification and timing is explored in more depth in Section 4.

The concept of combining static and dynamic analysis originates in the programming language community in the context of typestate analysis. Typestate properties are properties of paths in a program and are similar to behavioral specifications studied in the RV literature. A typical typestate property may be expressed as a state machine constraining valid sequences of API calls in a program. Residual typestate analysis has been proposed in [129]. It is based on the observation that, while static typestate analysis often leads to inconclusive results, it produces valuable information that can reduce the state machine that needs to be analyzed at run time.

Complementary to the residual analysis is the work of Eric Bodden and his colleagues [130, 131]. Here, static analysis of the code to be monitored is performed with the goal of identifying instrumentation points that can be safely removed. Results of this work have been implemented in the tool Clara [132].

Combination of symbolic techniques in automata-theoretic setting and compiler optimization techniques have also been explored to optimize runtime overheads of temporal monitors. Along this line, [133] proposed an approach for reducing overheads of runtime monitors for SystemC program. Here, overhead optimization is achieved by means of symbolic representations of both state and alphabet of the monitor, Satisfiability-Modulo-Theory solver, and state-folding technique.

In some cases, it may be acceptable to lower the accuracy of monitoring by missing some of the execution events. In these cases, there is a trade-off between accuracy and lower overhead. A control-theoretic approach to implementing this trade-off has been proposed in [134].

While the existing overhead reduction techniques are promising, their evaluations are primarily restricted within the context of the monitor and the component under monitoring. Effects of such monitoring overhead optimization in a broader context within which the monitor is being deployed need to be considered. One example is the connection between error detection and software health management, which has been discussed in [135], where both the monitor and the diagnosis module co-observe the program's behaviors. It is hence critical to ensure that the runtime monitor preserves the information required by diagnosis and vice versa.

3.5 Feedback and recovery

An important question when designing an RV system is what to do when a violation is discovered. If the system is undergoing testing, it may be sufficient to alert the operator, who will stop the system and diagnose the problem. However, in order to realize the vision of RV for a post-deployment alternative to the design-phase verification, a more programmatic approach is needed. In our RTA setting, this involves invoking a recovery mechanism that switches the system from the currently unsafe state to a safe state.

Several RV systems, including MaC and MOP, have the ability to invoke user-specified recovery routines. Monitoring specifications allow the user to specify, what actions are to be invoked when a particular violation occurs, and what information is to be passed to the recovery routine. The system can be partially reset or switched to a failsafe mode. In some situations, for example, when monitoring performance and quality-of-service properties [113], the system can be steered to a more suitable state. For safety property, the Simplex architecture [136] proposes to use a recovery controller to steer the system to a safe state. Specifically, when a safety violation occurs, the system's controller is switched (from the current unsafe controller) to a recovery controller, which will then bring the system to a safe state within the area of operation of a fully verified baseline controller.

The above approach relies on an implicit assumption that the recovery routine will be effective, regardless of the reason the property was violated. For a property that depends on the interaction of several parts of a system, this assumption may not hold. To give a simple example, suppose the property is that the size of a queue, buffering traffic from a sender to a receiver, should be always below a threshold. If the property is violated, it may mean that the sender is sending the messages too fast or that the receiver is processing them too slow. Different recovery actions may be warranted in each of these cases.

A possible way to solve this deficiency is to incorporate diagnostic facilities into an RV system. The first work to explore this line of research was [137]. A more efficient approach, that combines on-line and off-line techniques has been studied in [138]. Using safety cases as a guidance for detecting property violation and designing recovery mechanism, such as the approach employed in [139], is a promising direction for providing effective feedback mechanism.

Instead of having the monitor react to a property violation, it is sometimes possible to prevent the violation by delaying or reordering events, or by preventing certain events from happening. This extension of the RV research has come to be known as *runtime enforcement*. In general, some events may not be under the control of a monitor. However, runtime enforcement has proved very effective in many important areas such as security. The power of an enforcement monitor determines the class of properties that can be enforced. Blocking of events is sufficient to enforce safety properties [140], while the capability to delay events allows one to handle some liveness properties as well [141]. Falcone, *et al.*, [142] generalize existing enforcement approaches in a single framework and further extends the class of enforceable properties.

3.6 Challenges in applying runtime monitoring and switching in RTA

All of the research problems considered in this section, such as design of efficient property checking algorithms, overhead reduction, checking under limited observability, specification and implementation of feedback and recovery, remain applicable in the RTA context. Here, we discuss an additional challenge that is specific to RTA.

The question of *what properties to monitor* always arises in the design of a runtime verification layer. In a typical runtime verification scenario, we monitor correctness properties of the system. The same properties are considered during design-time verification efforts, thus there is no disconnect between properties that the system needs to satisfy and properties that are checked by the runtime verification layer.

By contrast, in the RTA setting, the properties that need to be assured by the RTA framework are different from the ones to be monitored by the monitoring and swithching layer. To avoid confusion, we call the former *assurance properties* and the latter *monitoring properties*. Assurance properties are typically system-level properties, such as safety, stability, possibly performance and quality of service. They are often not monitorable directly. For example, a stability property is related to future time, while runtime verification, effectively, monitors the past. Even when the assurance property is directly monitorable, for example in the case of safety properties, we cannot simply monitor for the violations of the property. Instead, the monitoring layer needs to raise an alarm and invoke the switch to a safety controller so that the violation is avoided. Thus, a somewhat different property needs to be monitored. The monitoring property that corresponds to a given assurance property needs to be predictive in real time: a violation of the monitoring property has to occur before the assurance property is violated, but when such a violation is imminent. Currently, there are no general techniques for converting assurance properties into monitoring properties. Some answers may be provided by techniques considered in Section 2, but more general approaches are needed.

4 Timing Constraints and Worst Case Execution Time

Dr. Lee Pike

RTA implementations must take into account the operational environment of the system under observation. Such environmental considerations include, for example, reliability constraints, physical constraints (i.e., size, weight, and power (SWaP), and as we focus on in this section, timing constraints. Time is a fundamental aspect of cyber-physical (CPS) systems [143]. While timing is difficult enough to reason about in the traditional embedded setting, these difficulties are exacerbated in the context of RTA.

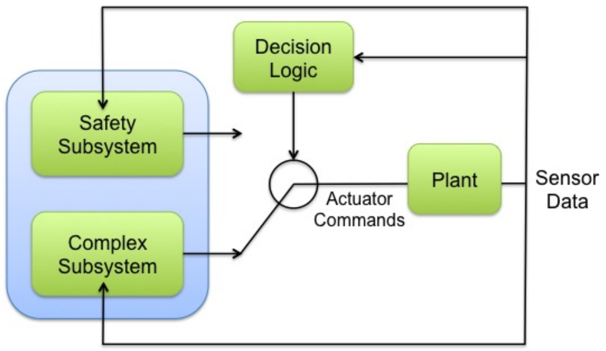


Figure 9: Simplex architecture (reproduced from [144]).

Without loss of generality, consider RTA in the context of the Simplex architecture [145, 144], depicted in Figure 4. There, we have a complex controller and a simple (safety) controller. The runtime monitor implements the decision logic determining whether actuation commands should be switched from the complex to the simple controller (in an implementation, the decision logic also includes the switching logic to handle the switch-over implementation).

Timing considerations arise from each component in the architecture as well as cross-cutting concerns that apply to the entire system. Consider, for example, the following list of ways in which timing is affected:

- *Complex-controller computational complexity*: the time complexity of the algorithm being monitored. For example, is the computational time constant ($O(1)$), linear ($O(n)$), or perhaps more expensive? How does the average time-compare to the worst-case time?
- *Monitor complexity*: the time complexity required to monitor the specified properties.
- *Switching logic complexity*: the time complexity of the Simplex switching logic.
- *Software implementations*: the timing effects of control-flow, memory management, and algorithm implementations for each software component (the complex controller, runtime monitor, and switching logic). Some timing effects will also depend on the hardware—for example, if the hardware contains a floating-point unit (and the algorithm depends on floating point), these computations can be performed more efficiently in hardware.
- *Hardware nondeterminism*: timing nondeterminism introduced by clock drift, cache usage, pipelining, etc. Additional nondeterminism may be introduced by hardware interrupts and delays caused by peripherals.

- *Faults*: effects of stochastic faults in the hardware affecting timing.

The cumulative effects are extraordinarily difficult to control given that they occur within different layers of abstraction in a system. In this section, we explore the problem of timing with respect to RTA. We first describe background concepts about timing and scheduling in Section 4.1, providing pointers into the vast literature on the subject. In Section 4.2, we overview work in runtime monitoring that has specifically focused on being time-aware. We close by pointing to challenges and future work in the area in Section 4.3.

4.1 Timing Concepts

Here, we briefly overview some timing concepts relevant to timing-aware RTA, focusing on concepts that relate to *hard real-time* systems in which deadlines must be strictly met. We overview scheduling algorithms, worst-case execution time, networking, and faults.

4.1.1 Scheduling

In our context, scheduling algorithms provide a mechanism for logically distinct programs to share a single computational resource. RTA approaches in hard real-time systems will deal at some level with scheduling. For example, one implementation of the Simplex architecture would be to have the simple controller, the complex controller, and the decision logic be scheduled on a single processor. At the least, it is likely that the complex controller is one of multiple programs scheduled on a processor.

We call scheduled programs *tasks*. The *scheduler* is itself a task that determines which task to execute next, given some scheduling algorithm. Scheduling can be *cooperative* or *preemptive* (or a combination of the two). In cooperative scheduling, tasks themselves are responsible for yielding to the scheduler. In preemptive scheduling, tasks can be interrupted at arbitrary points of execution (from the interrupted task's perspective).

Scheduling does not require the use of an operating system; for example, the following is a cooperative round-robin scheduler for tasks `foo` and `bar`, implemented as a loop in C:

```c
int main(void) {
  while(1) {
    foo();
    bar();
  }
  return 0;
}
```

The following are three of the most common scheduling algorithms used for preemptive scheduling in hard real-time systems, and particularly in real-time operating systems (RTOS); see [146] for details. In particular, we only consider *fixed-priority scheduling* in which each task is given a priority at design-time from some partial order of priorities. Dynamic-priority scheduling is more complex and not even available in some RTOSes.

- *Round-robin scheduling*. One of the simplest scheduling algorithms, round-robin scheduling provides each task a fixed duration to execute and schedules tasks in a fixed order.

- *Fixed-priority preemptive scheduling*. In round-robin scheduling, there is no notion of prioritization among the scheduled tasks. In fixed-priority preemptive scheduling, each task is given a fixed priority from some partial order of priorities. After some fixed time duration, the scheduler preempts the currently-executing task and schedules the highest priority task, from among those that are *enabled* (i.e., waiting to execute), to run next. (If more than one enabled task is at the highest priority, then the scheduler may nondeterministically choose from among the set.)

  Of the preemptive algorithms, fixed-priority preemptive scheduling is one of the simplest but it can be effective to meet many hard real-time constraints.

  The principal weakness of this algorithm is the danger of *starvation*, where a low-priority enabled task is never executed. Various modifications to the algorithm help reduce the risk of starvation.

- *Rate-monotonic scheduling*. In rate-monotonic scheduling (RMS), tasks are assumed to have fixed *periodic deadlines*. For example, if a task T has a periodic deadline of 100ms, then T must complete one execution at least every 100ms. In RMS, tasks are assigned priorities inversely proportional to the size of their periodic deadlines, so that tasks with shorter deadlines have higher priorities.

  A theoretical result of RMS is that it is *optimal* insofar that if any static priority algorithm can schedule tasks to meet their periodic deadlines, then RMS can, too. However, RMS suffers the risk of "priority inversion" in which a higher-priority task is blocked indirectly by a lower-priority task. This can happen when there are shared resources between tasks; see [147] for details. Priority inversion is not hypothetical: a Mars Pathfinder bug was the result of priority inversion [148].

4.1.2 Worst-Case Execution Time

Worst-case execution time (WCET) is the maximum amount of time required for a program to complete; see [149] for a survey of the state-of-the-art. WCET is only valid with respect to a hardware execution environment. Two major criteria are used to evaluate WCET approaches: (1) whether they produce estimated or hard bounds for performance (*safety*) and (2) the *accuracy* of the bounds generated [149]. The usual objective of WCET analysis is to find the most accurate over-approximation—scheduling based on an under-approximation can cause deadlines to be violated.

WCET can be analyzed by either testing or static analysis. Testing may not produce an over-approximation if test-cases do not exercise the most expensive portions of the program. The difficulty of testing is exacerbated by the unpredictability of modern hardware including pipelining architectures and cache memory.

In static analysis, a model of program execution on a hardware platform is used. Static analysis is also difficult due to the unpredictability of modern hardware; as well, static analysis must symbolically execute software. A number of academic and commercial WCET tools have been developed [149].

4.1.3 Networking

Typically, CPS systems are distributed systems in which computation nodes communicate. Communication can be *synchronous* in which every node shares a *global clock* and so has the same time reference, or *asynchronous* in which there is no shared time reference. A global clock does not have to be physical; a clock synchronization algorithm [150] can implement a global clock by ensuring that each node's local clock is synchronized within a small error bound.

Safety-critical systems, like flight-control systems, usually implement synchronous communication. Many fault-tolerant algorithms require a synchronous assumption; moreover, synchronous systems are easier to reason about [151, 152]. In particular, modern fault-tolerant data-buses, such as NASA's SPIDER [153] and the Time-Triggered Architecture [151], ensure strict synchronization among non-faulty nodes.

4.1.4 Timing and Faults

Finally, there is a subtle relationship between faults and timing. For example, *Byzantine faults*, in which two receivers interpret the same message differently, are the most nefarious class of faults masked in distributed systems [154]. While sometimes considered an academic curiosity, Driscoll *et al.* at Honeywell report that in fault-injection experiments of a fault-tolerant network, Byzantine faults resulting from slightly-unsynchronized systems were observed:

> The dominant Byzantine failure mode observed was due to marginal transmission timing. Corruptions in the time-base of the faultinjected node led it to transmit messages at periods that were slightly-off-specification (SOS), i.e. slightly too early or too late relative to the globally agreed upon time base. A message transmitted slightly too early was accepted only by the nodes of the system having slightly fast clocks; nodes with slightly slower clocks rejected the message. [155].

Conversely, hardware faults can cause cascading timing issues throughout a system, which can affect assumptions about WCET. See [156] for an overview of approaches to architectural-level fault-tolerance.

In any case; reliable controllers will be fault-tolerant, and faults will be one cause of property violations in a monitor. RTA must be fault-aware.

4.2 Timing-Aware RTA

In the following, we survey research in RV that is time-aware. Broadly, this includes work that attempts to monitor timing properties as well as work that attempts to control the unpredictability of timing within the monitor. For a general overview on the topic, Goodloe and Pike have written a survey of RV in the context of real-time systems [157].

In the following, we begin by discussing RV approaches to monitor timing properties. Then we discuss a range of approaches to *control* the timing overhead of RV.

4.2.1 Monitoring Time and Real-Time Logics

An early application of online monitoring to distributed real-time systems focused on verifying that timing constraints are satisfied; the approach is based on clock-synchronization and time-stamps [158]. Many variants of temporal logics have been developed for specifying properties of real-time systems; a survey of these logics is given in Alur and Henzinger [159]. Several efforts in the monitoring community have focused on monitoring metric temporal logic (MTL) specifications of real-time systems [160]. MTL can be used to reason about quantitative properties over time. For instance, one can specify the time elapsed between two events.

