A Safety-Case Approach For Certifying Adaptive Systems

John Rushby*
SRI International, Menlo Park CA USA

Adaptive systems—those that can change their behavior at runtime—pose new challenges for certification, and particularly for traditional, *standards-based* methods of certification such as DO-178B. These traditional methods are effective in conservative fields because they can establish a solid basis in experience and can incorporate the lessons learned from previous systems. They seem likely to prove less effective in fast-moving fields where innovation outstrips the pace at which experience can be incorporated into standards. Argument-based *safety cases* offer a plausible alternative basis for certification in these fast-moving fields.

A safety case provides an explicit statement of safety *claims*, a body of *evidence* concerning the system, and an *argument*, based on the evidence, that the system satisfies its claims; standards-based methods, in contrast, specify only the evidence to be produced.

A reasonable objection to safety cases is that many arguments—especially large, complex ones—can appear plausible, yet harbor flaws. There is a need for tools that can help analyze arguments. Some model-based design tools can do this, but generally operate at a far more detailed level of design than is appropriate for much of safety analysis. Some interactive theorem provers can do it, too, but they generally require notation and skills far removed from those found in aerospace and safety engineering.

In this paper we argue that analysis tools based on recent advances in formal methods (SMT solvers, infinite bounded model checkers, and k-induction) can provide suitable modeling notations, effective analysis, and push button automation. We illustrate the approach with a simple example based on a self-checking pair.

We further argue that *monitors* derived from a safety case provide a potentially certifiable means for entering an adaptive mode of behavior, and that monitors generated from a formally analyzed case can be "possibly perfect," which is a property that allows a novel kind of reliability analysis.

I. Introduction

ADAPTIVE controls are found in many applications to improve various system qualities, including safety and performance. Despite potential benefits and increasing appeals to use these systems in next generation aircraft (both unmanned and manned), adaptive systems have not moved yet into civil aviation. Historically, new techniques seem to be adopted more slowly in civil aviation than in other industries. The adoption of adaptive techniques conforms to this pattern, at least in part because of concerns about the perceived unpredictability of adaptive systems, which poses a significant challenge for safety-based certification.

The operational behavior of conventional control systems (in the absence of errors) is fully determined and known at the time of certification; and, certification is performed on a final, complete, configuration that will not change in operation. In contrast, the operational behavior of an adaptive control system is not fully determined or known at the time of certification because the system can alter its behavior during operation in response to parameters that are not completely known in advance. Current regulatory standards do not have provisions for the assurance of system aspects that are not predictable at the time of certification. Because adaptive methods are implemented in software, we consider certification challenges for adaptive systems from a software perspective.

*Program Director, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025, Member AIAA

In the civil aviation industry, compliance with RTCA/DO-178B ("Software Considerations in Airborne System and Equipment Certification")[1] is the primary means for obtaining regulatory approval of software. For an adaptive control system to be used in commercial aircraft operating in civil airspace, the software aspects of the system would have to comply with this document, or with an equivalent alternate means of compliance. With respect to the guidance in DO-178B, adaptation presents problems especially in the software requirements and verification activities. Completely specifying requirements and expected behaviors and verifying them is essential under DO-178B. But because the requirements for fully adaptive systems are intentionally incomplete (that is, some operational behavior will be learned in flight), there is no clear means of complying with DO-178B. Training aspects of adaptation also may require additional verification beyond the review, analysis, and testing requirements in DO-178B. Consequently, one possible prospect for adoption of adaptive systems lies in developing an alternative framework for arguing the safety of those systems.

Within the Integrated Resilient Aircraft Control (IRAC) project in NASA's Aviation Safety Program, a safety-case approach is being developed for certifying software systems, including adaptive software systems. A safety case is based on construction of explicit safety claims (also called goals), evidence, and arguments, whereas traditional standards-based approaches such as DO-178B typically specify only the evidence to be produced by pre-defined processes and activities. In a safety case, the claims, evidence, and arguments for safety assurance are presented explicitly and are evaluated by the certifying authority or its designee. The exact form of the safety case is a matter for negotiation by the parties involved, but it must generally conform to a given outline.

An advantage of the safety-case approach is that it focuses on the specifics of the system under consideration, and hence can tailor the methods of assurance appropriately (for this reason, it is sometimes referred to as a goal-based approach to assurance). Safety cases can also support technical means for ensuring safety. For example, the safety case can identify critical properties to be monitored at runtime. If the properties are stated formally, then the monitors can often be synthesized automatically. In some cases, an adaptive controller can be synthesized to guarantee the property (under suitable assumptions). Safety cases are proving particularly attractive in fast-moving fields such as medical devices, where technical innovations outpace standards-based assurance methods. Software-enabled innovations are now bringing the same rapid pace of change to civil aircraft systems, and adaptive controls will increase this pace.

Although it is likely that a safety case will suggest different arguments and supporting evidence for software assurance than those that are implicit in DO-178B, the overall approach is compatible with current civil aircraft certification where, at the top level, the applicant proposes a "Means of Compliance" and a "Plan for Software Aspects of Certification"; often the latter will cite DO-178B and other guidelines, but these are not mandatory.

