Contract-Based Verification of Complex Time-Dependent Behaviors in Avionic Systems

Devesh Bhatt[1], Arunabh Chattopadhyay[1], Wenchao Li[2], David Oglesby[1(✉)], Sam Owre[2], and Natarajan Shankar[2]

[1] Honeywell Aerospace Labs, Golden Valley, USA
david.oglesby@honeywell.com
[2] SRI International, Silicon Valley, USA

Abstract. Avionic systems involve complex time-dependent behaviors across interacting components. This paper presents a contract-based approach for formally verifying these behaviors in a compositional manner. A unique feature of our contract-based tool is the support of architectural specification for multi-rate platforms. An abstraction technique has also been developed for properties related to variable time bounds. Preliminary results on applying this approach to the verification of an aircraft cabin pressure control system are promising.

Recent years have seen a large growth in the size and complexity of avionics systems due to increasing system functionality and closer integration among existing and new aircraft subsystems. Verifying the safety of these systems and their compliance with their intended functional requirements is a critical certification objective that is becoming prohibitively expensive due to this increasing complexity. Compositional verification approaches [1,2] manage the verification complexity by decomposing requirements into component-level contracts and applying assume-guarantee reasoning. Contract-based tools such as AGREE [3] and OCRA [2] have been used in recent effort to formally verify control modules in avionic systems. Most existing work has focused primarily on verifying safety properties, e.g., the system must not exhibit behaviors that can result in a catastrophic failure. Using such techniques to verify complex behavioral requirements of a system in a distributed setting, however, has not received much attention. In many avionics application domains such as flight controls, cockpit displays, flight management, and environment control systems, a variety of complex time-dependent behaviors are present that can cut across several components. Verifying such behaviors for compliance to the intended functional requirements is essential and has been traditionally accomplished using testing techniques that can be expensive but still not exhaustive in revealing the presence of errors. There is a strong need for a verification approach that can enable scalable use of formal verification (e.g., model checking) tools for complex time-bounded properties and the composition of such properties over components in

This research was supported in part by NASA Contract NNA13AC55C and DARPA under agreement number FA8750-16-C-0043.

S. Rayadurgam and O. Tkachuk (Eds.): NFM 2016, LNCS 9690, pp. 34–40, 2016.
DOI: 10.1007/978-3-319-40648-0_3

a distributed environment. This short paper presents a compositional approach of verifying such systems involving complex time-dependent behaviors.

1 Case Study Example

The motivating case study for this work comes from aircraft cabin pressure and environment control system applications. In a typical mechanical system with sensors and actuators, it is necessary to perform calibration, initialization, or other built-in-tests on these components (e.g., a pressure sensor) only at certain specific times in successive aircraft takeoff and landing cycles. The specific signal to invoke this activity is called `finalize_event` in this example, and it is triggered after the aircraft door is open for a minimum amount of time. Figure 1 shows the components involved in triggering this event. The Timing Computation module computes the amount of time the door should be open before the `finalize_event` is triggered as a function of altitude reached during flight. The Mode Transition Logic controls the changes in aircraft modes, for example `LANDING` (`L`) to `GROUND` (`G`). The Mode Detect Logic module evaluates sensor values to determine the current aircraft state, for example, if the aircraft is on the ground or climbing. The mode, state, and timing all inform the decision when to finalize the aircraft. This example illustrates an end-to-end subsystem from sensors to actuator, even though it is only a small part of an environmental control system. Depending on the architectural platform, components in this subsystem may be physically distributed across the aircraft. For instance, each component executes periodically (with different rates) and communicates with one another over a shared bus.

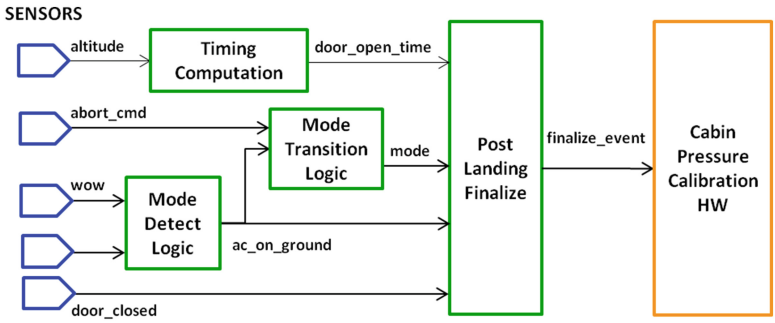

Fig. 1. Case study example – diagram of interacting component subsystems

The `finalize_event` signal is activated by several events and aircraft states occurring in a particular temporal sequence shown in the Simulink diagram of Fig. 2. The door may open any time after landing, and once it has been open continuously for `door_open_time`, the `finalize_event` is broadcast.