Mok and Liu developed the language Real-Time Logic (RTL) for expressing real-time properties for monitoring. A timing constraint specifies the minimum/maximum separation between a pair of events. A deadline constraint on an event E_1 and an event E_2 is violated if the event E_2 does not occur within the specified interval after event E_1. To monitor RTL constraints, Mok *et al.* employ sensors that send timestamped events to the monitor, and an algorithm computes the satisfiability of the constraint [161, 162].

An application in the area is to telecom switches that have real-time constraints, and approaches to monitoring these are investigated by Savor and Seviora [163]. The authors introduce a notation for specifying time intervals in the Specification and Description Language (SDL). Algorithms are developed for processing different interleavings of signals as well as to monitor that the specified signals occur within their designated timing intervals.

4.2.2 Time-Controlled RTA

We describe three classes of approaches for controlling the timing overhead in RTA: time-triggered monitoring, networked monitoring, and probabilistic monitoring. In general, this area of research is young with a number of promising approaches:

Time-triggered monitoring. *Time-triggered* computation is the idea of events being triggered by the passage of time rather than by other events occurring [151]. Time-triggered systems are fundamentally synchronous systems, and the design philosophy has been adopted in safety-critical distributed-system design [151, 164].

Time-triggered monitoring is an approach in which the state of the system being observed is sampled at time-triggered intervals. Until recently, monitoring based on sampling state-variables has largely been disregarded as a runtime monitoring approach, for good reason: without the assumption of synchrony between the monitor and observed software, monitoring via sampling may lead to false positives and false negatives [165]. For example, consider the property $(0;1;1)^*$, written as a regular expression, denoting the sequence of values a monitored variable may take. If the monitor samples the variable at the inappropriate time, then both false negatives (the monitor erroneously rejects the sequence of values) and false positives (the monitor erroneously accepts the sequence) are possible. For example, if the actual sequence of values is

$$0, 1, 1, 0, 1, 1$$

then an observation of

$$0, 1, 1, 1, 1$$

is a false negative by skipping a value, and if the actual sequence is

$$0, 1, 0, 1, 1$$

then an observation of

$$0, 1, 1, 0, 1, 1$$

is a false positive by sampling a value twice.

Time-triggered sampling is an effective approach to reduce monitoring overhead [128, 166, 167, 168]. In the approach, the runtime verification system evaluates a given property based on reconstruction of the execution path of the observed program from the sampled state variables. There are a few challenges to this approach, however. One challenge is approximating the minimal sampling period required given a property and monitored program. When a minimal sampling period is determined, it may be too small to be efficiently used—for example, if a state variable is updated in a tight loop. In this case, the sampling period can be extended by adding auxiliary state to store the “history” of state variable changes. Approaches for finding optimal trade-offs between additional state and sampling periods are also explored in the work.

While the research described above attempts to apply the time-triggered approach to arbitrary programs, in a synchronous model of computation the problem becomes somewhat simplified. Synchronous languages are data-flow languages without hidden state, much like in functional programming. Thus, the problem of determining control-flow is vastly simplified. The synchronous model of computation is well-suited for designing control-systems and other embedded programs [169, 170, 171]. In his manifesto on the need for time-aware computing in CPS, Edward Lee notes the benefits of synchronous programming:

> The synchronous languages [171], such as Esterel, Lustre, and Signal, do not have explicit timing constructs in them, but because of their predictable and repeatable approach to concurrency, can yield more predictable and repeatable timing than most alternatives [143].

Synchronous programs can easily monitor other synchronous programs without additional overhead since synchronous programs are inherently parallel. A language called LOLA specifically focuses on monitoring synchronous systems [172].

For control systems that deliver periodic control signals, a monitor can observe it in a time-triggered fashion, regardless of its implementation. This is the approach taken in Copilot, a synchronous language for monitoring hard real-time systems [173, 174]. Copilot generates C programs from synchronous specifications; the C code is guaranteed to use constant memory and be constant-time, modulo perturbations due to the hardware platform. Copilot is implemented as an *embedded domain-specific language*, meaning that it is actually a library, with its own syntax, embedded in the high-level language Haskell. This approach simplifies the effort to ensure the compiler is high-assurance [175].

While simple, the synchronous approach can be suitable if the monitor must only sample control signals directly to calculate a property. Sensor values change at a slow rate relative to modern processor speeds. For example, Copilot was used to sample (injected) faults in a fault-tolerant airspeed sensor system [174]. Indeed, any fault that lasted on the order of microseconds is likely a transient fault, which may be best to simply ignore [176].

Networked monitoring. The obvious approach to ensure that monitor has no overhead on the system being observed is to execute it on separate hardware. Such monitors are called *out-line monitors*. the BusMOP project [177, 178] takes this approach, in which high-speed monitors are executed on FPGAs to verify properties of a PCI bus. The monitors observe data transfers on the bus and can verify if safety properties are satisfied.

As noted in Section 3.3, this approach is quite unique as compared to other RV approaches. The main weakness of this approach, however, is that the monitor can recover state from messages sent over the bus (one avenue of research would be automatic light-weight instrumentation of programs to send necessary state over the bus). As well, approaches to state-estimation (described below) might be combined with a BusMOP-like approach to recover hidden state.

Probabilistic time-aware monitoring. The concept of probabilistic RV is discussed briefly in Section 3.2.3. Here we focus on its relation to controlling monitoring overhead. In 2011, Stoller *et al.* used probabilistic monitoring to control monitoring overhead [121]. The approach, called *state estimation*, computes a Hidden Markov Model (HMM) model for a given system being observed and property. In this context, the observations are traces observed by the monitor, and the hidden states are (some abstraction of) the states of the program. The idea is that while a monitor might make time-triggered observations and potentially miss trace elements between samples, using the HMM, a probability of the property being satisfied, given the observed trace, can computed. The approach requires learning the HMM in advance. Techniques exist to learn an HMM automatically, but they require being exposed to sufficiently many traces of the system behavior.

The technique is a novel approach to controlling monitoring overhead while recovering information about the program's hidden state. However, the approach is in its infancy; one experiment was presented in the paper, based on a model of a portion of the Mars Rover software (in the model, there were only six hidden states). For example, some directions for future research include determining what sort of probability distributions the approach works for—failures may be so rare that inaccurate probabilities are learned during the learning phase. Also, it needs to be determined the extent to which the approach scales-up. Finally, it may be possible to combine state estimation with the time-triggered RTA approaches described above.

Summary of approaches. We summarize the three approaches described above in Figure 10. We consider three criteria: (1) whether the approach *proves* a property (for a trace), (2) whether the approach requires instrumentation of the system being observed, and (3) to what extent control-flow (vs. data-flow) properties can be monitored.

As can be seen, no one approach is a panacea. The time-triggered and networked approaches may be used to prove properties, but only if there is assurance that value changed between samples (or the control-flow can be reconstructed from observations). The time-triggered approach may require modest instrumentation, whereas the other approaches do not. Finally, the networked approach cannot be used to monitor control-flow intensive properties (without extra instrumentation), whereas the time-triggered and probabilistic approaches can do so.

	Property proof	Requires no code instrumentation	Control-flow properties
Time-triggered	depends	depends	✓
Networked	depends	✓	χ
Probabilistic	χ	✓	✓

Figure 10: Comparing three time-aware RTA paradigms.

4.3 Recommended Future Research

While timing has always been a critical aspect of embedded computing, timing analysis has mostly been relegated to a second-class citizen in programming and development paradigms rather than something that can be reasoned about and controlled directly. The state of affairs within the runtime verification community is no different. Beyond the general need to promote time to be a first-class concept in computer science and particularly in CPS [143], the following are specific research directions within RTA.

1. *New metrics*. To control time, we have to first be able to measure it. Current metrics, like worst-case (or average-case) execution time, are too course-grained and make comparing trade-offs between approaches and abstraction levels difficult. As a simple example of new

kinds of metrics, Stoller *et al.* define the concept of *monitor overhead* to be M/R where R is the execution time of the program and $M + R$ is the execution time of the monitor and program together [121]. Other metrics might measure the variability in time-triggered monitoring due to control-flow (e.g., [168]), maximal vs. actual efficiency of monitors, or an efficiency function based on parallelism.

2. *Empirical studies*. The RV approaches described above—such as time-triggered runtime monitoring or probabilistic monitoring—are suitable for some tasks but not others. Having a better sense of the real-world use-cases for monitoring influences the kinds of RV research pursued. For example, the following are just a few classes of questions we do not have good data for currently:

   - What are the characteristics of the controller being monitored? What are the characteristics of its implementation. Are there unbounded loops? Is there data-dependent control-flow? What does its memory footprint look like?
   - What are the characteristics of the properties to be monitored? Are the properties predominantly control-flow properties (e.g., properties on program modes or states) or data-flow properties (e.g., temporal properties on sensor values)? What are the timing properties of the monitor itself?
   - What false-positive/false-negative rates are acceptable? It is not even clear what sort of distributions of events might lead to false-positives and false-negatives in different environments. Research here relates to reliability analysis for fault-tolerant systems [179, 180].

3. *Multi-core real-time monitoring*. Multi-core is the future of computing hardware, even for embedded systems. To what extent can multi-core technology be used to alleviate the problems of timing interference in monitoring?

4. *Timing analysis/runtime monitoring integration*. Worst-case execution time (WCET) analysis is a standard part of real-time system embedded development (described in Section 4.1.2). There has been little on integrating RTA monitors within the WCET analysis workflow. WCET analysis is applicable to multiple components of a Simplex RTA architecture, including the complex and safety controllers, the switching logic, and the monitor itself.

5. *Assurance, certification, and RTA.* The goal of RTA ultimately is to provide additional evidence about the assurance of a fielded system. To date, little work has been done to understand how assurance gathered via RV relates to assurance gather by means such as certification [181, 182] or formal verification [183]—one notable exception is [184]. Indeed, one attraction of RTA is to *improve* the assurance of an already-fielded system. How to systematically add RV to a system without having to re-certify the system in its entirety is unknown. The re-certification problem is particularly germane to timing, as timing (so far) has proven to be a non-compositional property.

6. *Scaling-up and system integration*. RV is largely a field of research with few industrial applications. Consequently, much of the work has been strongly theoretical and even in cases in which tools exist, they have been applied to relatively modest case-studies. Research is necessary to determine the extent to which techniques can be integrated into an overall system design. This may include, for example, integration existing middleware (e.g., within an RTOS) or within data-bus controllers [152]. Significant work is need to be able to provide RV

as a architectural level service, together with off-the-shelf monitors, that does not have to be redesigned from scratch in each instance.

5 State of the Art in Model Based Design for RTA

Dr. Xenofon Koutsoukos and Dr. Joseph Porter

5.1 Research Objectives

The development of safety-critical embedded control systems in complex engineering systems such as aircraft, robotics vehicles, and automobiles is challenging because of the high degree of uncertainty and variability in the interactions with the physical world. Verification and validation of embedded control software requires analyzing functional and behavioral correctness in the presence of multiple sources of nondeterminism due to the interactions with the physical environment. Model-based design provides a flexible framework to address essential needs of embedded control systems. Models are used not only to represent and design the system, but also to synthesize, analyze, verify, integrate, test, and operate embedded control systems. Model-based design focuses on the formal representation, composition, and manipulation of models in the design process [185]. Model-based design tools allow simulation, rapid prototyping, automatic code generation, testing, and verification.

Fig. 11 represents a simplified model-based design flow for a system composed of a physical plant and a embedded control system. In a conventional design flow, the controller dynamics are synthesized with the purpose of ensuring stability and safety and optimizing performance. The selected design platform (abstractions and tools used for control design in the design flow) is frequently provided by a modeling languages and simulation tools such as Matlab, Simulink, and Stateflow [186]. The controller specification is passed to the implementation design layer through a “Specification/Implementation Interface”. The implementation in itself has a rich design flow that we compress here into just two layers: System-level design and Implementation platform design. The software architecture and its mapping on the (distributed) implementation platform are generated in the system-level design layer. The results - expressed again in the form of architecture and system models - are passed on through the next specification and implementation interface to generate code as well as the hardware and network design. This simplified flow reflects the fundamental strategy in platform-based design [187]: Design progresses along precisely defined abstraction layers. The design flow usually includes top-down and bottom-up elements and iterations (not shown in the figure).

Verification is one of key components of model-based design. Verification involves analysis and reasoning using models of the system in each step of the design to ensure that it satisfies the specifications. Specifications are typically expressed as logic formulae and they are checked at design-time using verification methods including simulation, testing, model-checking, and theorem proving. Model-based design techniques are gaining increased attention because they allow the decomposition of the design process into an iterative progression from requirement models to implementation models using the repeated step of model construction, model verification, and model transformation. In this process, verification techniques can focus on the semantics of a suite of abstract system models that are incrementally refined into implementation models.

In spite of significant progress in model-based verification techniques, verification of heterogeneous systems such as aircraft control systems remains a very hard problem. The primary unsolved challenges are scalability and applicability in emerging system categories that have complex nondeterministic dynamics. Such complex components are often used in the system design in order to satisfy performance requirements. A typical example is a controller for an aggressive aircraft maneuver that results in complex behavior that cannot be verified at design time.

Run-Time Assurance (RTA) provides a complementary approach to traditional verification methods. The main idea is to allow the use of unverifiable components but determine appropriate

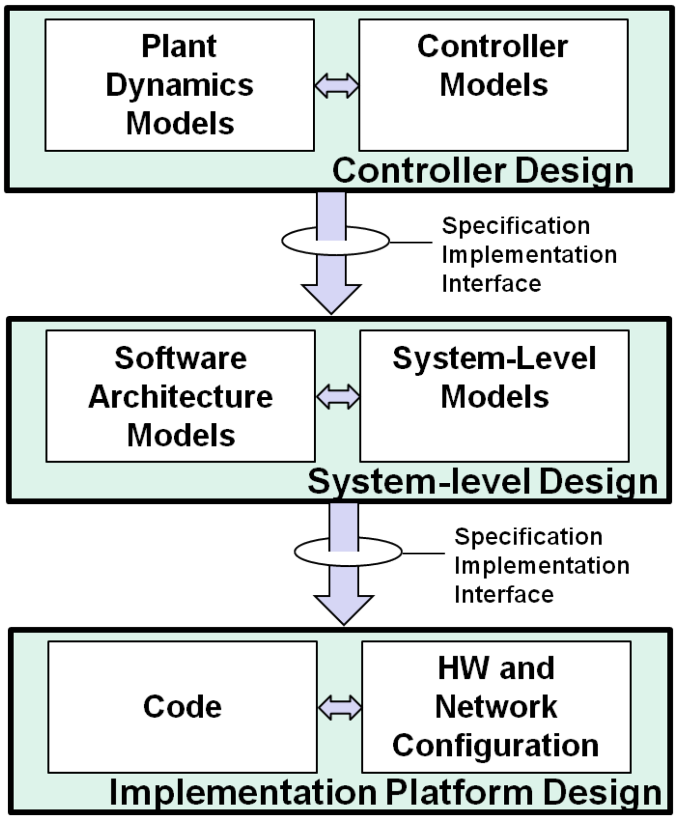

Plant Dynamics Models
Controller Models
Controller Design
Specification Implementation Interface
Software Architecture Models
System-Level Models
System-level Design
Specification Implementation Interface
Code
HW and Network Configuration
Implementation Platform Design

Figure 11: Simplified model-based design flow.

bounds on their behavior in order to design run-time control and reconfiguration strategies that ensure safety of the overall system. In addition to characterizing safe state sets at design-time, the systems needs to satisfy several run-time properties that include efficient and accurate system monitoring and state estimation, bounding the predicted system behavior for a short time horizon, online fault diagnosis and isolation, and switching and reconfiguration. The overall system behavior can be described by a (distributed) hybrid dynamical system. Verification of system-level requirements such as safety depends on a set of run-time properties for monitoring the system state and switching between different modes of operation that need to be satisfied by the implementation.