The upper-level evidence and arguments for aircraft safety are based on various kinds of safety analysis such as hazard analysis, failure modes and effects analysis, and fault tree analysis, and these penetrate down into subsystems and their software. DO-178B envisages that methods for safety assurance penetrate only to the upper levels of software development (requirements specifications), and that the goal of assurance for the software development process is to establish that the delivered software exactly matches (is correct with respect to) those specifications. A safety-case approach would cause safety analysis to penetrate deeper into the software design and would refocus software assurance on safety rather than correctness. By regarding adaptive software as a system that is subjected to safety analysis that examines the specific kinds of hazards this system may introduce, a safety case will focus on these hazards and their mitigations, not on "correctness" of the neural network or other learning or adaptive component.

Some adaptive software is intended to operate only when the aircraft is damaged, or in an unusual attitude, or otherwise outside the envelope of its conventional controls, and therefore potentially doomed. In these cases, a certification authority might relax its concerns about the adaptive system, but would have extreme concern about mechanisms that activate the switch to adaptive mode. There are numerous examples where well-intentioned recovery mechanisms have put a flight at risk (e.g., on 12 May 1997, anomaly detection and mitigation mechanisms reset a bus on American Airlines Flight 903, an Airbus A300, because the indicated roll rate of more than 40 degrees/second was considered implausible; in fact, the pilots were attempting recovery from a major upset and the roll rate was real; the loss of all instruments at this critical time jeopardized the recovery).

An attractive idea is that the switch to adaptive mode is triggered by a monitor driven from the safety case. A sound safety case will have identified all assumptions on which safe operation of the standard system depends, and a safe design will have ensured that these assumptions are valid (if they concern other

engineered subsystems) or trustworthy (if they concern the external environment). By monitoring these assumptions and other claims in the safety case at runtime, it may be feasible to make a strong argument that exceptional mode switches are made only in exceptional circumstances, when no conventional choice is available. The strength of this argument will depend on the precision of the assumptions and claims being monitored, and therefore on the soundness and completeness of the overall safety case and the quality and precision of its claims.

In the next section we consider how the quality and precision of a safety case can be assessed, and the feasibility of mechanized analysis of the arguments in a safety case. In Section III, we present a simple example in some detail, and provide discussion and conclusions in Section IV.

II. Assessing the Quality of a Safety Case

One of the attractions to standards-based methods of assurance such as DO-178B is that they are based on much prior experience and, in their more prescriptive interpretations, they provide strong guidance on process and procedure and thereby help establish a quality "floor." Conversely, although the argument in favor of safety cases is compelling in the abstract, it is not obvious how to assess the soundness and quality of a specific, concrete case.

There is a whole field devoted to the general concept of "argument mapping",[2] which employs visual representations of arguments. And there are graphical notations such as GSN (Goal Structuring Notation)[3] and CAE (Claims Argument Evidence)[4] that are specialized to the presentation of safety cases. Tools such as ASCE[5] support these notations and allow a safety case to be developed and explored at different levels of detail. However, these notations and tools lack automated support for tasks such as deciding whether a safety case is adequately complete, or whether its argument is sound.

It can be argued that safety cases must reason about partially unknown or unknowable contingencies and necessarily involve judgement—so that focus on formal concepts like completeness and soundness is beside the point. For this reason, some proponents of safety cases look to Toulmin[6] rather than classical logic in framing cases;[4] Toulmin stresses *justification* rather than *inference*. But even those who advocate Toulmin-style justification would surely concede that whatever arguments, or parts thereof, that can be assessed mechanically within the framework of classical logic should be so assessed, thereby preserving the precious resource of expert judgement for the circumstances that truly require it. Furthermore, human judgement and mechanized assessment are not antithetical: an argument derives conclusions from premises; mechanized assessment can focus on the soundness of that argument, while judgment is applied to plausibility of the premises.

That is to say, elements of a safety case argument can be represented in classical logic as theorems of the form

$$A_1, \ldots, A_n, S \vdash C$$

where $A_1, \ldots, A_n$ are the assumptions under which the system or design S satisfies the claim C. Mechanized tools such as theorem provers and model checkers can analyze theorems of this form and provide an inestimable benefit by helping to identify necessary assumptions. They do this be refusing to "sign off" on a proof until all necessary assumptions have been formulated appropriately and included in the statement of the theorem. Once we have made assumptions explicit, we can subject them to analysis in the same way as other claims: we can ask whether they can be substantiated by subsidiary arguments and evidence, in what circumstances might they be invalidated (cf. fault-tree analysis), and what might be the consequences if they are false (cf. failure modes and effects analysis). In some circumstances, we may choose to monitor them at runtime.

The challenge to providing mechanized support for safety case arguments is in finding a good match between the forms of expression and description that are appropriate to a safety case, and those that are tractable to automated kinds of formal analysis. In some areas of detailed system design, a good match between engineering practice and mechanized formal analysis has been found in model based design (MBD). In MBD, engineers develop designs using graphical tools such as Simulink/Stateflow, and can then explore their properties using the Simulink Design Verifier,[7] which is often able to prove certain properties of a design (or exhibit a counterexample when the property is violated). Similar capabilities are available for other MBD notations and tool chains.

The problem in applying MBD tools to safety case analysis is that their modeling notations are oriented toward simulation and are best suited to very concrete models of the kind developed during detailed design.

A safety case will generally focus on more abstract descriptions of the system and on its interactions with the environment. We will generally want to characterize the environment by stating assumptions, rather than provide a model that "implements" it, and we will want to explore hazards and very general forms of failure (as in fault tree analysis) rather than very concrete behavioral properties.