Additionally, the activation must occur only once in a landing-to-takeoff cycle even through certain states of aircraft (e.g., door opening or closing) may change

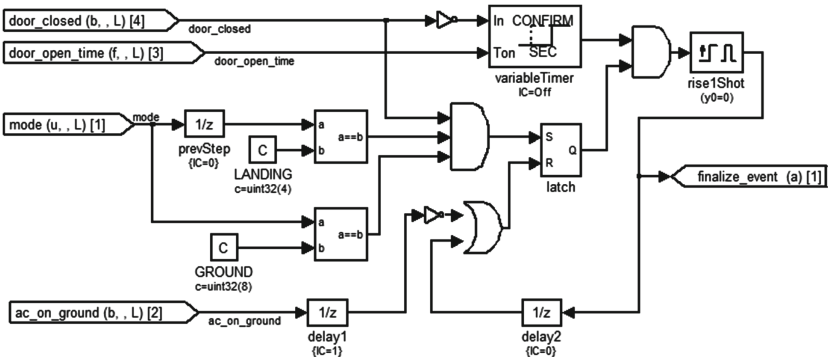

Fig. 2. Diagram of "Post Landing Finalize" subsystem

Table 1. Requirements for the "Post Landing Finalize" subsystem

Requirement 1	A finalize event will be broadcast after the aircraft door has been open continuously for `door_open_time` seconds while the aircraft is on the ground after a successful landing.
Requirement 2	A finalize event is broadcast only once while the aircraft is on the ground.
Requirement 3	The finalize event will not occur during flight.
Requirement 4	The finalize event will not be enabled while the aircraft door is closed.

multiple times after a landing and before the next takeoff. It is also essential that the activation state for `finalize_event` is reset after broadcast so that it is ready to be activated for the next landing-takeoff cycle. The requirements for the "Post Landing Finalize" subsystem are outlined in Table 1.

The assumptions in Table 2 must be proven on upstream components. The proofs of the requirements for this module depend on the possibility of the necessary sequence of inputs. Note that because the modules may run at different rates, the assumption that mode transitions directly from `LANDING` to `GROUND` does not require a direct transition in the Mode Transition Logic module. It could be possible to transition through a third state, as long as it is possible to reach `Ground` before Post Landing Finalize can see the intermediate state. The translation of assumptions to requirements on upstream components must take into account different rates of execution. Assumption 3 is not specific about the definition of "aircraft is on the ground." This would likely be formalized as `altitude` is within some tolerance of zero since that is the only signal into the Timing Computation module on which the proof of this assumption will fall. Assumption 4 is more of an assertion that the door will not be open during flight. As an user-driven input signal, this cannot be proven using upstream modules; but it does assure Requirement 3.

Table 2. Assumption for the "Post Landing Finalize" subsystem

Assumption 1	`ac_on_ground` can be true before the mode transitions to `GROUND`.
Assumption 2	The mode can transition directly from `LANDING` to `GROUND` as observed by "Post Landing Finalize."
Assumption 3	`door_open_time` does not change while the aircraft is on the ground.
Assumption 4	`door_closed` must be true if `ac_on_ground` is false.

2 Verification Techniques and Tools

2.1 Translation of Simulink Models and Contract Language

The integration of Simulink into a high-assurance design flow can bring significant benefits especially for safety-critical applications. Our tool `Sim2SAL` automates the generation of formal models by translating Simulink designs into transition systems in SAL. A unique feature in our tool is the support for real-time multi-rate systems. `Sim2SAL` allows the specification of the multi-rate architecture as annotations attached to Simulink subsystems. An example annotation is shown below that describes the timing characteristics of a periodically executing subsystem.

```
arch_begin
period: 5 ms;     /* the subsystem executes every 5 ms */
jitter: 0.1 ms;   /* max jitter of the clock is 0.1 ms */
latency[1]: 1 ms; /* max latency at input port 1 is 1 ms */
init[1]: 0;       /* initial value at input port 1 is 0 */
arch_end
```

The formal model of computation for these systems is given in [4]. Several abstractions of multi-rate systems are also possible and handled automatically, including zero communication delays and zero jitters. In this paper, we present the use of `Sim2SAL` in verifying the properties of the case study example in the discrete-time setting. In reality, the system is implemented on a distributed architecture with periodic components. Hence, the multi-rate model is more faithful to the actual implementation. In such a setting, conventional assume-guarantee methods are insufficient since they were developed for discrete transition systems. We explore compositional techniques for verifying these models in future work.