Effectiveness in model-based design largely depends on the degree to which design concerns (captured in the different abstraction layers) are orthogonal, i.e., how much the design decisions in the different layers are independent. RTA introduces major challenges in this regard. The controller dynamics are typically designed without considering runtime properties (e.g., numerical accuracy of state estimation and monitoring software components and time and memory requirement of online reachability algorithms). Runtime properties of the implementation emerge at the confluence of design decisions in software componentization, system architecture, coding, and hardware and network design choices. However, system-level specifications such as safety in RTA depend on runtime properties of various system components. Thus, design in one layer depends on a web of assumptions and properties to be satisfied by other layers. For example, safety depends on the ability to estimate the current state of the system, bound the set of reachable states from the current state given the current controller, check at runtime how close the state is to the boundary of the reachable states, and switch to a different controller if needed. Implementation properties such as numerical accuracy and computational efficiency of state estimation, online reachability computation, monitoring, and switching affect control design properties related to how close to the unsafe boundary the system can be allowed to go. Since runtime properties can only be evaluated after implementation, the overall design needs to be verified against the system requirements also after implementation. Even worse, changes in any layer may require re-verification of the full system.

Fig. 12 illustrates the application of the model-based design flow to RTA. A hybrid dynamical system is synthesized to satisfy system-level requirements such as safety. The hybrid system should be viewed as a specification that captures the behaviors as the system switches between various controllers to ensure safety. Based on this hybrid system specification as well as environment models, hardware description, and resource constraints, model-based design aims at synthesizing software components so that the system implementation is consistent with the specification. The software implementation includes monitors for testing runtime properties. The runtime monitors are used, in turn, for switching between different modes of operation according to the hybrid system specification. These interactions between design-time hybrid dynamical models and runtime implementation models are not typically addressed in traditional model-based design.

Interactions across the abstraction layers create significant challenges in applying model-based design for networked embedded control systems. An increasingly accepted way to address these problems is to enrich abstractions in each layer with implementation concepts. An example for this approach is TrueTime [188] that extends Simulink with implementation-related modeling concepts (networks, clocks, schedulers) and supports simulation of networked and embedded control systems. While this is a major step in improving designers' understanding of implementation effects, it can be used mainly for simulation-based analysis. A control designer can now factor in implementation effects, but can not facilitate verification techniques. Decoupling the design layers is a very hard problem and typically introduces significant restrictions and/or over-design. For example, the Timed Triggered Architecture (TTA) orthogonalizes timing, fault tolerance, and functionality [189], but it comes on the cost of strict synchrony, and static structure. A passivity-based design approach

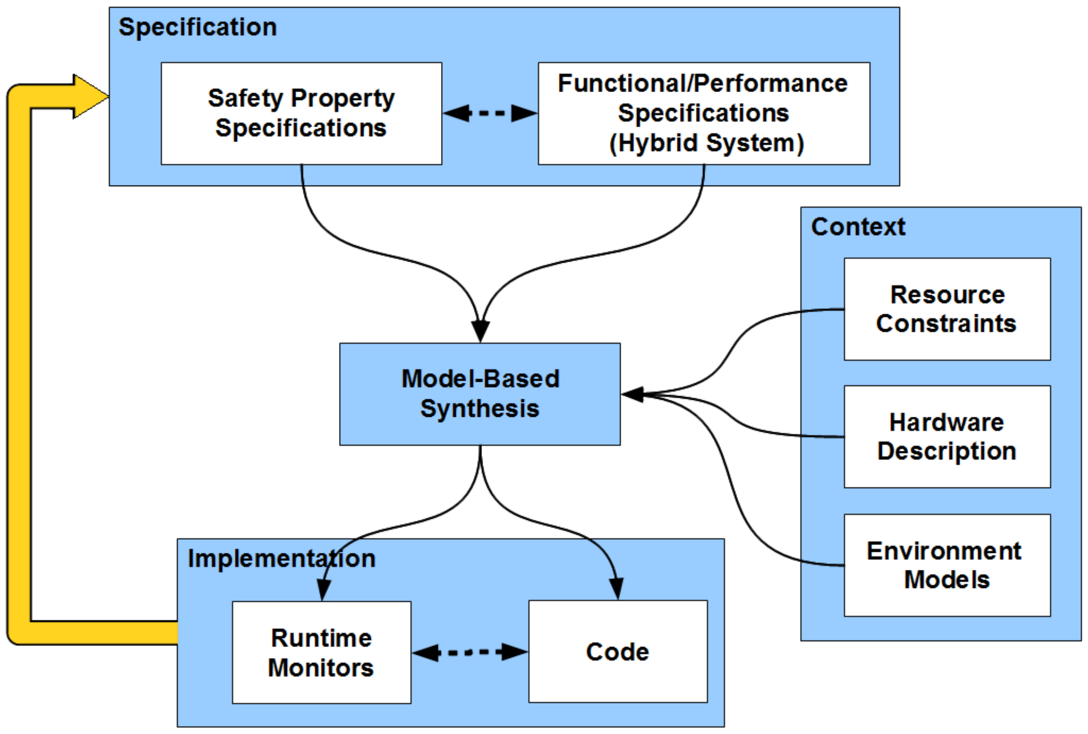


Figure 12: Model-based design flow for RTA.

that decouples stability from timing uncertainties caused by networking and computation is presented in [190]. Cross-domain abstractions that provide effective solutions for model-based fully automated software synthesis and high-fidelity performance analysis are also presented.

The approaches described above provide some capabilities for decoupling the various design concerns but they are not adequate for addressing the challenges introduced by RTA. The interaction between design-time specifications and runtime verification properties arises in several aspects of model-based design and create challenges that need to be addressed. In the following, we identify the main aspects of model-based design that are likely to require significant research developments to support RTA. First, we describe frameworks, languages, and tools used in model-based design of embedded control systems. Then, we focus on software design and code generation. We also discuss how methods for behavioral analysis, dependability analysis, and model-based testing can be used in model-based design. We also describe case studies in related application domains that are candidates for RTA and can be used for evaluation of the methods to be developed. Finally, we conclude with open challenges that need to be addressed in order to apply model-based design techniques for RTA.

5.2 Modeling Frameworks and Languages

Modeling languages and tools are pervasive in the embedded systems community, but RTA presents a number of challenges for the use of model-based engineering and model-based real-time software development.

5.2.1 Foundational Aspects

Model-based engineering offers several promising developments which may be extended or adapted to address the problems inherent in RTA applications.

- **Customizing abstractions for DSMLs**

  Model-integrated computing (MIC) relies on the design of domain-specific modeling languages (DSMLs) to capture abstractions from different facets of the design process, and ensure that their syntactic and semantic details are integrated consistently between design models and tools [185]. DSML domains can be as focused as processor details or the input language for a particular verification tool, or as abstract as generic hierarchical functional design diagrams. A common use of MIC is to restrict the use of engineering design languages to support the semantics of a particular domain, and to define domain-specific abstractions which can be used to integrate the models used in a particular design. These integration abstractions can be customized for particular problem domains.

- **Models of computation**

  Recent work in concurrent models of computation has led to frameworks and modeling tools for accurately representing concurrency and nondeterminism in simulation and analysis [191] [192]. The Ptolemy project created tools supporting semantic composition of different concurrent models of computation (MoCs). Safety, performance, and guarded controllers may all be most naturally defined in different MoCs. For example, the safety controller may be strictly synchronous, while the performance controller may react to asynchronous events. The MoC used to specify the behavior of the guard will have to subsume both sets of behavior, and allow discrete switching events to change modes. Formal behavior models should support all of the models of computation required to analyze or simulate the design. A compositional framework for models of computation precisely defines the behavioral interactions between models created in different MoCs.

The use of formal, executable models of computation to define the semantics of a modeling language requires tool infrastructure. Modeling language semantics are defined either directly, where the syntax of the modeling language directly represents the executable behavior model that will be simulated, or by translation, where the syntax of the modeling language is translated into another language which provides those semantics. Ptolemy (described above) and Mathworks' Simulink/Stateflow citetools:mathworks are good examples of languages with direct semantics. Frequently tool infrastructures employ model transformations to create analysis models from design models. Some good examples can be found in Whalen et al [193] and Miller et al [194], which describe the translation of different design artifacts into models for theorem provers and model checkers to support avionics software development. The MIC tool suite provides tools for defining model transformations over the languages within a domain[195].

- **Flexible design flow**

  Fig. 12 illustrates another impact of the complexity of RTA architectures, exhibited in the design flow. As runtime guards are triggered and fault information is captured, the design flow must feed back the data to designers which may refine the design to reduce failover switching or address design flaws. This feedback loop in the design process is complicated by the verification and analysis burdens of operating in a safety-critical environment.

  In order to support the rapid iteration of designs, incremental techniques show great promise. Incremental design methods have been proposed for formal deadlock analysis [196], schedulability analysis [197] [198], and other areas pertinent to safety-critical systems [199] as a means to reduce cost and schedule for design revisions. The key concept is that incremental analysis operates only on the detailed behavior models of new or changed components, while the behaviors of existing design components are abstracted to reduce the cost of re-evaluating the design. In many cases incremental analysis abstractions introduce approximations which must be carefully accounted for in order to retain the validity of the results. Thiele et al [200] and Wandeler [201] discuss abstraction validity in the context of compositional schedulability analysis.

5.2.2 Candidate Modeling and Design Tools

Finally, we discuss some of the existing modeling languages and frameworks for high-confidence design, along with comments on their suitability for addressing the problems discussed above.

- **Commercial Tools** Most popular commercially available modeling tools provide textual and graphical languages for the specification of functional models to provide simulation and controller design (e.g. Simulink and Stateflow from the Mathworks, Inc. [186]). Other tools provide built-in verification capabilities by building design environments on formally defined languages. A good example is the SCADE tool suite, which is built on the synchronous languages Esterel and Lustre. For software component and system design, numerous commercial tools build on the foundations provided by the Unified Modeling Language (UML). SysML is a UML profile directed towards support of physical controller designs as well as customization of the language for particular design domains.

- **ESMoL** The Embedded Systems Modeling Language (ESMoL) enables rapid modeling, simulation, and realization of controller software [202]. Models include software component models, hardware topology, and deployment maps instantiating the software components on the hardware. Integrated tools calculate static schedules for time-triggered networks, and generate code for platform-specific simulation (via TrueTime [203]) or for time-triggered execution on hardware (via the time-triggered FRODO virtual machine [204]). ESMoL was designed to be much simpler than AADL (though not as expressive), to provide the ability to examine the platform effects of a controller design, and to allow for experimentation with design abstractions through language and tool customization. The ESMoL framework could support the development of RTA abstractions by extending the design language and adding tools to support the required analysis of design models and generation of runtime controllers, guard code, and monitors.

- **ACME Studio** Acme Studio is a component-based architecture description language (ADL) for software designs, together with a modeling tool in which design views can be customized for particular applications or architecture styles [205]. Components can have alternative representations which meet different design or deployment criteria. Prior research results with ACME Studio have included support for parametric abstractions of design models[206] and the analysis of model consistency between different architectural views[207]. ACME is extended by writing plugins to support the application of analysis and synthesis tools to the design models.

- **AADL** The Architecture Analysis and Design Language (AADL) is a standard which supports analysis and design of complex safety-critical applications with a suite of tools for performing various analysis[208] [209] [210]. It is widely used for to specify controller implementations in the avionics industry. Extensions to AADL take the form of profile definitions with supporting tools. The overall standardization and review process for AADL, along with its size and complexity make it a less ideal target for creating research prototypes, though its wide use means that prototypes for model-based integration of RTA abstractions and tools should keep AADL in view as an ultimate delivery target for model-based RTA design and synthesis.

- **BIP** The BIP specification language is a notation and associated formal behavioral model which allows building complex software systems by coordinating the behavior of a set of atomic components. Component behavior is described as a Petri net extended with data and functions described in C code. Interactions between components model concurrency as the sets of possible event interactions between components. BIP also includes tools for automated deadlock analysis on the concurrent exchange of events represented by the specification. Gossler and Sifakis introduce the foundational concepts in [211], and Basu et al [196] present a summary and overview of this significant research effort. BIP is a concrete modeling framework, an abstract theoretical proof framework, and a formal system integration layer for distributed computations.

  From the point of view of RTA, BIP provides a formal framework for incremental analysis of the design models, as well as the ability to integrate the behavioral models directly into a deployed system. Bliudze and Sifakis describe an algebra of connectors [212], where the structure of interactions between components can be defined using an algebraic notation, and where existing interactions can be refined hierarchically by attaching components with new behaviors. The key idea is that complete system verification is not required for design extensions, but a reduced set of behaviors can be considered. Bensalem et al describe the re-engineering of legacy robotics code using BIP specifications to formally verify and and then execute the robot controllers in a deadlock-safe execution environment[213].

5.3 Software Design

The most important capability of model-based design is automated software design based on the models. Model-based design is widely used to transform design and simulation models into deployable implementation models. From the RTA perspective automated software design is essential, as both controller models and runtime guards will be derived from models, and may be complex.

5.3.1 Capabilities

- **Code generation**

  Code generation is an essential part of modern model-based controller design. Modern controller modeling and simulation tools include features to generate code for entire models and for individual components and subsystems. This is true both for commercial tools as well as tools created by the research community, as code generation is considered the bridge between abstract modeling techniques and the concrete implementations.

  Specifically for monitors (as in RTA), code generation assumes that properties are given as LTL conditions (or something very similar). Rosu and Havelund present a rewriting-based approach to generating runtime monitors from LTL specifications[214]. The LTL syntax is modified to allow the specification of finite traces. They present a handful of approaches to generating the monitors (using rewriting specifications written in Maude) in order to explore the feasibility and efficiency of the approaches. Ghezzi et al [215] attempt to infer the structure of the runtime monitor from observations. Though this approach necessarily has serious limitations, the idea of consistency checking between collected data and monitor representations could be useful for RTA applications.

  One key problem in model-based code generation is that of representing the proper model of computation for distributed environments. Commercial tools often generate standalone code for particular control functions. Any code generation provided for a distributed platform is specialized to a particular network protocol (often even to a particular network device) and to specific processors. Research in this area focuses on generating code to target a more generic model of computation so that the generated code can be reused on any platform that implements that model of computation [216] [217] [204] [202].

- **Model-Based Integration of Schedulability Analysis**

  A number of modeling tools provide automated computation of schedule feasibility and scheduling parameters. The Cheddar project covers utilization-based scheduling analysis for real-time system specifications, and has been integrated with AADL[218] [219]. Scheduling models for AADL have also been presented in Sokolsky et al [220] and Gui et al [221]. For cyclic time-triggered scheduling Zheng et al describe methods for computing schedule offsets using constraint solvers[222]. An alternative approach can be found in Schild and Wurtz[223], which has been integrated with the ESMoL tools as described in [224]. For mixed models of computation, additional research attempts to bridge the gap between scheduling models for time-triggered systems and those for event-triggered systems [225].