Now formal methods do provide very abstract means of expression that nonetheless support automated reasoning: examples include uninterpreted types and functions. An uninterpreted type is just some collection of values and an uninterpreted function is some $f(x)$, where all that is known is $x = y$ implies $f(x) = f(y)$ (and, dually, $f(x) \neq f(y)$ implies $x \neq y$), and $\forall x : f(x) = g(x)$ implies $f = g$. Additional properties can be specified by axioms. The difficulty with this approach is that support for mechanized reasoning is generally provided by an interactive theorem prover. As the name suggests, these generally require human guidance to develop a proof—and effective guidance may demand considerable expertise. Furthermore, the specification language supported by a theorem prover is generally based directly in logic, so that developing effective specifications also demands considerable expertise. And the expertise required for these tasks is in logic and theorem proving and formal methods, rather than in safety analysis, or in the subject domain of the system concerned.

Thus, although traditional formal methods provide methods and tools that can help in assessing arguments in a safety case, the character of those methods and tools renders them unlikely to win acceptance in engineering practice. We need methods that share some of the concepts made familiar by MBD—e.g., use of state machines—and the fully automated kinds of analysis that tools such as Design Verifier bring to MBD, but that permit more abstract levels of description. There are tools for AADL that lean in this direction (e.g., OSATE[8] and EDICT[9]) but they focus on attributes that can be attached to the flows between component. Instead, we advocate in this paper an approach based on fully automated analysis of state machines using SMT solvers.

SMT solvers are fully automated software tools for the problem of checking Satisfiability Modulo Theories;[10] what this means is that they can find satisfying instances—or prove there are none—to problems involving propositional (i.e., Boolean) combinations of terms drawn from a combination of theories that generally includes uninterpreted functions as well arithmetic, bitvectors, and arrays, records, and tuples of these. SMT solvers partake in an annual competition that has seen dramatic improvements in performance and capabilities over the last few years. Although SMT problems are NP-hard, good solvers routinely can solve problems with hundreds of thousands of terms and thousands of variables in a few seconds.

SMT solvers can provide automated analysis for abstract system descriptions specified as combinations of state machines. The simplest form of analysis checks whether a given property is true of all reachable states of the system (i.e., is an invariant). If the property is an invariant, it can often be proved to be so using k-induction (where k is a small integer) to generate an SMT problem; if the property is not an invariant, a counterexample can often be generated using k-bounded model checking to generate a different SMT problem. A tool that can generate these kinds of SMT problems is called an *infinite bounded model checker* (because it solves bounded model checking problems for systems specified over possibly infinite domains—a traditional model checker can deal only with finite state descriptions).

In the following section we illustrate this approach on a simple example.

III. Example

The purpose of this section is to illustrate use of an infinite bounded model checker to explore the argument in support of part a simple safety case. We use the modeling notation and tools of SRI's SAL system (Symbolic Analysis Laboratory).[11] SAL's analysis uses SRI's Yices tool, which is a state of the art SMT solver.[12] Both SAL and Yices are freely available from SRI.

The example considers the safety of a computer control system that employs a self-checking pair. It is part of a larger example that is very loosely based on the architecture of the Fuel Control and Monitoring Computers (FCMCs) of the Airbus A340-600 as described in a recent incident report.[13] Presentation of the example and its analysis exposes some fairly technical aspects of the SAL modeling and property languages and may look more difficult than it really is. In industrial use, we would expect all these aspects to be hidden behind a graphical modeling notation and GUI interface.

A safety case will likely be presented to an assessor as a completed argument; for pedagogical purposes, however, we prefer here to develop the case incrementally. We begin by specifying the behavior of an ideal, fault-free controller. Abstractly, a controller takes some sensor data as input, applies control laws, and sends

the resulting values to actuators. This computation is repeated in regular, cyclic fashion, with the sensor inputs that arrive in one cycle being used to compute the control outputs for the next.

In SAL, we can specify this behavior at exactly this level of abstraction.

```
sensor_data: TYPE;

actuator_data: TYPE;
init: actuator_data;

laws(x: sensor_data): actuator_data;

ideal: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  ideal_out: actuator_data
INITIALIZATION
  ideal_out = init;
TRANSITION
  ideal_out' = laws(data_in)
END;
```

This specification says that `sensor_data` is an *uninterpreted type*, as is `actuator_data`; furthermore, `laws` is an *uninterpreted function* from `sensor_data` to `actuator_data`, and `init` is an *uninterpreted constant* of type `actuator_data`.

The next part of the specification introduces `ideal` as a state machine that takes a value `data_in` of type `sensor_data` as input, and produces a value `ideal_out` of type `actuator_data` as output; this value is initialized to the value `init`. The `TRANSITION` section specifies that at each cycle, the new value of `ideal_out` (the prime symbol indicates a new value) is calculated by applying the `laws` function to the previous value of `data_in`.

This state machine is conceptually similar to those that many engineers will have encountered in MBD notations such as Simulink/Stateflow, but it is far more abstract. In MBD, the concern is to specify the exact control laws, and the data representations of the input sensor data and output actuator data. But our concern in this analysis is to examine issues in redundancy and fault coverage, so it is appropriate to treat other issues abstractly to better focus on the ones of interest: hence we abstract away how many and what types of data constitute the sensor data inputs and actuator outputs, and all details about the control laws.

Notice, however, that some issues concerning the control law calculation are represented here. For example, the `TRANSITION` specifies that outputs are based on inputs from the *previous* cycle. If outputs use inputs from the same cycle, we would have written

Illustrative aside

```
  ideal_out' = laws(data_in')
```

(observe the prime on the right hand side). This issue is significant to our analysis because it affects latency of fault detection; the particular choice made here was selected purely for pedagogical reasons.