In short, the contract language of `Sim2SAL` consists of two parts – behavioral specification and architectural specification. Similar to OCRA [2], the behavioral part is based on Linear Temporal Logic (LTL) [5] over assume-guarantee pairs. In addition to the typical syntax of LTL, additional constructs are provided for simplifying expressions that involve large step sizes. For instance, we use $\mathbf{G}\_[\mathbf{0}, \mathbf{50}]\ f$ to specify that the formula f has to hold true during the next 50 time steps including the current one. Certain real-time properties can also be specified at the system level. For example, an end-to-end property may require that a request will always be serviced by a response within some t time units, where $t \in \mathbb{R}$. `Sim2SAL` uses additional timer constructs and translates these properties

to an equivalent LTL formula involving these timers. We refer interested readers to [4] for details of this technique. Currently, `Sim2SAL` can handle a subset of Simulink's discrete blocks. `Sim2SAL` takes the Simulink model (.mdl file) and the contract annotations as input and generates an encoding of the model and properties in SAL.

2.2 Abstraction Pattern for Variable Time Bounds

Dwyer et al. have created a Specification and Pattern System [6] for defining various types of behaviors. Tool chains have been prototyped [7] to facilitate the capture of requirements using such patterns such as minimum duration, bounded response etc. For example, the *minimum duration* pattern specifies the minimum amount of time a formula must hold once it becomes true. There are, however, practical considerations of model-checking complexity; a large time bound can translate into a large number of discrete steps to be explored, resulting in exponential growth of the search space. Another consideration in many systems is that variable time bounds are provided to a component that are dynamically computed by another component.

For example, for the "Post Landing Finalize" component, Requirement 1 in Table 1 specifies a timed response where the value of `door_open_time` for the response is dynamically computed by "Timing Calculation" component. The Simulink model of this component (Fig. 2) contains a block named `variableTimer` that receives a variable external input `door_open_time`. Due to the dynamic value of time, one cannot express the LTL property for this requirement as a fixed duration response. Furthermore, a proof of the requirement's property for a fixed value of time will not be valid when the value of time can be different in the dynamical system.

A Timer Abstraction Pattern. Our approach is to create a *timer abstraction pattern* to construct and verify properties related to variable time bounds in a compositional manner. Consider the changes to the states and signals in the model of Fig. 2, as the behavior for Requirement 1 is realized. Initially the state of the `finalize_event` is false when the aircraft is landing: the `door_closed` input is true and `ac_on_ground` is true before mode changes from `LANDING` to `GROUND`. When the `door_closed` input is false (the door is open), the `variableTimer` block starts counting. Note that the relevant state variables do not change in the time steps when the timer is just counting up. This establishes a bound on the minimum number of steps to which the timer behavior can be collapsed while still preserving the properties. Such an abstraction is useful in making the verification problem tractable for the compositional verification of interacting components with variable time bounds. For example, we can prove the properties on the "Post Landing Finalize" component using a small time bound, t_{min}, and specify that as an assumption for this component. We can then independently prove this assumption, using a static analysis tool, as a guarantee provided by the "Timing Calculation" component that it will always produce a value of `door_open_time` $\geq t_{min}$. This allows the application of different types of model checking and static analysis tools for proving properties for different components.

Computation of Minimum Time-Steps Bound for the Timer Abstraction. We briefly describe the algorithm to compute the minimum number of time steps $tsteps_{min}$ required to explore all possible input to output behaviors in the model while the timer (e.g., `variableTimer`) is counting. We start by noting that all states in the Simulink model of interest are expressed by `unit delay` blocks: a unit delay block stores the last value of its input and outputs that stored value in the current step (higher-level blocks such as `latch` and `rise1Shot` use `unit delay` blocks to hold a state variable). If a path from an input to an output contains a unit delay in series (e.g., block `prevStep`) then that adds 1 step to $tsteps_{min}$ since with 1 additional step one can produce a value at the output of the unit delay that is possible to be created at its input. A Boolean unit delay in a feedback loop (e.g., `delay2`) also adds 1 to $tsteps_{min}$, since the feedback of a Boolean variable creates a repeating sequence the second time around the loop. Feedback loops that create an accumulator using a numeric variable can exhibit a long non-repeating sequence of values and such models are excluded from our analysis. Our observations across avionics systems show that most instances of timer patterns do not have other accumulators in the same subgraph as the timer. The algorithm is simply to enumerate all unique paths from models inputs to relevant outputs, count the number of unit delays (including feedback) in each path, then compute $tsteps_{min}$ as the largest count across all paths. This algorithm provides a conservative bound on $tsteps_{min}$. A precise smaller bound can be derived by considering constraints on the variables used in the property. Such a computation, however, complicates the analysis algorithm while yielding little practical benefit.