  For RTA designs automation is essential to support the required design flows, and incremental techniques can further strengthen the applicability of the tools to rapid design iterations. Incremental scheduling analysis techniques are discussed in Easwaran [198], where a component or subsystem-level scheduling profile is created to represent the available supply of scheduled resource (processor time, in this case) so that new real-time components can be rapidly evaluated for admission to the system design. For computing incremental schedules for distributed systems with dependencies, Pop et al [226] and Porter [199] present techniques.

- **Performance Analysis**

  Several examples of integrating performance analysis into AADL tools for aerospace systems design can be found in Bozzano et al [227, 228] and in Varona-Gomez and Villar [229]. The integration of these types of tools into the design environment supports rapid evaluation of complex designs.

5.4 Behavioral Analysis

Behavioral analysis is a key component of any system design methodology. In model-based design, behavioral analysis is performed using models of the system at various abstraction levels before the system is implemented. At each step of the design, the designer needs to ensure that the model satisfies the requirements by generating the behavior at the corresponding level of abstraction in an efficient manner. In general, techniques for analyzing the behavior of the models can be classified into (1) simulation-based methods and (2) formal verification methods. There also methods that combine ideas from both simulation and formal verification such as statistical model checking [230].

The basic principle of both simulation-based and formal verification methods is to test whether the model satisfies a set of properties that correspond to the requirements. Model-based design techniques provide an iterative progression from requirement models to implementation models by (1) model construction and refinement and (2) model verification. At each design step, models are enhanced with constructs that can describe the system behavior and analysis methods focus on the semantics at the corresponding level of abstraction.

Simulation is the predominant technique for behavioral analysis because of the availability of software tools that are well-developed and easy to use. Of course, simulation is a partial test that checks instances of a property in a particular context. Moreover, as designs become complex, simulation becomes computationally expensive especially when heterogeneous models need to be integrated. On the other hand, formal verification methods test a property under a large number of scenarios that can include many possible inputs, initial conditions, and interactions with the environment but they typically impose restrictions on the models.

5.4.1 Simulation-based methods

In RTA, simulation of control design models should include implementation models that capture how runtime properties are monitored. Such models require capturing the effects of the hardware platform on the behavior of software-based controllers. Aspects that need to be modeled include time delays, quantization effects, and platform faults. Modeling and simulation platforms such as Matlab/Simulink/Stateflow [186], Ptolemy [191], and Modelica [231] offer broad component libraries that allow designing complex systems especially at the control design layer. While such tools are essential especially at the initial design stages, they do not provide sufficient capabilities for refining the models to capture the implementation effects due to the software/hardware platform.

One approach to address these issues is *hardware/software co-simulation* [232]. A key aspect of co-simulation is the use of a high-fidelity model of the hardware platform. Simulation of the involved software components takes place within the hardware simulation, i.e. the hardware simulation is a virtual machine within which the software executes. As long as the hardware model is of high enough fidelity, co-simulation techniques are able to achieve accurate results. The hardware platform model is typically specified at a low-level of abstraction, e.g., the Register Transfer Level (RTL level), using languages such as VHDL or Verilog [233, 234]. This level of abstraction provides high resolution but is computationally expensive [232]. Moreover, co-simulation requires detailed hardware models and results in very slow simulation execution times. Co-simulation research has focused on single node, or system-on-a-chip (SoC), simulation [235]. Modeling and simulation of large-scale distributed systems using co-simulation is not yet common, perhaps due to its slow simulation time or the requirement for detailed hardware models.

Hardware/software co-simulation can also be performed also using Transaction-Level Modeling (TLM). TLM focuses on what data is being transferred rather than how it is being transmitted [236], thus a TLM communication model abstracts away communication details like channel coding to speed up simulation while retaining certain accuracy. System level design languages (SLDLs), such as SystemC, SpecC, and SystemVerilog, support TLM and are used in the system-level design process [237, 238, 239]. SLDLs can be used to build executable specification models at early design stages to simulate the system, and then refine these models for implementation. SLDLs typically support discrete-event simulation semantics that must be integrated with modeling and simulation tools for dynamic systems. SystemC/TLM is a representative SLDL which has been widely accepted for embedded system design [240]. The SystemC implementation has a discrete-event simulation kernel enabling concurrent behavior simulation. A TLM library provides a rich set of modeling primitives for specifying software components and hardware platforms for embedded systems and facilitates the generation of complex models. By adding appropriate timing annotations, a SystemC model can reveal timing behavior of the corresponding hardware/software platform.

Simulation of critical embedded systems such as avionics by generating SystemC code from an AADL specification of the hardware components has been presented in [241]. AADS, an AADL simulation and performance analysis framework based on SystemC, is presented in [242]. The tool can support prototype-based design allowing functional and non-functional (execution times, power consumption, etc.) verification of the system while it is being refined until the final implementation. A specification for SystemC-AADL interoperability is presented in [243]. Although these preliminary methods allow defining the architecture before the implementation, considerable work is required for analyzing software and hardware architectures for real time, mission-critical, embedded systems.

Another approach that allows incorporating implementation effects is *system-level performance simulation*. Model-based design tools such as Simulink and Modelica do not directly provide simulation capabilities for the deployment platform and cannot simulate the impact of deployment on controller behavior. However, extensions to Simulink such as the TrueTime toolbox [188] have been developed for this purpose. TrueTime supports modeling, simulation, and analysis of distributed real-time control systems including real-time task scheduling and execution, various types of communications networks, and suitable interfaces with the continuous-time plant model. While gaining insight into platform effects is crucial, models need to be manually developed and integrated with the control design model, which imposes an additional burden on the system designer.

Tools for synthesizing TrueTime models have been developed as part of the ESMoL tool chain [244]. Functional code is generated from an ESMoL system model and it is integrated with the TrueTime runtime code. Additionally, a new Simulink model, with the appropriate TrueTime blocks for the given hardware configuration, is automatically synthesized. This model combined with the generated code is capable of simulating platform effects introduced by the deployment of the controller model onto the hardware platform. Using such an automated synthesis approach allows the analysis of deployment flaws and schedule timing inconsistencies. Moreover, runtime properties that affect stability and safety can be identified far in advance of deploying the controller onto real hardware. Currently, the approach assumes a time-triggered architecture and time invariant controllers.

Other embedded system modeling and simulation frameworks such as SCADE/ Lustre [245], Giotto [246], Timing Definition Language (TDL) [247] do not provide include capabilities for modeling the deployment platform. Giotto and TDL, in particular, emphasize a transparent deployment approach to mapping system functionality onto the hardware platform, and thus limit the detail of the hardware models.

An alternative approach for incorporating implementation effects into simulation of control systems is the development of formalized frameworks for *simulation integration*. The High Level Architecture (HLA), for example, is a standard for simulation interoperability that allows independently developed simulations, each designed for a particular problem domain, to be combined into a larger and more complex simulation [248]. In HLA, the independent simulators are known as federates and the larger simulation formed by the interconnection of the federates is known as the federation. The HLA standard provides a set of services to accurately handle time management and data distribution among the heterogeneous simulators. HLA can be viewed as a general integration framework for simulations that can be used to integrate control design and platform deployment models. An example of an integrated modeling and simulation tool for networked control systems based on HLA which combines the network simulation capabilities of NS-2 with the control design and simulation capabilities of Matlab/Simulink has been developed in [249]. The integration of Matlab/Simulink with NS-2 allows the analysis of networked control systems using a detailed implementation of the network stack for packet level data transmission. The tool is developed following a model-based integration approach that allows for rapid synthesis of complex high-level architecture-based simulation environments [250]. The main challenge for such integration frameworks is scalability in order to handle large and complex systems.

Another integration framework for simulation interoperability is the Functional Mock-up Interface (FMI) standard [251]. The goal of FMI is that dynamic system models of different software systems can be used together for software/model/hardware-in-the-loop simulation especially for embedded software in vehicles. Integration of a model in a simulation environment can be performed either by model integration or by co-simulation at various levels. The approach allows the integration of controller code for controlling a vehicle that can include platform effects into modeling environments such as Modelica and Simulink.

5.4.2 Formal verification methods

Formal verification methods use mathematical techniques to ensure that the design satisfies properties that capture notions of functional and behavioral correctness. The main difference from the simulation-based methods is that there is no need to assume a specific input sequence and environment interactions. Formal verification methods algorithmically and exhaustively search all possible behaviors represented by the model and either prove that the properties of interest are satisfied or generate counterexamples that violate the properties.

In model-based design, formal verification methods can be applied at many different levels of abstraction. Focusing on the model-based design flow for RTA shown in Fig. 12, we categorize formal verification methods broadly into: (1) hybrid system verification and (2) model-based software verification and design. At the control design layer, hybrid system verification techniques aim at analyzing safety and reachability properties of the closed-loop system. At the software implementation layer, model-based software verification and design focuses on the formal representation, composition, and manipulation of software models during the design process. It addresses system specification, model transformation, synthesis of implementations, model analysis and validation, execution, and design evolution.

Hybrid systems theory provides the most comprehensive theoretical framework for analysis and synthesis of control systems. Continuous-state dynamics model physical systems and the interactions with the environment, as well as continuous feedback control loops. Discrete-state dynamics and event-driven models allow the abstraction of computational platforms and software systems, and facilitate the analysis and design of real-time systems. The integration of continuous and discrete dynamics has led to a fundamental understanding of the dynamics of complex embedded control systems. Hybrid system verification methods in the context of RTA are reviewed in Section 2.

Hybrid system design and verification methods are used for synthesizing the control dynamics to satisfy safety, stability, and performance requirements. The control models must be implemented by software and deployed in the platform. Model-Integrated Computing (MIC) provides the technology foundation for model-based software design [185] and has been used extensively in high-confidence applications. Model-based design utilizes transparency and simplicity for synthesizing software implementations that allow reasoning and verification. Transparency requires the explicit representation of layers of abstractions, their semantics, and their relationship along all stages of the design flow and system lifecycle. These representations need to be formal and manipulatable for design automation and expressive enough for inspection and inference. Simplicity is another cornerstone of high-confidence system design. Since exhaustive testing of real-world systems to reduce the probability of errors to acceptable levels is impossible, layers of abstractions in the design flow captured by architecture specifications need to be used in order to preclude classes of design errors. This approach radically simplifies the integration, test and certification process by emphasizing correctness via construction instead of analysis and testing. Examples of composition platforms which provide support for containment of design errors are the time-triggered architecture (TTA) [189] and the ARINC 653 standard.

An interesting approach for achieving compositionality for selected properties is to introduce restrictions in the system design. BIP (Behavior Interaction Priority), for example, is a formal framework for modeling heterogeneous real-time components [196]. The BIP model achieves compositionality in state invariants (such as deadlocks) by introducing a specific interaction and scheduling technique. BIP sacrifices some runtime performance for improving compositionality but in safety critical systems this may be acceptable. Investigating new compositional frameworks for multiple interacting properties and understanding the performance tradeoffs are important research goals.

Because BIP focuses on the organization of computation between components, it can be viewed as an architecture description language (ADL) [196] that provides a rigorous system modeling framework that is suitable for verification. A method for translating AADL into BIP for verification of real-time systems has been presented in [252, 253].

Model-based design provides processes and tools for systematic code synthesis based on model transformations as described in Section 5.3.1. Once models are verified, automatic code generation produces executable code from the models. Recent work in formal verification addresses the significant challenge of proving the equivalence between behaviors specified in a design language such as Simulink and the behavior of code generated from those Simulink models. A framework for making such comparisons, where particular model transformations are used to perform code generation are verified as instances is described in [254]. Proofs may be constructed for the realization of that transformation on a particular source and destination model.

5.5 Dependability Analysis

Achieving dependability in safety critical systems is of extreme importance since failures could have severe negative impacts. Dependability is the ability to deliver service that can justifiably be trusted, and it integrates attributes such as reliability, availability, safety, security, and maintainability [255]. There exist several means to achieve dependability, which includes fault detection, isolation, mitigation, and fault tolerance.

Contrary to its importance, fault management for the high-confidence systems such as UAVs is not a solved problem. The need for autonomous operations in unknown and difficult environments makes this problem even more acute. Aircraft, for example, incorporate subsystems (called Line Replaceable Units (LRUs) in the aerospace industry) that typically include physical devices such as pumps, valves, and pipes, and one (or more) embedded computing systems. With use over time these units may develop faults and degrade in performance. Subsystem manufacturers often provide monitoring algorithms to detect these faults, and report them to a centralized computer that runs a supervisory controller. The controller may run a fault management scheme to deal with the faults so that safe operation may be maintained. For example, the Central Maintenance Computer (CMC) of the Boeing 777 aircraft computer logs fault codes reported by the LRUs for later inspection by the maintenance crew. For faults with more immediate consequences, corrective actions must be taken. In present day systems, typically fault management and remedial tasks are left to humans (e.g., pilots). However, alarms generated by faults in complex systems often cause information overload. Further, autonomous vehicles require sophisticated online automated mechanisms that are provably robust and reliable in fault detection, isolation, and recovery tasks to ensure safe operation.

Dependability is a requirement that can be best addressed by model-based design. Fault and health management architectures can be developed and integrated into the system using model-based design techniques. RTA introduces considerable challenges because of the strong dependence on runtime properties. The runtime environment is much more complex in RTA than traditional systems, and therefore, there is a need to design the runtime components so that they are robust to faults and degradations. In the following, we focus on how model-based design has been applied to (1) hybrid system fault management and (2) software health management, two complementary aspects that are important for RTA.

5.5.1 Fault Management in Hybrid Control Systems

The Fault-Adaptive Control Technology (FACT) tool suite has been developed in order to address the challenges of fault management of such systems [256, 257]. FACT is a systematic model-based approach to the design and implementation of control systems that can accommodate faults. The FACT tool suite uses a model-integrated computing approach to automatically synthesize simulation models, hybrid observers, and diagnoser software-code from hierarchical component based system models. Figure 13 illustrates the fault adaptive control architecture. Capabilities include fault detection in the presence of nonlinear dynamics and noisy sensors, rapid fault isolation, estimation of the fault magnitude, and control reconfiguration to accommodate the fault. Reconfiguration is performed online which means that set points and control parameters may have to be changed, or a different controller may have to be selected to continue system operation. The FACT tool suite and various extensions have been applied to many practical systems that include robotic vehicles [258, 259], spacecraft power distributions [260], and advanced life support systems [261].