Another design decision represented here is that the controller is stateless: that is, it is a pure function from inputs to outputs (so in fact, it is not a state machine but a combinational circuit). A controller that maintains internal state could be described as follows

Illustrative aside

```
state: TYPE;
s_init: state;

laws(s: state, x: sensor_data): control;
update(s: state, x: sensor_data): state;

modified_ideal: MODULE =
BEGIN
INPUT
  data_in: sensor_data
LOCAL
  s: state
OUTPUT
  ideal_out: actuator_data
INITIALIZATION
  s = s_init;
  ideal_out = init;
TRANSITION
  ideal_out' = laws(s, data_in);
  s' = update(s, data_in);
END;
```

where `state` is the type of the internal state, `update` is the state update function, and both `update` and `laws` are now functions of both the current state and sensor input. A real controller is likely to maintain local state; we prefer to omit it here in the interests of brevity and simplicity.

We now specify a controller that is prone to errors. Unlike the `ideal` controller, this one can exhibit two different behaviors: its normal behavior is just like the ideal case, but in the presence of a hardware fault, it can assign any value *except* the correct one to its output.

```
metasignal: TYPE = {up, down};

controller: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  control_out: actuator_data,
  errorflag: metasignal
INITIALIZATION
  control_out = init;
  errorflag = down;
TRANSITION
[
normal:
  TRUE --> control_out' = laws(data_in); errorflag' = down;
[]
hardware_fault:
  TRUE --> control_out' IN {x: actuator_data | x /= laws(data_in)}; errorflag' = up;
]
END;
```

These two different behaviors are specified by *guarded commands* in the `TRANSITION` section; the guards are Boolean expressions that precede the `-->` symbols and the commands are the sequences of assignments to their right. In each cycle, all the guards are evaluated; if any of them are true, then one of these is selected nondeterministically, and the commands on the right of the corresponding `-->` are performed atomically; if no guard evaluates to true, the module deadlocks (a special `ELSE` guard is available to preclude this if required). Here, both guards are the constant `TRUE`, so either set of commands will be selected nondeterministically in each cycle. The strings `normal:` and `hardware_fault:` are simply labels that are useful in interpreting the outputs from analysis. The first assignment in the `hardware_fault` command

employs further nondeterminism: the `IN` construction selects an arbitrary value of the type given to the left of the `|` symbol that satisfies the Boolean expression to its right; here it selects some value other than the correct one.

Real hardware faults do not announce themselves: there is no infallible signal that indicates when a component is behaving incorrectly. That is why designing fault tolerant systems is difficult—we have to make inferences from behavior that we can observe and take appropriate steps. For analysis, however, it is often useful to supply such a signal, provided it is used only for these purposes and not to endow modeled components with magical abilities. Thus, although a Boolean variable could be used as such a signal, we instead introduce the new type `metasignal` (with values `up` and `down`) to make it clear that values of this type are to be used for analysis only. Then we add an `errorflag` of this type that is set `up` whenever an incorrect value is being output.

We have now modeled enough behavior that we can run a small analysis as a sanity check. First we specify `monitor` as a module that runs the `ideal` and the error-prone `controller` as a *synchronous composition*. Inputs and outputs of the same names are automatically "wired together"; here both modules take `data_in` as input and will receive the same values. Next, we specify a property we wish to examine; the `LEMMA` (that is simply SAL's keyword to introduce a property or assertion) `check_non faulty` claims that in the `monitor` module, it is *always* the case that if the `errorflag` signal is `down`, then the output of the error-prone controller is the same as the ideal one. SAL uses *Linear Temporal Logic* (LTL) as its property language; the `G` operator of LTL means "always" (and `=>` means "implies").

```
monitor: MODULE = ideal || controller;

check_nonfaulty: LEMMA monitor |- G(errorflag = down => control_out = ideal_out);
```

We examine this claim by using the SAL infinite bounded model checker to explore it.

```
sal-inf-bmc selfcheck.sal check_nonfaulty -d 10
```

Here, `sal-inf-bmc` invokes the bounded model checker, `selfcheck.sal` is the name of the file that contains this specification, `check_nonfaulty` is the name of the assertion to be examined, and -d 10 instructs the bounded model checker to explore all possible executions of length up to 10 cycles.

The model checker reports:

```
No counterexample between depths: [0, 10].
Total execution time: 0.05 secs
```

Of course, there might be counterexamples requiring longer runs, but we have some evidence the assertion might be true, so we attempt to prove it.

```
sal-inf-bmc selfcheck.sal check_nonfaulty -i -d 1
```

The flag `-i` instructs the model checker to attempt proof by induction, and `-d 1` sets the induction depth to 1 (this is traditional induction; larger depths correspond to stronger forms of k-induction).

```
Executing k-induction with k=1
Proved.
Total execution time: 0.06 secs
```

We have confirmed that the outputs agree when there is no error; now we check that they disagree when there is an error.

```
check_faulty: LEMMA monitor |- G(errorflag = up => control_out /= ideal_out);
```

This assertion is true, and is proved by induction in the same way as the previous example.

We can combine the two previous assertions into one, which is then proved by induction in the usual way.

```
check_controller: LEMMA monitor |- G(errorflag = up <=> control_out /= ideal_out);
```

Now errorflag is an analysis artifact; it is not available in the real system, so we need to find a way of detecting errors using values that are available. A common strategy for detecting and masking hardware errors is to employ redundancy: given two controllers, we can cross-compare their outputs as a means of error detection. Such an architecture is called a *self-checking pair*.