3 Preliminary Results and Future Work

The LTL specifications of the properties from Sect. 1 are listed in Table 3. We have introduced two auxiliary variables. We use `latch` so we need not repeat the entire landing condition in both Property 1.2 and 1.3; it also allows arbitrary time between landing and the opening of the door. With the `timer_count` variable we can capture the properties without nested temporal operators.

Table 3. Formal requirements for "Post Landing Finalize" in LTL

Requirement 1	**Property 1.1**	$\mathbf{G}\left(\begin{array}{l}\texttt{ac\_on\_ground} \wedge \texttt{mode} = \texttt{L} \wedge \texttt{door\_closed} \wedge \\ \mathbf{X}\left(\texttt{mode} = \texttt{G} \wedge \texttt{door\_closed} \wedge \texttt{ac\_on\_ground}\right) \\ \Rightarrow \mathbf{X}\left(\texttt{latch}\right)\end{array}\right)$
	Property 1.2	$\mathbf{G}\left(\neg\texttt{finalize\_event} \Rightarrow \texttt{timer\_count} \leq 4 \vee \neg\texttt{latch}\right)$
	Property 1.3	$\mathbf{G}\left(\begin{array}{l}\texttt{timer\_count} \geq 4 \wedge \neg\texttt{door\_closed} \wedge \texttt{latch} \\ \Rightarrow \texttt{finalize\_event}\end{array}\right)$
Requirement 2	**Property 2.1**	$\mathbf{G}\left(\begin{array}{l}\texttt{finalize\_event} \Rightarrow \mathbf{X}\left(\mathbf{G}\left(\neg\texttt{finalize\_event} \vee\right.\right. \\ \left.\left.\left(\texttt{latch} \wedge \texttt{timer\_count} \geq 4 \wedge \neg\texttt{door\_closed}\right)\right)\right)\end{array}\right)$
Requirement 3	**Property 4.1**	$\mathbf{G}\left(\texttt{door\_closed} \Rightarrow \mathbf{X}\left(\neg\texttt{finalize\_event}\right)\right)$
Requirement 4		

In this case, `timer_count` $= x$ is equivalent to $\mathbf{G}\_[\mathbf{0}, \mathbf{x}]$ (¬`door_closed`) because `timer_count` counts the time the door is open, and resets to 0 when the door is closed. For these properties, we used the minimum bound analysis in Sect. 2.2 to show that four time-steps are sufficient to prove any property in the model. All properties were proven in SAL using k-induction.

The assumptions in Sect. 1 implicitly underlie these proofs. As model inputs, `mode`, `ac_on_ground`, and `door_closed` are not constrained in successive values or interaction. The fact that the `finalize_event` follows the appropriate input sequence does not indicate that sequence is possible. We are working to automate the translation of those assumptions into guarantees that can be proven on the upstream components. Other future work includes feeding the computed minimum time bound directly to `Sim2SAL` to generate the timer abstraction and translate the properties to utilize the new bound, and compositional techniques for multi-rate systems.

References

1. Brat, G., Bushnell, D., Davies, M., Giannakopoulou, D., Howar, F., Kahsai, T.: Verifying the safety of a flight-critical system. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 308–324. Springer, Heidelberg (2015)
2. Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., Robinson, R., Tonetta, S.: Formal design and safety analysis of AIR6110 wheel brake system. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 518–535. Springer, Heidelberg (2015)
3. Backes, J., Cofer, D., Miller, S., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 82–96. Springer, Heidelberg (2015)
4. Li, W., Gerard, L., Shankar, N.: Design and verification for multi-rate distributed systems. In: ACM/IEEE International Conference on Formal Methods and Models for Codesign, September 2015
5. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems (1992)
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, New York, NY, USA, pp. 411–420. ACM (1999)
7. Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool chain to support automated formal verification of avionics simulink designs. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 78–92. Springer, Heidelberg (2012)