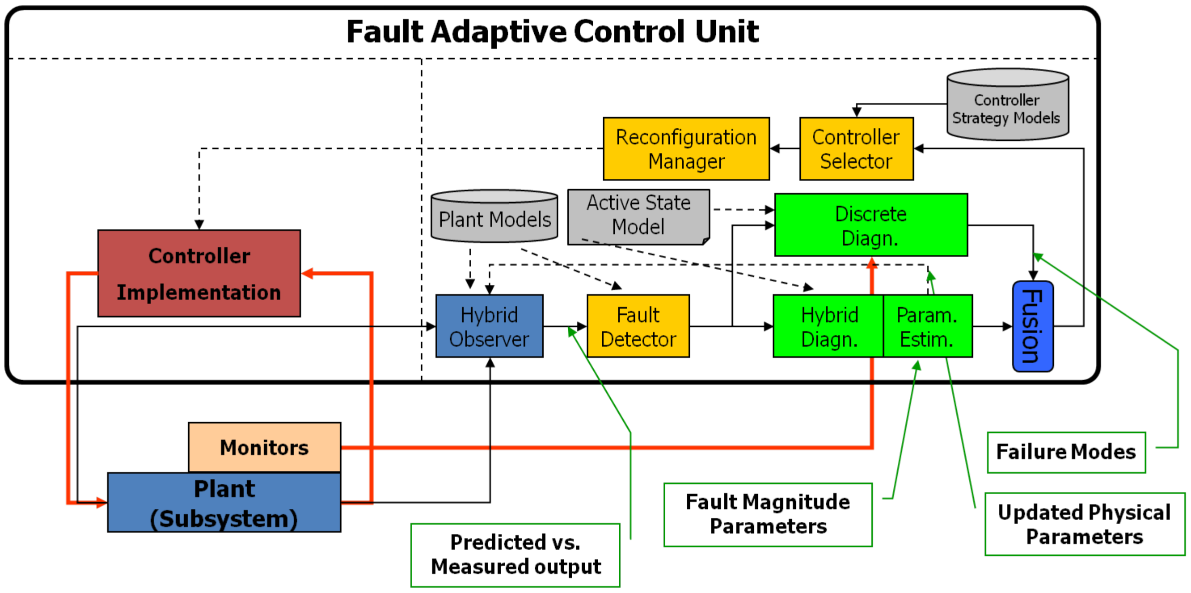


Figure 13: FACT Runtime System Architecture.

5.5.2 Software Health Management

Architectures such as FACT provide capabilities that are useful for RTA such as state estimation, monitoring, and control reconfiguration. Because of the interactions between the control design layer and the implementation layer, it is necessary to integrate hybrid system fault management techniques with health management of the software that implements the required functionality.

A promising approach for dependability analysis of embedded software is to annotate architecture models with dependability-related information such as faults, fault propagation, and policies for fault tolerance (e.g., replication and voting). The Error Model Annex standard, for example, can be used in conjunction with AADL and the resulting models can then be used for dependability analysis [210]. Dependability analysis methods include fault tree analysis, failure modes and effects analysis, and other model-driven approaches. A comparative study of methodologies using AADL's error annex can be found in [262]. Although such methods allow various dependability analyses of the system, they must be complemented with runtime fault management systems.

An approach to Software Health Management (SHM) that uses concepts from the field of Systems Health Management such as detection, diagnosis and mitigation similarly to the hybrid system fault management techniques presented above has been developed in [263, 264]. Conventional Systems Health Management is associated with the physical elements of the system, and includes anomaly detection, fault source identification (diagnosis), fault effect mitigation (at runtime/ online during operation), maintenance (offine), and fault prognostics (online or offine). Software Health Management borrows concepts and techniques from Systems Health Management and is a systematic extension of classical software fault tolerance techniques. SHM is performed at runtime, and just like Systems Health Management it includes detection, isolation, and mitigation to remove fault effects. SHM can be considered as a dynamic fault removal technique. An example of an Inertial Measurement Unit subsystem is presented in [265].

5.6 Model-based testing

Research and tool development in the Electronic Design Automation (EDA) industry has led to the use of test bench models to integrate functional verification and platform-specific simulation. These tools support the comparison of analyses and simulations between different design stages (for example, integrating the guard design and modifying it to account for implementation effects) and between design refinements as evaluation uncovers behaviors that are out of bounds. Test bench models can also coordinate the evaluation and comparison of design models that exhibit nominal behavior and faulty behavior. Automation of these comparisons is key to streamline the development flow to accommodate the additional complexity of the RTA infrastructure.

Dynamic systems and their controller components are represented using various design models. The development flow of of a full control system design can proceed incrementally, with subsystems behavior developed and evaluated in independent test bench models. In the world of EDA electronic sub-modules for a chip design are specified independently along with elements which provide the necessary inputs to simulate the behaviors of interconnected components. Throughout this section we include a number of representative citations from the extensive EDA literature to illustrate the facets of test bench-based design and analysis. Test bench models are key for defining evaluations for software-implemented controllers and the platforms on which they are deployed to ensure that requirements are met.

The capability to integrate simulation-based evaluation and verification in a single design model is valuable when considering the limitations of each approach. Formal verification is exhaustive, but limited by the complexity of the behaviors represented in the design model. Simulation-based evaluation (and hardware testing) function well for complex models, but do not provide the level of assurance available from formal verification. Realistically, both techniques will be required. The modeling environment must ensure that behaviors represented in the design have the same meanings in both contexts. Unifying the collection and representation of evaluation results can also streamline the design flow and reduce development time.

5.6.1 Test Automation

Test bench modeling allows developers to define test and evaluation models that reference system and subsystem behavior models, and define an environment which supports execution of both simulations and target systems, and both testing and verification[266]. Test bench models solve the problem of keeping different behavioral representations (simulations, analysis models, and target systems) together so their results can be easily compared for consistency. Automation allows test bench models to be updated as the design changes[267].

For embedded software design and evaluation, Hause et al describe the use of models to decouple the test definitions from the specification of the system under test using UML and SysML models[268]. They introduce the notion of a test bench model which configures the hardware and software involved in a particular test. Lattmann et al use an extended notion of test bench, where the test bench model captures the inputs, outputs, performance, metrics, and requirements for each test [269].

Model-based test automation does not change directly for RTA, rather it complements the design flow requirements for evaluating RTA models at both design time and run time. The system under test takes one of three different forms – 1) a legacy controller (presumably with test models already defined); 2) a performance controller - this must be evaluated with as many of the existing test bench models as possible, though the results may be interpreted differently, as exhaustive testing and verification are not possible. Additional test models should be added to establish the performance bounds of the controller where possible; and 3) a guarded performance controller -

test/verification models for both the performance controller and the legacy (safety/recovery) controllers should be applied to the guarded performance controller. If the guard is operating correctly, then nearly all the tests for both controllers should pass. Additional test models should be defined to exercise the switching conditions of the guard using extreme trajectories and injected faults.

5.6.2 Test Vector Generation

The EDA community uses automated test generation at a number of levels. Lavagno et al give a survey of automated test pattern generation (ATPG) for sequential and combinational circuits [270]. As software is much more flexible than hardware, automated testing tools and program infrastructure have many facets in the software domain. Godefroid et al give a survey of recent techniques in the automated testing of software [271].

The set of possible tests is virtually infinite (i.e. larger than any reasonable evaluation effort can support) for the input, output, and state space of any realistic engineering system. Current research investigates efficient mechanisms for exhaustive evaluation, and the generation (and reduction) of test sets from system design models. Software tests are generated from behavioral specifications - for example, some methods use formal automata with temporal logic properties (as in Micskei and Majzik [272]), where others start from UML activity diagrams with OCL constraints (as in Yin and Liu [273]). One of the most promising techniques in this area is presented by Tan, Kim, and Lee [274]. Instead of generating lengthy test traces which can only be evaluated offline, their approach generates test input sequences from hybrid automata specifications and encodes the sequences as automata that can be implemented to test the system online, even in environments where memory space is limited.

In an RTA context test vector generation concerns illustrate some overall limitations of the RTA approach. The complexity of the performance controller will likely lead to an intractable space of test cases. If the guard conditions used at runtime require state-tracking, or if the guard conditions are too complex then testing and verification may not be feasible. The RTA also places some challenges on the generation of guards from specifications. If automated test generation is important, as it will be when considering the implementation of the controllers and guard as software and their deployment to (possibly distributed) computing hardware, then the test generation methods may impose constraints on the size and form of the guard implementation, as adequate testing is not optional for high-assurance applications.

5.6.3 Test Space Reduction

Numerous techniques attempt to reduce the size of generated test vector sets, while maintaining requirements coverage and fault detection. Lin and Huang describe one such approach [275]. They describe a test set minimization technique that maintains design coverage. As ties occur in an objective function, they choose the test case that provides better fault detection. Many prior results only aim to find the smallest test set that preserves coverage, compromising the ability to detect faults in some cases.

5.6.4 Test Interpretation

Test bench models can help a designer develop intuition and understanding about the behavior of a component or subsystem. Below we list some of the possible use cases and interpretations of test models which are relevant to RTA.

1. Validating runtime data against simulation – Bogosyan et al present an interesting way to use simple controllers in the simulation environment to integrate simulated component dynamics for a hybrid electric vehicle with data measured from the actual vehicle [276].
2. Parameter space exploration – One interesting idea (from Donze et al [277]) uses guided simulation (i.e. tests and/or state exploration) to partition the system design model parameter sets into partitions based on whether each configuration leads only to safe behavior, could lead to unsafe behavior, or whether safety cannot be determined for a particular partition.
3. Exhaustive property evaluation (safety/performance bounds)
4. Trajectory-based performance assessment for simulations and target systems – Henftling et al [278] and Huang et al [279] describe the use of a testbench model to integrate both abstract (verifiable) behavior representations and models which can be executed either by an emulator or by the target hardware.
5. Debugging implementation models – test bench models and tools should support fault injection and isolation in order to assist in the debugging of design models. This is the subject of recent work in the EDA research community [280] [281].
6. Defining regression models for prior defects to ensure quality under design revisions.

5.7 Applications and Evaluation

The research literature provides a number of interesting examples which illustrate different facets of model-based design for complex and safety-critical systems. They are multi-faceted in that they each cover multiple aspects of the modeling and design process, and they are generally detailed. Many of these could be adapted as test cases to evaluate and extend the use of model-based design for RTA techniques.

5.7.1 Example: Modeling of the simplex architecture

Sha et al claim to be able to achieve highly reliable designs using simple, reusable, robust, and formally verified architecture patterns [282]. The simplex architecture is just an example of a more general pattern that minimizes the size and complexity of the dependency tree in a system design. They present an example of a high-level architecture suitable for RTA, and describe the integration of tools to support it, based on AADL models.

5.7.2 Example: Health Management of an IMU

Dubey et al describe an avionics application where a model-based health management architecture was used to mitigate faults in a redundant sensor architecture [265]. The authors provide significant detail regarding the design and fault models, including a detailed discussion of the distributed execution model and detailed fault behavior.

5.7.3 Example: Modeling and analysis of a network protocol

Pike et al [283] give a case study emulating a physical-layer network protocol, including formal parameterized models of the physical link (signal levels and timing jitter), asynchrony of the communication endpoints, real-time constraints, random parameter-space exploration, and statistical evaluation.

5.7.4 Example: NASA Planetary Rover

X9 is a test-case generation and monitoring environment based on rewriting techniques[284]. X9 uses a model checker inline, and is capable of determining whether mission plans are being executed in the proper order, and also whether fault notices are propagated correctly. Each mission plan contains a set of temporal logic formulae that must be satisfied during the execution of the plan.

5.8 Challenges

Model-based design provides a powerful framework for the development of safety-critical systems because behavioral analysis methods can be integrated in the design process based on formal models at different levels of abstraction. The main challenge in RTA lies on the integration of heterogeneous models that allow system evaluation of the control design while incorporating runtime properties that depend on the software and hardware platform. To address this challenge, it is necessary to define the system architecture of the system before its implementation and analyze the constraints imposed on the system by the architecture from the beginning of the design cycle until the final implementation.

5.8.1 Modeling Languages and Frameworks

The RTA approach increases the fundamental complexity of an engineering design. Each RTA subsystem has two controllers, each of which may operate according to different assumptions and execution models. Further, the guard logic which integrates the two controllers must be modeled and evaluated. Modeling languages and tools must support the specification of the functions and behavior of these elements, and support abstractions which assist the verification and testing efforts in specializing those evaluations to the various execution scenarios (e.g. high-performance operation, safe operation, switching).

Updating existing design models based on data from run time characteristics is key to overcoming the additional design complexity introduced by the RTA architecture. Unless the overall time and cost of re-verification (and re-certification) of design changes can be reduced, the RTA approach may have limited applicability. Modeling languages and tools must support the creation of appropriate abstractions to be fed back along the design flow. Exploration of the applicability of incremental design techniques is a key challenge area that promises to address some of these issues. Beyond exploring applicability, extending modeling languages and design tools to support incremental design is also an open problem area.

5.8.2 Software Design

Generation of assured software code for runtime components (such as active guard logic) is still an open area of research. Software model checking has made significant progress in recent years, but many problems remain open. Model-based techniques integrating assured code generation and software model checking, and relating the software models and results to those of the system design models will provide many useful research problems.

For schedulability analysis, the performance controller may not have deterministic timing, so the guard logic may be forced to monitor execution time. The maximum allowable execution time of the controller would be constrained by the accuracy of trajectory prediction in the guard, and the dynamics of the safety/recovery controller. Accurately modeling the relationship between runtime bounds in the guard and the potential performance and timing of the controller is a significant challenge. One example of variable timing is an adaptive controller that sporadically recomputes its dynamic model when conditions change. Another is a heuristic controller derived from learning techniques that is either learning online or for which the calculation complexity is non-uniform in different state regions.

Current research considers compositional and incremental analysis of design models to establish correctness properties. End-to-end analyses complicate the application of compositional or incremental design methods. Many correctness properties which must be supported in a design (e.g. end-to-end latency) depend on a sequence of components together satisfying a design constraint. These end-to-end properties are further complicated by indirect coupling with other components that are specified independently, but which influence each other by sharing common resources.

5.8.3 Simulation-based Methods

Simulation is currently the most widely used method to verify system models. For RTA, it will be necessary to develop methods that incorporate implementation effects into the simulation of dynamic systems in order to analyze runtime properties of the system design. Although several existing approaches have focused on related problems, the strong interactions between runtime properties and system verification in RTA requires (1) new model-based methods that are capable of refining control models with implementation details and (2) efficient and scalable simulation techniques for heterogeneous systems.

RTA will require the integration of a hardware/software co-simulation framework with modeling and simulation platforms for dynamic systems such as Simulink and Modelica. Existing methods for performance evaluation of embedded systems lack the necessary precision or are computationally inefficient. Significant challenges for such integration include the availability of tools that allow rapid design and modeling of embedded control systems of various complexity and size and the runtime scalability of the simulation environment.

A promising direction for simulation-based methods required by RTA is the development of simulation integration frameworks that enable flexible and extensible support for multiple models of computation at different levels of abstraction. The frameworks should be customizable and allow the integration of both existing and future design tools.

5.8.4 Hybrid System Verification

The use of formal verification methods in model-based design faces significant challenges, especially in the case of RTA. Although hybrid verification methods provide the mathematical foundations for analyzing interacting physical and computational processes, thus far they have been used only on relatively simple models combining finite state machines and linear differential equations. For applicability to the design of high-confidence, distributed aerospace systems, hybrid system theory needs to be extended to far richer classes of models involving different hierarchies of heterogeneous representations of computation and concurrency, resource awareness, adaptability, quality of service (QoS), and system complexity.

Typically, models for hybrid control systems capture design intent and are not suitable for verification. For example, Simulink/Stateflow models are created for simulation and are typically deterministic either because the designer or the language has resolved any nondeterminism. Hybrid system formal verification methods typically assume that the models are nondeterministic. In general, there is a significant conceptual gap between a design language and a verification language that needs to be addressed. Therefore, there is a need for the transformation of design models into verification languages including the translation of properties that need to be verified.