We continue our analysis by exploring the behavior of a self-checking pair of error-prone controllers. We could do this by extending the model of the controller so that it is cognizant that it is one of a pair, or we could simply attach two standard controllers to a checker component. We follow the latter course here.

```
checker: MODULE =
BEGIN
INPUT
  con_out: actuator_data,
  mon_out: actuator_data
OUTPUT
  safe_out: actuator_data,
  fault: boolean
INITIALIZATION
 safe_out = init;
 fault = FALSE;
TRANSITION
safe_out' = con_out';
[
disagree:
  con_out' /= mon_out' --> fault' = TRUE
[]
ELSE -->
]
END;
```

The idea is that the checker takes the outputs of two controllers (we will speak of one as the controller and the other as the monitor) as its inputs, passes the first of these through as its own output, and additionally latches a Boolean fault variable to TRUE if the two values ever diverge. Unlike the controllers, the checker sets its outputs in the same cycle as it reads its inputs: this is indicated by use of primed variables on the right hand sides of assignments; we do this simply for illustrative purposes. Notice that unlike the errorflag metasignal, the Boolean fault variable is one that can be observed and used in the modeled world, not merely in analysis of that world.

We now need to "wire up" a pair of controllers and a checker,

```
cpair: MODULE = checker
 || (RENAME control_out TO con_out, errorflag TO cerror IN controller)
 || (RENAME control_out TO mon_out, errorflag TO merror IN controller);
```

This design is represented graphically as part of Figure 1 (the Figure also includes other elements that are introduced later). Components and variables that are part of the real design are in black; those that are analysis artifacts are in red.

Our goal is that the safe_out output of the checker should be trustworthy provided the fault output is not TRUE. As a first step in exploring this, we check that fault does go TRUE if either of the errorflags go up.

```
test_cpair1: LEMMA cpair |- G((cerror = up OR merror = up) => fault);
```

The SAL model checker finds that this assertion is false and generates a counterexample.

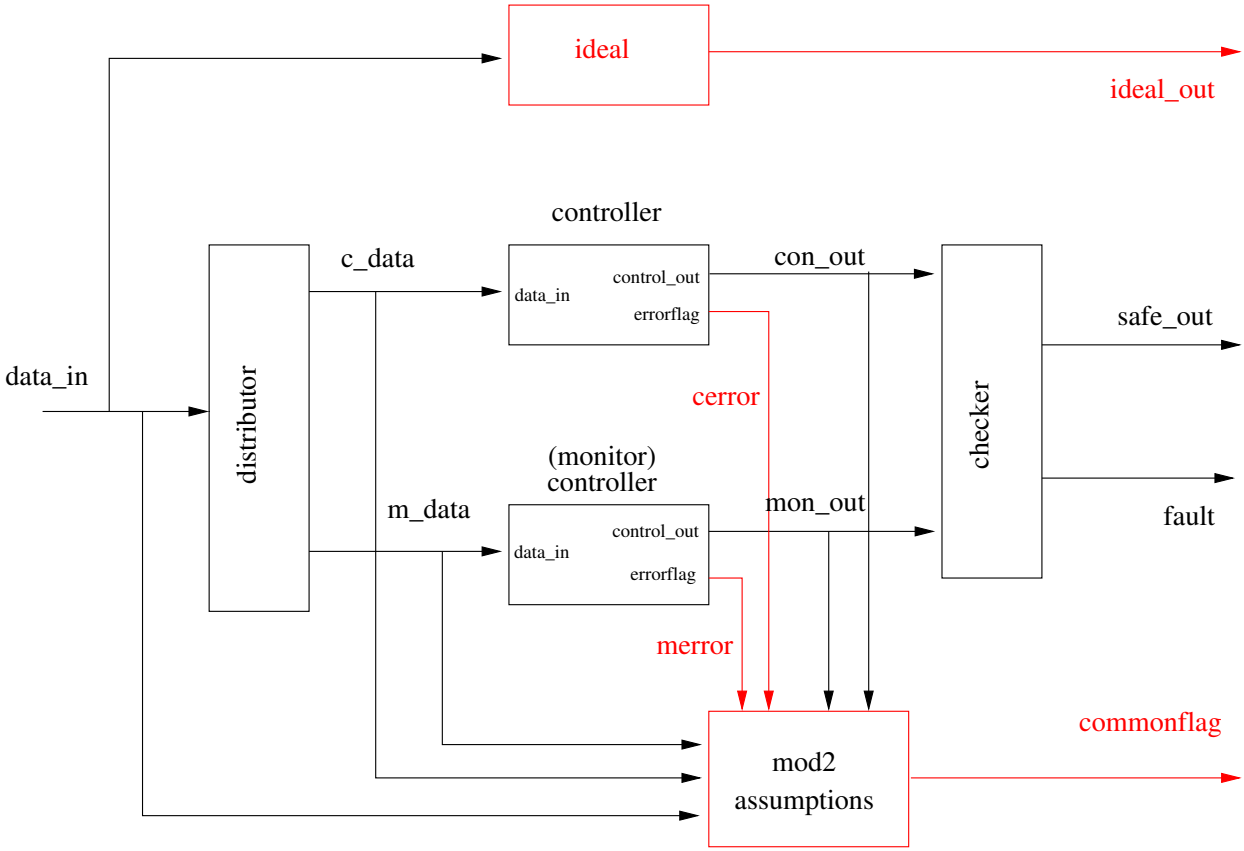

Figure 1. Graphical Representation of the Example Model (red indicates analysis artifacts)

```
Total execution time: 0.05 secs
Counterexample:
========================
Global Constraints
========================
laws(2) = actuator_data$(3);
init = actuator_data$(1);
========================
Path
========================
Step 0:
--- Input Variables (assignments) ---
--- System Variables (assignments) ---
cerror = down
fault = false
merror = down
--- Constraints ---
safe_out = actuator_data$(1);
con_out = actuator_data$(1);
mon_out = actuator_data$(1);
data_in = sensor_data$(2);
------------------------
Transition Information:
(module instance at [Context: selfcheck, line(99), column(19)]
  (((module instance at [Context: selfcheck, line(94), column(16)]
    else transition at [Context: selfcheck, line(89), column(0)])
   (module instance at [Context: selfcheck, line(95), column(58)]
     (label hardware_fault
       transition at [Context: selfcheck, line(42), column(2)])))
   (module instance at [Context: selfcheck, line(96), column(58)]
     (label hardware_fault
       transition at [Context: selfcheck, line(42), column(2)]))))
------------------------
Step 1:
--- Input Variables (assignments) ---
--- System Variables (assignments) ---
cerror = up
fault = false
merror = up
--- Constraints ---
safe_out = actuator_data$(4);
con_out = actuator_data$(4);
mon_out = actuator_data$(4);
```

We learn from the counterexample that SAL has been able to violate the assertion by causing both `controller`s to be in error, but to generate the *same* incorrect output.

What we really need to do next is to explore assumptions about "common mode" errors in both controllers. However, before doing that, we can at least make sure that the design behaves as expected when only one `controller` is in error—note how `XOR` has replaced `OR` in the following assertion.

```
test_cpair2: LEMMA cpair |- G((cerror = up XOR merror = up) => fault);
```

This generates no counterexamples, and can be proved by induction in the usual way, thereby confirming that problems arise only when both controllers are in error.

Before exploring assumptions, we might also want to make sure that the `fault` signal does indeed indicate that one of the `controller`s has suffered an error.

```
test_cpair3: LEMMA cpair |- G(fault => (cerror = up OR merror = up));
```

This generates an eight-step counterexample that is a little hard to interpret. We can cause SAL to generate the shortest counterexample by using the `-it` (iterative) flag.

```
sal-inf-bmc selfcheck.sal test_cpair3 -it
```

This generates a two-step counterexample, from which it can be seen (in the interests of space, we do not include this and later counterexamples here) that the problem is that `fault` latches, whereas the `errorflag`s do not. Thus we specified the assertion incorrectly; we need to say that if the `fault` variable is `TRUE`, then one of the `errorflag`s was `up` sometime in the past. SAL does not have past-time temporal operators, but it does have a next-state operator so we can focus on the instant where the `fault` variable goes from `FALSE` in one state to `TRUE` in the next, and assert that one of the `errorflag`s is `up` in that second state. This is specified formally using the `X` (next state) operator of LTL as follows

```
test_cpair4: LEMMA cpair |- G((NOT fault AND X(fault)) => X(cerror = up OR merror = up));
```

This assertion is true and is proved by 2-induction (we need depth two because of the `X` operators).

```
sal-inf-bmc selfcheck.sal test_cpair4 -i -d 2
```

We now turn to the topic of the assumptions under which the self-checking pair provides correct fault detection. From our examination of the assertions `test_cpair1` and `test_cpair2` we know that fault detection is correct in the presence of a single faulty `controller`, but can fail when both `controller`s suffer an error at the same time. From the counterexample to `test_cpair1` we know that one specific circumstance in which the fault detection can fail is when the erroneous `controller`s both generate the *same* bad output. But is this the only circumstance?

We would like to state the assumption that when both `controller`s are in error, they produce different outputs, then constrain examination of the design to *all and only* those executions that satisfy the assumption. If we were examining a concrete system by testing, we would need to generate all possible test runs, except those that violate the assumption; this might be feasible in this simple case, but becomes difficult or impossible when assumptions are complex. An alternative would be to generate all possible inputs, then monitor the system in execution and discard those runs where the monitor indicates violation of the assumption. For analysis by model checking, we use a technique similar to the second alternative. We specify assumptions in a monitor component that observes the behavior of the system (for this reason, such a monitor is often called a *synchronous observer*) and raises a `commonflag` variable of type `metasignal` when assumptions are violated. Then we modify our assertions so that they need hold only when `commonflag` is `down`.

For this example, we specify that the assumption is violated when both `controller`s of a self-checking pair are in error, but they send the same values to the actuators. In this case, `commonflag` is set `up`: note that it latches in the `up` position, as we wish to exclude runs in which the assumption is *ever* violated.

```
pair_assumptions: MODULE =
BEGIN
OUTPUT
  commonflag: metasignal
INPUT
  data_in, c_data, m_data: sensor_data,
  cerror, merror: metasignal,
  con_out, mon_out: actuator_data
INITIALIZATION
 commonflag = down
TRANSITION
[
assumption_violation:
  (cerror' = up AND merror' = up AND con_out' = mon_out') --> commonflag' = up;
[]
usual:
  ELSE -->
]
END;

test_cpair1_a: LEMMA cpair || pair_assumptions |-
   G(commonflag = down => ((cerror = up OR merror = up) => fault));

test_cpair4_a: LEMMA cpair || pair_assumptions |-
   G(commonflag = down => (NOT fault AND X(fault)) => X(cerror = up OR merror = up));
```

We synchronously compose a self-checking pair with its assumptions and then specify `test_cpair1_a` by modifying the assertion `test_cpair1`, to hold only when `commonflag` is `down` (i.e., the assumption is satisfied). This modified assertion is valid and is proved by induction in the usual way. We also check that the similarly modified version of `test_cpair4` remains valid (it does, and is proved by 2-induction).