Previous work has investigated the transformation of Simulink/Stateflow models to formal models but imposes restrictions on the components that can be used for control design [285, 286]. Moreover, for such approaches to be practical, the transformation itself should be verified otherwise the verification results could be meaningless.

In order to address some of these limitations, RTA provides an alternative to design-time verification methods. Complex controllers that cannot be verified at design-time will be integrated with verifiable safe controllers using appropriate switching policies. Therefore, RTA must rely on design-time formal verification techniques for the implementation of the safe controllers. Moreover, the safe controllers will interact with other components, e.g., unverifiable controllers, monitors, and switching policies. The design framework should support the integration of design-time and runtime verification methods.

Hybrid verification in RTA must also address the dynamic runtime properties of both the controlled plants (e.g., groups of UAVs with changing control objectives and system configurations) and the control platforms (with changing communication and computation resources) which cannot be modeled completely at design-time. Hybrid system models need to be extended with hierarchical control layers providing different levels of assurance for behavioral properties. New online verification methods need to be developed and design-time verification methods need to be migrated to operation-time to address adaptivity and provide increased robustness. Model-based design must provide a framework for integration of design-time and runtime verification techniques that cover the system requirements.

Scalability of existing hybrid verification methods is another significant challenge. The complexity of a given method is not only determined by its accuracy (and the properties that can be addressed) but mainly by the sheer size of the model analyzed – which can be measured in the number of components, tasks, variables, etc... In order to develop methods that scale and use them in model-based design that addresses the needs of real defense systems, compositionality is paramount. Compositionality would allow properties to be inferred only by consideration of the properties of components and their interactions.

5.8.5 Software Verification

Model-based design employs systematic code synthesis based on model transformations. This step also requires that the model transformation be verified and that it generate code that is ensured to be correct. The verification of model transformations is possibly as difficult as the verification of compilers. Compiler verification is an active research topic and it is too difficult in the general case. However, in model transformation there are two important differences from general compiler verification: (1) the model transformations are explicitly modeled using a well-defined modeling language and (2) the model transformations are domain-specific. Based on these features, the verification problem of model transformation tools can be addressed using certifiable model transformers. The approach is based on automatic program synthesis tools and it can be used to (1) automatically produce verification conditions in the generator (in addition to the normal output, i.e. code); (2) generate a proof of correctness of the code from a theorem prover or a proof checker, and (3) attach the proof to the code as a “certificate” for the correctness.

While extensive and systematic use of model-based methods in systems and software development and the application of safe composition platforms are expected to eliminate design flaws and spurious behavior, high-confidence systems must be prepared for the unexpected caused by limited modeling fidelity, and unmodeled environmental impacts, or unforeseen situations. The challenge is to develop verification methods that can be used at runtime in order to ensure the conformance of system behavior to a set of system models that are simple and nonrestrictive but sufficient for indicating the violation of safety properties.

5.8.6 Dependability Analysis

Model-based fault diagnosis schemes are the preferred approach to health management of hybrid control systems because they provide general and robust diagnosis solutions [287]. However, deployment of these schemes on real systems presents significant challenges in model development, system monitoring, and fault detection and isolation. Such a framework requires accurate and reliable models of the physical dynamics that for a realistic vehicle encompass multiple domains (e.g., hydraulic, electrical, and mechanical). Behaviors can be nonlinear, and the interactions between components and between the system and the environment can be difficult to capture. Monitoring complex systems to detect abnormal behavior also presents a number of challenges. A model of the system is used to predict nominal behavior, and deviations between observed and predicted behaviors signal the presence of faults. However, system monitoring is often performed with incomplete information due to lack of sensors, or with sensors that only provide data at rates slower than what is required to accurately estimate the system state. In addition, uncertainty in both the measurements and the system model may degrade the estimation accuracy. In spite of these difficulties, fault detection must be robust to minimize false alarms, missed detections, and detection delays. Challenges also arise in the fault isolation task. Different types of faults (abrupt, incipient, and discrete) can be manifest in system components, sensors, and actuators. Interactions among components may make it hard to distinguish between faults. Furthermore, fault isolation is impacted by the granularity of the model and the measurements that are available to the diagnosis system. Even with these issues, diagnosis algorithms must provide robust, accurate, and precise results in a timely manner. Computational issues arise in accomplishing this goal, particularly with hybrid nonlinear systems.

While traditional design-time and off-line approaches to testing and verification contribute significantly to improving and ensuring high-dependability software, they do not cover all possible fault scenarios that a system could encounter at runtime. Thus, runtime health management of complex embedded software systems is needed to improve their dependability. Self-adaptive systems must be able to adapt to faults in software as well as the hardware (physical equipment) elements of a system, even if they appear simultaneously.

RTA intensifies the challenges related to runtime support for fault and health management. Faults and errors may occur either in the physical or in the cyber components of system. Because of the required runtime infrastructure and the complex interactions, it is increasingly difficult to design active fault management systems that can reliably respond at runtime to a multitude of possible and even unforeseen fault conditions.

5.8.7 Model-based Testing

As complexity of control system designs grow, model analysis must ensure that the specified tests cover the behaviors specified in the design. This is a key challenge for automated testing, both in terms of specifying models and in model generation. Development of test bench modeling concepts for definition of tests, automation of all kinds of design-time and runtime evaluation, and interpretation of results is a key enabling technology for RTA.

In the RTA context we have three sets of tests/analyses that must be performed for each controller - legacy, performance, and guarded. This increases the test time burden. We usually assume that safety and performance analysis have already been performed for the legacy controller, and that many of the safety analyses will not apply for the performance controller. The guarded controller incurs a higher test burden than either of the other controllers - not only must it pass the safety and performance analyses, the guard logic must also be exercised against faults which are particular to the logic itself (and to its implementation). Automatic reduction of the size of the test space, while prioritizing tests and ensuring that critical fault cases are not missed is an open problem. This must be supported by the modeling environment from which the design evaluation is driven.

Incremental analysis provides one mechanism to improve development time. In the RTA context, any incremental methods must also support safety certification of the design models. Beyond simple streamlining of a development process, incremental model-based testing approaches will have to formally and clearly demonstrate the relationship between the designed behaviors before and after the changes are made. This demonstration will need to be made in the context of the RTA infrastructure. Use of incremental techniques to support re-certification of design changes is still an open area of research.

References

[1] Lael Rudd and Herb Hecht. Certification techniques for advanced flight critical systems. Technical report, WPAFB, 2008.

[2] J. Markoff. Google cars drive themselves, in traffic. *The New York Times*, 10:A1, 2010.

[3] Satish Balantrapu. Role of artificial neural networks in microgrid, 2010.

[4] NITRD. Development needs report by nitrd. high-confidence medical devices: Cyber-physical systems for 21st century health care.

[5] R. Horowitz and P. Varaiya. Design of an automated highway system. *Proceedings of the IEEE*. This issue.

[6] J. Lygeros, D.N. Godbole, and S. Sastry. Verified hybrid controllers for automated vehicles. *IEEE Transactions on Automatic Control*, 43(4):522–539, April 1998.

[7] P. Varaiya. Smart cars on smart roads: problems of control. *IEEE Transactions on Automatic Control*, 38(2):195–207, 1993.

[8] C. Livadas, J. Lygeros, and N. Lynch. High-level modeling and analysis of tcas. *Proceedings of the IEEE*. This issue.

[9] J. Lygeros, G. J. Pappas, and S. Sastry. An approach to the verification of the Center-TRACON Automation System. In T. Henzinger and S. Sastry, editors, *Hybrid Systems : Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 289–304. Springer Verlag, Berlin, 1998.

[10] C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management : A study in muti-agent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, April 1998.

[11] A. Balluchi, L. Benvenuti, M. DiBenedetto, C. Pinello, and A. Sangiovanni-Vincentelli. Hybrid control in automotive applications. *Proceedings of the IEEE*. This issue.

[12] A. Ohata, E. Gassenfeit, K. Butts, and B. Krogh. Automotive engine systems: A need for hybrid systems simulation and analysis. *Proceedings of the IEEE*. This issue.

[13] D. Pepyne and C. Cassandras. Hybrid systems in manufacturing. *Proceedings of the IEEE*. This issue.

[14] S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. Simulation, analysis and optimization of continuous-discrete interactions in chemical processing plants. *Proceedings of the IEEE*. This issue.

[15] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in CHARON. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems : Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[16] M. Song, T-J. Tarn, and N. Xi. Intelligent scheduling, planning and control: Analytical integration of hybrid systems. *Proceedings of the IEEE*. This issue.

[17] O. Maler and S. Yovine. Hardware timing verification using KRONOS. In *Proceedings of 7th Conference on Computer-based Systems and Software Engineering*, 1996.

[18] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[19] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[20] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.H. Ho. Hybrid automata : An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[21] X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid automata. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer-Verlag, 1993.

[22] A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In *Computer Aided Verification*, pages 95–104, 1994.

[23] G. Lafferriere, G. J. Pappas, and S. Sastry. O-minimal hybrid systems. *Mathematics of Control, Signals, and Systems*, 13(1):1–21, 2000.

[24] G. Lafferriere, G. J. Pappas, and S. Yovine. A new class of decidable hybrid systems. In *Hybrid Systems : Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 137–151. Springer Verlag, 1999.

[25] G. Lafferriere, G. J. Pappas, and S. Yovine. Reachability computation for linear hybrid systems. In *Proceedings of the 14th IFAC World Congress*, volume E, pages 7–12, Beijing, P.R. China, July 1999.

[26] T.A. Henzinger. Hybrid automata with finite bisimulations. In *ICALP: Automata, Languages, and Programming*, Lecture Notes in Computer Science 944, pages 324–335. Springer, 1995.

[27] T. Henzinger, P. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):540–554, April 1998.

[28] Rajeev Alur, Thao Dang, and Franjo Ivancic. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embedded Comput. Syst.*, 5(1):152–199, 2006.

[29] Rajeev Alur, Thao Dang, and Franjo Ivancic. Counterexample-guided predicate abstraction of hybrid systems. *Theor. Comput. Sci.*, 354(2):250–271, 2006.

[30] R. Kumar, C. Zhou, and S. Jiang. Safety and transition-structure preserving abstraction of hybrid systems with inputs/outputs. In *2008 Workshop on Discrete Event Systems*, pages 206–211, Goteborg, Sweden, 2008.

[31] C. Zhou and R. Kumar. Finite bisimulation of reactive untimed infinite state systems modeled as automata with variables. *IEEE Transactions on Automation Science and Engineering*. Accepted (2011).

[32] C. Zhou and R. Kumar. Modeling simulink diagrams using input/output extended finite automata. In *2008 IEEE International Computer Systems and Applications Conference*, pages 462–467, Seattle, WA, July 2009.

[33] M. Li and R. Kumar. Stateflow to extended finite automata translation. In *2011 IEEE International Computer Systems and Applications Conference*, Munich, 2011.

[34] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.

[35] S. Jiang. Reachability analysis of linear hybrid automata by using counterexample fragment based abstraction refinement. In *Proceedings of the 26th American Control Conference*, pages 4172–4177, New York, NY, July 2007.

[36] S. Jiang and R. Kumar. Prevention of sequential message loss in can systems. In *2011 IEEE International Computer Systems and Applications Conference*, pages 479–484, Seattle, WA, July 2009.

[37] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

[38] P. Tabuada. *Verification and Control of Hybrid Systems - A Symbolic Approach*. Springer, 2009.

[39] A. Girard, G. Pola, and P. Tabuada. Approximately bisimilar symbolic models for incrementally stable switched systems. *IEEE Transactions on Automatic Control*, 55(1):116–126, 2010.

[40] D. Angeli. A Lyapunov approach to incremental stability properties. *IEEE Transactions on Automatic Control*, 47(3):410–421, 2002.

[41] A. Girard. Synthesis using approximately bisimilar abstractions: state-feedback controllers for safety specifications. In *Hybrid Systems: Computation and Control*, pages 111–120, 2010.

[42] Antoine Girard. Synthesis using approximately bisimilar abstractions: Time-optimal control problems. In *IEEE Conference on Decision and Control*, 2010.

[43] G. Pola, A. Girard, and P. Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(10):2508–2516, 2008.

[44] G. Pola, P. Pepe, M. D. di Benedetto, and P. Tabuada. Symbolic models for nonlinear time-delay systems using approximate bisimulations. *Systems and Control Letters*, 59:365–373, 2010.

[45] M. Zamani, G. Pola, M. Mazo, and P. Tabuada. Symbolic models for nonlinear control systems without stability assumptions. 2010. arXiv:1002.0822, submitted for publication.

[46] P. Roy, P. Tabuada, and R. Majumdar. Pessoa 2.0: A controller synthesis tool for cyber-physical systems. In *Hybrid Systems: Computation and Control*, 2011.

[47] Y. Tazaki and J. Imura. Discrete-state abstractions of nonlinear systems using multi-resolution quantizer. In *Hybrid Systems: Computation and Control*, volume 5469 of *LNCS*, pages 351–365. Springer, 2009.

[48] J. Camara, A. Girard, and G. Goessler. Synthesis of switching controllers using approximately bisimilar multiscale abstractions. In *Hybrid Systems: Computation and Control*, 2011.

[49] A. Girard. Approximately bisimilar finite abstractions of stable linear systems. In *Hybrid Systems: Computation and Control*, volume 4416 of *LNCS*, pages 231–244. Springer, 2007.

[50] C. Zhou and R. Kumar. Semantic translation of simulink diagrams to input/output extended finite automata. *Journal of Discrete Event Dynamical Systems*, Dec. 2010. DOI 10.1007/s10626-010-0096-1.

[51] W. Kühn. Rigorously computed orbits of dynamical systems without the wrapping effect. *Computing*, 61:47–67, September 1998.

[52] Antoine Girard, Colas Le Guernic, and Oded Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In João P. Hespanha and Ashish Tiwari, editors, *HSCC*, volume 3927 of *LNCS*, pages 257–271. Springer, 2006.

[53] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *LNCS*, pages 291–305. Springer, 2005.

[54] M. Althoff, B. H. Krogh, and O. Stursberg. Analyzing reachability of linear dynamic systems with parametric uncertainties. In A. Rauh and E. Auer, editors, *Modeling, Design, and Simulation of Systems with Uncertainties*. Springer, 2011.

[55] M. Althoff, O. Stursberg, and M. Buss. Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization. In *Proc. of the 47th IEEE Conference on Decision and Control*, pages 4042–4048, 2008.

[56] M. Althoff, O. Stursberg, and M. Buss. Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes. *Nonlinear Analysis: Hybrid Systems*, 4(2):233–249, 2010.

[57] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 540–554. Springer, 2009.

[58] Ian M. Mitchell, Alexandre M. Bayen, and Claire J. Tomlin. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE T. Autom. Contr.*, 50(7):947–957, 2005.

[59] Ian M. Mitchell and Jeremy A. Templeton. A toolbox of Hamilton-Jacobi solvers for analysis of nondeterministic continuous and hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, number 3414 in LNCS, pages 480–494. Springer Verlag, 2005.

[60] L. Jin, R. Kumar, and N. Elia. Reachability analysis based transient stability design in power systems. *International Journal of Electrical Power and Energy Systems*, 32:782–787, 2010.