We now need to consider whether our assumption is justified. On the one hand, it can be interpreted as a statement about probabilities (independently faulty components are unlikely to exhibit the same behavior), but on the other, it can be interpreted as expressing the absence of common cause errors. The latter is a strong assumption that requires further exploration and justification.

There are several ways in which the intended independence of the two `controller`s in our self-checking pair could be compromised. Some of these are outside what we can specify in our modeling framework: for example, we should require separate power supplies for each member of the pair, lest noise or incorrect voltages trigger similar errors in both of them. This and related physical sources of common cause errors must be noted and considered in the safety case separately from the modeling activity performed here. But there are some sources of common errors that can be included within our model. One of these is the distribution of sensor data to each member of the pair. In the current model, each member of the pair consumes `data_in`; this is an "open" input (meaning it is not the output of any component in the model), so it is generated by the model checker (on behalf of the "environment") and supplied to all components that use it. The SAL modeling framework "magically" provides this same data value to both `controller`s, but in reality there will be some mechanism that collects incoming sensor data and distributes it to the two `controller`s. It is often assumed that such distribution is a simple task that can give rise only to simple errors. This is not so: weak voltages in a distribution mechanism can lead to signals intermediate between the voltages that represent binary 0 and binary 1 digits; one receiver may interpret this as a binary 0 digit while another receiver interprets as a binary 1 digit. Such *slightly out of specification* (SOS) errors (which can also be manifest in the timing domain) are one source of *Byzantine errors*, which occur more often than is generally realized and which few aircraft systems can tolerate.[14]

We decide that the distribution mechanism should be included in our model, and specify it as follows.

```
distributor: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  c_data, m_data: sensor_data
INITIALIZATION
  c_data = data_in; m_data = data_in;
TRANSITION
[
distributor_ok:
  TRUE --> c_data' = data_in'; m_data' = data_in';
[]
distributor_bad:
  TRUE --> c_data' IN {x: sensor_data | TRUE};  m_data' IN {y: sensor_data | TRUE};
]
END;
```

The `distributor` takes a single `sensor_data` input and generates two outputs. In the normal case, both the outputs are identical to the input, but in the other case they are chosen at random; the choice between normal and error behavior is nondeterministic. We then specify a `full_cpair`, which adds `distributor` to `cpair` and renames the `data_in` inputs of the `controller`s to "wire them up" appropriately (see Figure 1 for a graphical representation of this configuration).

```
full_cpair: MODULE = distributor || checker
 || (RENAME control_out TO con_out, data_in TO c_data, errorflag TO cerror IN controller)
 || (RENAME control_out TO mon_out, data_in to m_data, errorflag TO merror IN controller);

test_cpair1_x: LEMMA full_cpair || pair_assumptions |-
   G(commonflag = down => ((cerror = up OR merror = up) => fault));
```

Model checking `test_cpair1_x`, which is the same as `test_cpair1_a` adjusted to this new configuration, now generates a counterexample. Examination of the counterexample reveals the problem: it is possible for the `distributor` to deliver different inputs to each `controller`, but errors in the controllers then cause them to deliver the *same* outputs from these different inputs. Of course, it is possible that the control laws are such that the same outputs are correctly generated for different inputs. But we can also require that the `checker` compare the inputs of `controller`, as well as the outputs, which would provide error coverage for this case (although we would then have to extend the `distributor` and correspondingly the errors that it can introduce). We decide to record this case as an additional assumption and add it as a new clause in the guard of the `pair_assumptions` (which we rename to `mod_assumptions`) as follows.

```
assumption_violation:
  (m_data' /= c_data' and mon_out' = con_out') OR
    (cerror' = up AND merror' = up AND con_out' = mon_out') --> commonflag' = up;
```

With this extended assumption, we can now reexamine the two assertions that we previously proved in the absence of the `distributor`: here `full_cpair` replaces `cpair` and we use the suffix `_b` rather than `_a` to name the assertions.

```
test_cpair1_b: LEMMA full_cpair || mod_assumptions |-
   G(commonflag = down => ((cerror = up OR merror = up) => fault));

test_cpair4_b: LEMMA full_cpair || mod_assumptions |-
   G(commonflag = down => (NOT fault AND X(fault)) => X(cerror = up OR merror = up));
```

Both of these are proved, by 2- and 3-induction, respectively.