[61] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In Rajeev Alur and George J. Pappas, editors, *HSCC*, volume 2993 of *LNCS*, pages 477–492. Springer, 2004.

[62] Stephen Prajna, Ali Jadbabaie, and George J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE T. Automat. Contr.*, 52(8):1415–1429, 2007.

[63] S. Prajna and A. Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *Hybrid Systems: Computation and Control, LNCS 2993*, pages 477–492. Springer-Verlag, Heidelberg, 2004.

[64] S. Prajna, A. Papachristodoulou, P. Seiler, and P. A. Parrilo. SOSTOOLS and its control applications. In A. Garulli and D. Henrion, editors, *Positive Polynomials in Control*. Springer-Verlag, 2005. To appear.

[65] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[66] R. Alur. Formal verification of hybrid systems. In *11th International Conference on Embedded Software*, 2011.

[67] G. E. Fainekos, A. Girard, and G. J. Pappas. Temporal logic verification using simulation. In *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *LNCS*, pages 171–186. Springer, 2006.

[68] A. Girard and G. J. Pappas. Verification using simulation. In *Hybrid Systems: Computation and Control*, volume 3927 of *LNCS*, pages 272–286. Springer, 2006.

[69] A. Girard and G. Zheng. Verification of safety and liveness properties of metric transition systems. *ACM Transactions on Embedded Computing Systems*, 2011. To appear.

[70] A. A. Julius, G. Fainekos, M. Anand, I. Lee, and G. J. Pappas. Robust test generation and coverage for hybrid systems. In *Hybrid Systems: Computation and Control*, volume 4416 of *LNCS*, pages 329–342. Springer, 2007.

[71] F. Lerda, J. Kapinski, E. M. Clarke, and B. H. Krogh. Verification of supervisory control software using state proximity and merging. In *Hybrid Systems: Computation and Control*, volume 4981 of *LNCS*, pages 344–357. Springer, 2008.

[72] A. A. Julius and S. Afshari. Using computer games for hybrid systems controller synthesis. In *IEEE Conference on Decision and Control*, 2010.

[73] Goran Frehse and Rajarshi Ray. Design principles for an extendable verification tool for hybrid systems. In *ADHS*, 2009.

[74] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, LNCS. Springer, 2011. To appear.

[75] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In *CAV*, 2009.

[76] Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *Trans. on Embedded Computing Sys.*, 6(1):8, 2007.

[77] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008.

[78] André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, 2010. Advance Access published on November 18, 2008.

[79] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

[80] André Platzer. The complete proof theory of hybrid systems. In *LICS*. IEEE Computer Society, 2012.

[81] André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[82] André Platzer. Differential dynamic logic for verifying parametric hybrid systems. In Nicola Olivetti, editor, *TABLEAUX*, volume 4548 of *LNCS*, pages 216–232. Springer, 2007.

[83] D. G. Ward, J. D. Schierman, B. C. Dutoi, M. A. Aiello, J. F. Berryman, J. R. Grohs, M. D. DeVore, W. Storm, J. Wadley, and G. Tallant. Runtime validation and verification for safety critical flight control systems. Technical Report AFRL-RB-WP-TR-2009-3071, Barron Associates, Inc., 2009.

[84] D. Seto, B. Krogh, L. Sha, and A. Chutinana. Dynamic control system upgrade using the simplex architecture. *IEEE Control Systems*, 18(4):72–80, 1998.

[85] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 86–97, May 1989.

[86] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, August 2004.

[87] W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[88] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics about runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.

[89] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification (FATES/RV'06)*, August 2006.

[90] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, February 2006.

[91] K. Chen, S. Malik, and P. Patra. Runtime validation of transactional memory systems. In *International Symposium on Quality Electronic Design*, pages 750–756, 2008.

[92] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.

[93] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[94] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330, 2000.

[95] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356, April 2002.

[96] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

[97] E. Bodden. A lightweight LTL runtime verification tool for Java. In *9th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 306–307, October 2004.

[98] K. Havelund and G. Rosu. Monitoring Java programs with JavaPathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2001.

[99] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, March 2004.

[100] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An overview of the MOP runtime verification framework. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.

[101] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57, January 2004.

[102] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Proceedings of the 7th Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, March 2007.

[103] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *In Proceedings of FORMATS-FTRTFT. Volume 3253 of LNCS*, pages 152–166, 2004.

[104] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of Computer Science*, volume 4800 of *Lecture Notes in Computer Science*, pages 475–505. Springer-Verlag, 2008.

[105] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[106] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, March 2004.

[107] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, pages 2103–2110, 2010.

[108] H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proceedings of 17th International Symposium on Formal Methods(FM'11)*, volume 6664 of *LNCS*, June 2011.

[109] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[110] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, July 2001.

[111] W. Bartussek and D.L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Proceedings of the 2nd Conference on European Cooperation in Informatics*, volume 65 of *LNCS*, pages 211–236, 1978.

[112] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. *ACM SIGPLAN Notices - ICFP '11*, 46:176–188, September 2011.

[113] U. Sammapun, I. Lee, O. Sokolsky, and J. Regehr. Statistical runtime checking of probabilistic properties. In *Proceedings of the 7th Workshop on Runtime Verification*, volume 4839 of *LNCS*, pages 164–175, March 2007.

[114] A. Sistla and Abhigna Srinivas. Monitoring temporal properties of stochastic systems. In *Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2008.

[115] Lars Grunske and Pengcheng Zhang. Monitoring probabilistic properties. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 183–192, 2009.

[116] A. Sistla, Milo Zefran, and Yao Feng. Monitorability of stochastic dynamical systems. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 720–736. Springer, 2011.

[117] Zhiwei Wang, M.H. Zaki, and S. Tahar. Statistical runtime verification of analog and mixed signal designs. In *Proc. of the 3rd International Conference on Signals, Circuits and Systems (SCS)*, pages 1–6, nov. 2009.

[118] D.K. Peters and D.L. Parnas. Requirements-based monitors for real-time systems. *Software Engineering, IEEE Transactions on*, 28(2):146 –158, feb 2002.

[119] Cristina M. Wilcox and Brian C. Williams. Runtime verification of stochastic, faulty systems. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 452–459, 2010.

[120] Brian C. Williams, Seung Chung, and Vineet Gupta. Mode estimation of model-based programs: monitoring systems with complex behavior. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, pages 579–585, 2001.

[121] Scott D. Stoller, Ezio Bartocci, Radu Grosu, Havelund Klaus, Scott A. Smolka, Seyster Justin, and Erez Zadok. Runtime Verification with State Estimation. In *In Proc. of RV 2011: 2nd International Conference on Runtime Verification*, 2011.

[122] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *Proceedings of the 29th Real-Time Systems Symposium (RTSS'09)*, pages 481–491, December 2008.

[123] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, June 1997.

[124] M. D'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Workshop on Dynamic Program Analysis (WODA'05)*, 2005.

[125] C. Allan, P. Avgustinov, S. Kuzins, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, A. S. Christensen, L. Hendren, and O. Lhoták. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 345–364, October 2005.

[126] C. Colombo, G. Pace, and P. Abela. Compensation-aware runtime monitoring. In *Proceedings of the 1st International Conference on Runtime Verificaiton (RV'10)*, volume 6418 of *LNCS*, pages 214–228, November 2010.

[127] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S.A. Smolka, S.D. Stoller, and E. Zadok. Aspect-oriented instrumentation with GCC. In *Proceedings of the 1st International Conference on Runtime Verification*, volume 6418 of *LNCS*, pages 405–420, November 2010.

[128] Samaneh Navabpour, Chun Wah Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Proc. of the 2nd International Conference on Runtime Verification (RV)*, September 2011.

[129] M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE'07)*, pages 124–133, November 2007.

[130] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *International Conference on Software Engineering (ICSE'10)*, pages 5–14, May 2010.

[131] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 525–549, July 2007.

[132] E. Bodden and L. Hendren. The Clara framework for hybrid typestate analysis. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.

[133] Deian Tabakov and Moshe Vardi. Optimized temporal monitors for systemc. In *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 436–451. Springer Berlin / Heidelberg, 2010.

[134] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.

[135] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: can optimizing error detection improve fault diagnosis? In *Proceedings of the First international conference on Runtime verification*, pages 36–50, 2010.

[136] D. Seto, B. Krogh, L. Sha, and A. Chutinan. The simplex architecture for safe online control system upgrades. In *Proceedings of the 1998 American Control Conference*, 1998.

[137] A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)*, pages 243–252, April 2006.

[138] S. Tripakis. A combined on-line/off-line framework for black-box fault diagnosis. In *Proceedings of the 9^{th} Workshop on Runtime Verification (RV'09)*, pages 152–167, July 2009.

[139] Michael Wagner, Phil Koopman, John Bares, and Chris Ostrowski. Building safer ugvs with run-time safety invariants. In *National Defense Industrial Association (NDIA) Systems Engineering Conference*, 2009.

[140] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[141] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.

[142] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *Software Tools for Technology Transfer, Special Section on Runtime Verification*, in this volume, 2011.

[143] E. A. Lee. Computing needs time. *Communications of the ACM*, 52(5), May 2009.

[144] Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, and Lui Sha. The system-level simplex architecture for improved real-time embedded system safety. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:99–107, 2009.

[145] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[146] David Simon. *An Embedded Software Primer*. Addison-Wesley, 1999.

[147] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[148] Mike Jones. What really happened on Mars? Webpage (retrieved April 2012). `http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/`, 1997.

[149] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Muller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, April 2008.

[150] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[151] Hermann Kopetz. *Real-Time Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

[152] John Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, volume 2211 of *LNCS*, pages 306–323. Springer-Verlag, October 2001.

[153] Wilfredo Torres-Pomales, Mahyar R. Malekpour, and Paul Miner. ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center, 2005.

[154] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.

[155] K. Driscoll, B. Hall, Håkan Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP*, pages 235–248. Springer, 2003.

[156] Ricky W. Butler. A primer on architectural level fault tolerance. Technical Report NASA/TM-2008-215108, NASA Langley Research Center, 2008.

[157] Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010. Available at `http://ntrs.nasa.gov/search.jsp?R=278742&id=3&as=false&or=false&qs=Ns%3DArchiveName%257c0%26N%3D4294643047`.

[158] F. Jahanian, R. Rajkumar, and S. Raju. Run-time monitoring of timing constraints in distributed real-time systems. *Real-Time Systems Journal*, 7(2):247–273, 1994.

[159] R. Alur and T. Henzinger. Logics and models of real time: A survey. In *Real Time Theory and Practice*, volume LNCS. Springer-Verlag, 1992.

[160] R. Alur and T. Henzinger. Real time logics: complexity and expressiveness. In *Fifth Annual Symposium on Logic in Computer Science*, pages 390–401. IEEE Computer Society Press, 1990.

[161] A. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, pages 252–262, 1997.

[162] C. Lee. *Monitoring and Timing Constraints and Streaming Events with Temporal Uncertainties*. PhD thesis, University of Texas, 2005.

[163] T. Savor and R. Seviora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136–147, 1997.

[164] Paul Miner, Alfons Geser, Lee Pike, and Jeffery Maddalon. A unified fault-tolerance protocol. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *LNCS*, pages 167–182. Springer, 2004. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unified.html`.

[165] M.B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 228–237, 2008.

[166] S. Fischmeister and P. Lam. Time-aware instrumentation of embedded software. *Industrial Informatics, IEEE Transactions on*, 6(4):652 –663, nov. 2010.

[167] S. Fischmeister and Y. Ba. Sampling-based program execution monitoring. In *ACM International conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 133–142, 2010.

[168] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In *17th Intl. Symposium on Formal Methods (FM)*, 2011.

[169] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[170] P. Caspi, D. Pialiud, N. Halbwachs, and J. Plaice. LUSTRE: a declratative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[171] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), 1991.

[172] B. D'Angelo, S. Sankaranarayanan, C. Snchez, W. Robinson, Zohar Manna, B. Finkbeiner, H. Spima, and S. Mehrotra. LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation adn Reasoning*, pages 166–174. IEEE Computer Society Press, 2005.

[173] Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In *Proceedings of the 2nd Intl. Conference on Runtime Verification*, LNCS. Springer, September 2011.

[174] Lee Pike. Copilot: Monitoring embedded systems. Technical Report NASA/CR-2012-217329, NASA Langley Research Center, January 2012. Available at `http://ntrs.nasa.gov/search.jsp?R=20120001989&hterms=pike+goodloe&qs=Ntx%3Dmode%2520matchallpartial%2520%26Ntk%3DAll%26N%3D0%26Ntt%3Dpike%2520goodloe`.

[175] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. Submitted. Available at `https://www.cs.indiana.edu/~lepike/pubs/copilot-assurance.pdf`, 2012.

[176] Elizabeth Latronico, Paul Miner, and Philip Koopman. Quantifying the reliability of proven SPIDER group membership service guarantees. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 275, Washington, DC, USA, 2004. IEEE Computer Society.

[177] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *RTSS'08: Proceedings of the 29th IEEE Real-Time System Symposium*, pages 481–491, 2008.

[178] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Roşu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 481–491, Washington, DC, USA, 2008. IEEE Computer Society.

[179] Elizabeth Latronico. Design time reliability analysis of distributed fault tolerance algorithms. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 486–495. IEEE, 2005.

[180] Elizabeth Latronico and Philip Koopman. Design time reliability analysis of distributed fault tolerance algorithms. *International Conference on Dependable Systems and Networks*, pages 486–495, 2005.

[181] RTCA. Software considerations in airborne systems and equipment certification. RTCA, Inc., 1992. RCTA/DO-178B.

[182] Design assurance guidelines for airborne electronic hardware. RTCA, Inc., 2000. RCTA/DO254.

[183] John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report CR–4551, NASA, December 1993.

[184] John Rushby. Runtime certification. In *Eighth Workshop on Runtime Verification (RV08)*, volume 5289 of *LNCS*, pages 21–35, 2008. Available at `http://www.csl.sri.com/users/rushby/abstracts/rv08`.

[185] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proc. of the IEEE*, 91(1):145–164, Jan 2003.

[186] The MathWorks, Inc. Matlab/Simulink/Stateflow Tools. http://www.mathworks.com.

[187] A. L. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proc. of the IEEE*, 95(3):467–506, March 2007.

[188] Martin Ohlin, Dan Henriksson, and Anton Cervin. *TrueTime 1.5 Reference Manual*. Dept. of Automatic Control, Lund Univ., Sweden, January 2007. http://www.control.lth.se/truetime/.

[189] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proc. of the IEEE*, 91(1):112–126, Jan 2003.

[190] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and Shige Wang. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29 –44, jan. 2012.

[191] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[192] E. Lee and A. Sangiovanni-Vincentelli. A unified framework for comparing models of computation. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[193] Michael W. Whalen, David A. Greve, and Lucas G. Wagner. *Model Checking Information Flow*. Springer-Verlag, Berlin Germany, March 2010.

[194] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[195] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11), 2003.

[196] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *Software, IEEE*, 28(3):41 –48, may-june 2011.

[197] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *RTSS '04: Proc. of the IEEE Real-Time Systems Symp.*, pages 57–67, Dec 2004.

[198] Arvind Easwaran. *Advances in hierarchical real-time systems: Incrementality, optimality, and multiprocessor clustering*. PhD thesis, Univ. of Pennsylvania, 2008.

[199] Joseph Porter. *Compositional and Incremental Modeling and Analysis for High-Confidence Distributed Embedded Control Systems*. PhD thesis, Vanderbilt University, 2011.

[200] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, 2000.

[201] Ernesto Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland, Sep 2006.

[202] Joseph Porter, Graham Hemingway, Harmon Nine, Chris vanBuskirk, Nicholas Kottenstette, Gabor Karsai, and Janos Sztipanovits. The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Language, Framework, and Analysis. Sep 2010.

[203] G. Hemingway, J. Porter, N. Kottenstette, H. Nine, C. vanBuskirk, G. Karsai, and J. Sztipanovits. Automated Synthesis of Time-Triggered Architecture-based TrueTime Models for Platform Effects Simulation and Analysis. In *RSP '10: 21st IEEE Intl. Symp. on Rapid Systems Prototyping*, Jun 2010.

[204] Ryan Thibodeaux. The specification and implementation of a model of computation. Master's thesis, Vanderbilt Univ., May 2008.

[205] Jung Soo Kim and David Garlan. Analyzing architectural styles. *J. Syst. Softw.*, 83(7):1216–1235, July 2010.

[206] Akshay Rajhans, Ajinkya Bhave, Sarah Loos, Bruce Krogh, Andre Platzer, and David Garlan. Using parameters in architectural views to support heterogeneous design and verification. In *50th IEEE Conference on Decision and Control (CDC) and European Control Conference (ECC)*, Orlando, FL, 12-15 December 2011.

[207] A. Bhave, B.H. Krogh, D. Garlan, and B. Schmerl. View consistency in architectures for cyber-physical systems. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*, pages 151 –160, april 2011.

[208] SAE Architecture Analysis & Design Language (AADL) Fact Sheet, 2007.

[209] D. Delanote, S. Van Baelen, W. Joosen, and Y. Berbers. Using AADL in Model Driven Development. In A. Canals, S. Gerard, and I. Perseil, editors, *IEEE-SEE Intl. Workshop on UML and AADL 2007, Intl. Conf. on Eng. Complex Computer Systems (ICECCS07)*, Auckland, New Zealand, Jul 2007.

[210] Peter H. Feiler, Bruce A. Lewis, and Steve Vestal. The SAE Architecture Analysis & Design Language (AADL) – A Standard for Engineering Performance Critical Systems. In *IEEE International Symposium on Computer-Aided Control Systems Design (CACSD)*, Oct 2006.

[211] G. Gössler and J. Sifakis. Composition for component-based modeling. In *Proc. of FMCO'02, Springer LNCS 2852*, pages 443–466, Leiden, the Netherlands, Nov 2002.

[212] Simon Bliudze and Joseph Sifakis. The Algeba of Connectors - Structuring Interaction in BIP. *IEEE Trans. on Computers*, 57(10):1315–1330, Oct 2008.

[213] Saddek Bensalem, Lavindra de Silva, Andreas Griesmayer, Felix Ingrand, Axel Legay, and Rongjie Yan. A formal approach for incremental construction with an application to autonomous robotic systems. In Sven Apel and Ethan Jackson, editors, *Software Composition*, volume 6708 of *Lecture Notes in Computer Science*, pages 116–132. Springer Berlin / Heidelberg, 2011.

[214] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, April 2005.

[215] Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio. Runtime monitoring of functional component changes with behavior models. In *Models in Software Engineering Workshops and Symposia at MoDELS 2011*, Wellington, New Zealand, October 2011.

[216] Shanna-Shaye Forbes. Real-time C Code Generation in Ptolemy II for the Giotto Model of Computation. Master's thesis, EECS Department, University of California, Berkeley, May 2009.

[217] Christoph M. Kirsch, Marco A.A. Sanvido, and Thomas A. Henzinger. A programmable microkernel for real-time systems. In *First Intl. Conf. on Virtual Execution Environments (VEE)*, pages 35–45. ACM Press, 2005.

[218] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. *Ada Lett.*, XXIV(4):1–8, November 2004.

[219] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*, SIGAda '04, pages 1–8, New York, NY, USA, 2004. ACM.

[220] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of aadl models. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.

[221] Shenglin Gui, Lei Luo, Yun Li, and Lijie Wang. Formal schedulability analysis and simulation for aadl. In *Embedded Software and Systems, 2008. ICESS '08. International Conference on*, pages 429 –435, july 2008.

[222] Wei Zheng, Jike Chong, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *ACSD '05: Proc. of the Fith Intl. Conf. on App. of Concurrency to System Design*, pages 132–141, June 2005.

[223] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, Oct. 2000.

[224] Joseph Porter, Gabor Karsai, and Janos Sztipanovits. Towards a time-triggered schedule calculation tool to support model-based embedded software design. In *EMSOFT '09: Proc. of ACM Intl. Conf. on Embedded Software*, Grenoble, France, Oct 2009.

[225] Norman Scaife and Paul Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *ECRTS*, pages 119–126. IEEE Computer Society, 2004.

[226] P. Pop, P. Eles, Zebo Peng, and T. Pop. Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(8):793 – 811, aug. 2004.

[227] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. The compass approach: Correctness, modelling and performability of aerospace systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 173–186, Berlin, Heidelberg, 2009. Springer-Verlag.

[228] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. *The Computer Journal*, 54(5):754–775, 2011.

[229] Roberto Varona-Gomez and Eugeni Villar. AADL simulation and performance analysis in SystemC. In *14th IEEE Intl. Conf. on Engineering of Complex Computer Systems (ICECCS 09)*, pages 323–328, Washington, DC, 2009. IEEE Computer Society.

[230] H.L.S. Younes, M.Z. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *STTT*, 8(3):216–228, 2006.

[231] Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501 – 510, 1998.

[232] J.A. Rowson. Hardware/software co-simulation. In *Design Automation, 1994. 31st Conference on*, pages 439 – 440, june 1994.

[233] Charles H. Roth. *Digital Systems Design Using VHDL*. Wadsworth Publ. Co., Belmont, CA, USA, 1998.

[234] Samir Palnitkar. *Verilog&#174; hdl: a guide to digital design and synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[235] A. Hoffman, T. Kogel, and H. Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 760–765, 2001.

[236] Amal Banerjee and Andreas Gerstlauer. Transaction level modeling of best-effort channels for networked embedded devices. In *Proceedings of IESS*, pages 77–88, 2009.

[237] P.R. Panda. Systemc - a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75 – 80, 2001.

[238] Masahiro Fujita and Hiroshi Nakamura. The standard specc language. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 81–86, 2001.

[239] S. Sutherland, S. Davidmann, , and P. Flake. *SystemVerilog For Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer, New York, 2004.

[240] IEEE Standard SystemC Language Reference Manual, 2011.

[241] Michael Lafaye, Laurent Pautet, Etienne Borde, Marc Ganti, and David Faura. Model driven resource usage simulation for critical embedded systems. In *Design, Automation, & Test in Europe (DATE'12)*, 2012.

[242] R. Varona-Gomez and E. Villar. Aadl simulation and performance analysis in systemc. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 323 –328, june 2009.

[243] E. de las Heras and E. Villar. Specification for systemc-aadl interoperability. In *Intelligent Solutions in Embedded Systems, 2007 Fifth Workshop on*, pages 76 –86, june 2007.

[244] J. Porter and G. Hemingway. The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Language, Framework, and Analysis. Technical Report ISIS-10-109, ISIS, Vanderbilt Univ., 2010.

[245] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. *SIGPLAN Not.*, 38(7):153–162, June 2003.

[246] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84 – 99, jan 2003.

[247] Wolfgang Pree and Josef Templ. Modeling with the timing definition language (tdl). In *Model-Driven Development of Reliable Automotive Services*, volume 4922 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin / Heidelberg, 2008.

[248] IEEE Standard 1516: Standard for Modeling and Simulation High Level Architecture, 2010.

[249] Emeka Eyisi, Jia Bai, Derek Riley, Jiannian Weng, Yan Wei, Yuan Xue, Xenofon Koutsoukos, and Janos Sztipanovits. NCSWT: an integrated modeling and simulation tool for networked control systems. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, HSCC '12, pages 287–290, New York, NY, USA, 2012. ACM.

[250] Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits, and Gabor Karsai. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *Simulation*, 88(2):217–232, February 2012.

[251] T. Blochwitz et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th Modelica Conference*, 2011.

[252] M. Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Models in software engineering. chapter Translating AADL into BIP - Application to the Verification of Real-Time Systems, pages 5–19. 2009.

[253] Lei Pi, Jean-Paul Bodeveix, and Mamoun Filali. Modeling AADL Data Communication with BIP. In Fabrice Kordon and Yvon Kermarrec, editors, *Reliable Software Technologies: Ada-Europe 2009*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

[254] Matthew Staats and Mats P. Heimdahl. Partial translation verification for untrusted code-generators. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 226–237, Berlin, Heidelberg, 2008. Springer-Verlag.

[255] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[256] Gabor Karsai, Gautam Biswas, Sriram Narasimhan, Tal Pasternak, Sherif Abdelwahed, Tivadar Szemethy, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. *Towards Fault-Adaptive Control of Complex Dynamical Systems*, pages 347–368. John Wiley & Sons, Inc., 2005.

[257] G. Karsai, G. Biswas, S. Abdelwahed, N. Mahadevan, and E. Manders. Model-based software tools for integrated vehicle health management. In *Space Mission Challenges for Information Technology, 2006. SMC-IT 2006. Second IEEE International Conference on*, pages 8 pp. –442, 0-0 2006.

[258] Meng Ji, Zhen Zhang, G. Biswas, and N. Sarkar. Hybrid fault adaptive control of a wheeled mobile robot. *Mechatronics, IEEE/ASME Transactions on*, 8(2):226 –233, june 2003.

[259] M.J. Daigle, X.D. Koutsoukos, and G. Biswas. Distributed diagnosis in formations of mobile robots. *Robotics, IEEE Transactions on*, 23(2):353 –369, april 2007.

[260] M.J. Daigle, I. Roychoudhury, G. Biswas, X.D. Koutsoukos, A. Patterson-Hine, and S. Poll. A comprehensive diagnosis methodology for complex hybrid systems: A case study on spacecraft power distribution systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 40(5):917 –931, sept. 2010.

[261] G. Biswas, E-J. Manders, J. Ramirez, N. Mahadevan, and S. Abdelwahed. Online model-based diagnosis to support autonomous operation of an advanced life support system. *Habitation: An International Journal for Human Support Research*, 10(1):21–38, 2004.

[262] L. Grunske and Jun Han. A Comparative Study into Architecture-Based Safety Evaluation Methodologies Using AADL's Error Annex and Failure Propagation Models. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 283 –292, dec. 2008.

[263] A. Dubey, G. Karsai, and N. Mahadevan. Model-based software health management for real-time systems. In *Aerospace Conference, 2011 IEEE*, pages 1 –18, march 2011.

[264] Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai. Application of software health management techniques. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 1–10, New York, NY, USA, 2011. ACM.

[265] Abhishek Dubey, Nagabhushan Mahadevan, and Gabor Karsai. The inertial measurement unit example: A software health management case study. Technical Report ISIS-12-101, ISIS, Vanderbilt University, Feb 2012.

[266] M. Bauer and W. Ecker. Hardware/software co-simulation in a vhdl-based test bench approach. In *Design Automation Conference, 1997. Proceedings of the 34th*, pages 774 –779, jun 1997.

[267] M. Nodine. Automatic testbench generation for rearchitected designs. In *Microprocessor Test and Verification, 2007. MTV '07. Eighth International Workshop on*, pages 128 –136, dec. 2007.

[268] M. Hause, A. Stuart, D. Richards, and J. Holt. Testing Safety Critical Systems with SysML/UML. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 325 –330, march 2010.

[269] Zsolt Lattmann, Adam Nagel, Jason Scott, Kevin Smyth, Johanna Ceisel, Chris vanBuskirk, Joseph Porter, Ted Bapty, Sandeep Neema, Dimitri Mavris, and Janos Sztipanovits. Towards automated evaluation of vehicle dynamics in system-level designs. In *to appear, ASME IDETC/CIE 2012*, Chicago, IL, 2012.

[270] Lou Scheffer, Luciano Lavagno, and Grant Martin. *Electronic design automation for integrated circuits handbook*. CRC/Taylor & Francis, Boca Raton, FL, 2006.

[271] P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, and M.Y. Levin. Automating software testing using program analysis. *Software, IEEE*, 25(5):30 –37, sept.-oct. 2008.

[272] Zoltan Micskei and Istvan Majzik. Model-based automatic test generation for event-driven embedded systems using model checkers. In *Proceedings of the International Conference on Dependability of Computer Systems*, DEPCOS-RELCOMEX '06, pages 191–198, Washington, DC, USA, 2006. IEEE Computer Society.

[273] Yongfeng Yin and Bin Liu. A method of test case automatic generation for embedded software. In *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, pages 1 –5, dec. 2009.

[274] Li Tan, Jesung Kim, and Insup Lee. Testing and monitoring model-based generated program. In *Third Workshop on Runtime Verification (RV'03)*, volume 89:2 of *Electronic Notes in Theoretic Computer Science*. Elsevier Science, 2003.

[275] Jun-Wei Lin and Chin-Yu Huang. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology*, 51(4):679 – 690, 2009.

[276] S.. Bogosyan, M.. Gokasan, and D.J. Goering. A novel model validation and estimation approach for hybrid serial electric vehicles. *Vehicular Technology, IEEE Transactions on*, 56(4):1485 –1497, july 2007.

[277] Alexandre Donzé, Bruce Krogh, and Akshay Rajhans. Parameter synthesis for hybrid systems with an application to simulink models. In *Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control*, HSCC '09, pages 165–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[278] R. Henftling, A. Zinn, M. Bauer, W. Ecker, and M. Zambaldi. Platform-based testbench generation. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 1038 – 1043, 2003.

[279] Haocheng Huang, Aiwu Ruan, Yongbo Liao, Jianhua Zhu, Lin Wang, Chuanyin Xiang, and Pin Li. A new event driven testbench synthesis engine for FPGA emulation. In *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pages 373 –376, oct. 2011.

[280] N.A. Banciu and G. Toacse. Testbench components verification using fault injection techniques. In *Optimization of Electrical and Electronic Equipment (OPTIM), 2010 12th International Conference on*, pages 997 –1003, may 2010.

[281] A.W. Ruan, H.C. Huang, C.Q. Li, Z.J. Song, Y.B. Liao, and W. Tang. Debugging methodology for a synthesizable testbench fpga emulator. In *Integrated Circuits (ISIC), 2011 13th International Symposium on*, pages 593–596, dec. 2011.

[282] Lui Sha and José Meseguer. In Martin Wirsing, Jean-Pierre Banâtre, Matthias Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, chapter Design of Complex Cyber Physical Systems with Formalized Architectural Patterns, pages 92–100. Springer-Verlag Berlin Heidelberg, 2008.

[283] Lee Pike, Geoffrey Brown, and Alwyn Goodloe. Roll your own test bed for embedded real-time protocols: A Haskell experience. In Stephanie Weirich, editor, *Proceedings of the ACM SIGPLAN Haskell Symposium*, pages 61–68. ACM, 2009.

[284] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, ASM'03, pages 87–108, Berlin, Heidelberg, 2003. Springer-Verlag.

[285] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109(0):43 – 56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[286] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, November 2005.

[287] S. Narasimhan and G. Biswas. Model-based diagnosis of hybrid systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 37(3):348 –361, may 2007.