These assertions merely examine the relationships between errors that are internal to the `distributor` and the `controller`s (i.e., indicated by `metasignal`s) and the explicit `fault` output of the self-checking pair. But what we really care about is that, provided it does not signal a `fault`, the self-checking pair produces the correct output for the actuators (i.e., the same output as the fault-free `ideal` controller). This assertion is specified as follows.

```
cpair_ok: LEMMA full_cpair || mod_assumptions || ideal |-
   G(commonflag = down => (NOT fault => safe_out = ideal_out));
```

Model checking this assertion generates a counterexample, whose examination indicates the need for another assumption. The problem is that the `distributor` could provide the *same, but incorrect* input to both `controller`s. We can exclude this by adding another clause to the assumption as follows, and renaming the modified module `mod2_assumptions`.

```
assumption_violation:
  (m_data' = c_data' AND m_data' /= data_in') OR
    (m_data' /= c_data' and mon_out' = con_out') OR
      (cerror' = up AND merror' = up AND con_out' = mon_out') --> commonflag' = up;
```

We are now able to prove `cpair_ok` by 1-induction.

The important point here is that we have now established that the three clauses in the guard above represent assumptions that are necessary and sufficient to ensure that the self-checking pair either gives an explicit fault indication or delivers the same output as an idealized component that suffers no errors. These are strong items of information for consideration in the safety case for a self-checking pair.

IV. Discussion and Conclusion

A safety case for a system that employs a self-checking pair would be expected to include an argument that the architecture is able to detect faults in either member of the pair and thereby prevent incorrect outputs being sent to the actuators.

An informal argument in support of the self-checking pair would employ subsidiary claims, and provide evidence in support of those claims; subsidiary claims would state that the members of the pair are sufficiently isolated, are powered from separate supplies, and so on, that their faults may be considered independent. The argument would then continue by observing that, given independence, cross-comparing the outputs of the members of the pair will allow a faulty member to be detected. On detection, the pair will shut down, or raise a fault flag, thereby preventing possibly erroneous values being sent to the actuators. The larger, surrounding case will presumably describe how this loss of function is rendered safe (perhaps by invoking yet more redundancy, or by reverting to manual control).

The details of the argument in support of the claim that cross-comparing the outputs of the two members of the pair provides adequate fault detection would include consideration of common cause errors that could vitiate the subclaim that faults are independent. The specific issues considered at this point will likely depend on the experience and skill of the developers of the system and its safety case and those of the assessors, and there has previously been no impartial, or potentially mechanizable, method for examining the soundness or completeness of the arguments and evidence adduced.

We have argued that by describing relevant elements of the design in an abstract model based design (MBD) notation, and by employing analysis tools that can consider *all possible* cases, it is possible to formalize a safety argument and to provide automated support for examination of its soundness and completeness. The traditional difficulties in applying this approach have been that MBD typically focuses on highly detailed, algorithmic aspects of design, rather than the bigger picture that is needed for many aspects of safety analysis, and its analysis tools (e.g., testing and simulation) lack the ability to consider all possible cases. Theorem proving does have the abilities to consider all possible cases and to represent high levels of abstraction, but traditionally has lacked suitable modeling notations and "push button" automation. New developments in SMT solvers, bounded model checking, and k-induction, remedy these deficiencies, and provide push button automation for analysis of state machines that employ uninterpreted types and functions for very abstract levels of specification.

We illustrated development of an abstract model for a self-checking pair using the SAL modeling notation and Yices SMT solver. Although this notation is textual, it could clearly be presented graphically in a Simulink/Stateflow-like manner. The analyses, both to generate counterexamples for false claims, and proofs for valid ones, employed "push button" automation. The example illustrated systematic discovery of three assumptions, together with demonstration that these are necessary (the absence of any one produces a counterexample) and sufficient to prove that the self-checking pair correctly raises its fault flag in the presence of errors, and exhibits the same behavior as an idealized fault-free component otherwise.

The three assumptions are:

1. When both members of the pair are faulty, their outputs differ.
2. When the members of the pair receive different inputs, their outputs differ.
3. When both members of the pair receive the same input, it is the correct input.

These assumptions provide direction for other parts of the safety case. The first demands exploration of common cause errors, and also a probabilistic assessment of the likelihood of independent errors producing the same behavior. The second can be addressed by including the inputs in the data that is cross-compared, while the third requires consideration of the fault behavior of the mechanism that distributes sensor data to both members of the pair.

We hope this example illustrates the potential value of the approach that we propose. It is applicable to safety cases in general, but may be particularly useful in cases where novel behavior—e.g., adaptive control—is triggered by monitors that are derived from the safety case, as described in the next section.

A. Application to Adaptive Systems

Adaptive systems—those that can change their behavior at runtime—pose new challenges for certification and particularly for traditional, standards-based methods of certification. These traditional methods are effective in slow-moving fields because they can establish a solid basis in experience and incorporate the lessons learned from previous systems. They seem likely to prove less effective in fast-moving fields where innovation outstrips the pace at which experience can be incorporated into standards. Argument-based safety cases offer a plausible alternative basis for certification in these fast-moving fields.

We have outlined the general case for argument-based safety cases in earlier papers,[15] but adaptive systems pose special challenges and opportunities. Passenger aircraft with safety-critical functions that operate adaptively all the time seem especially challenging to certify, so there is likely to be a discrete switch to adaptive mode and this will be employed only when conventional controls are unable to cope. The trustworthiness of the mode switch from normal to adaptive behavior is therefore particularly critical. One attractive idea is for the mode switch to be triggered by monitoring the runtime behavior of the system against its safety case.[16,17]

It is plausible to make a case that such monitoring is "*possibly perfect*" and to estimate its probability of perfection. A perfect system is one that never fails; to say that a system has probability 0.999 of perfection means that of 1,000 systems engineered in a similar way, only 1 may be expected ever to suffer a failure. This can be contrasted with a system whose *reliability* is 0.999, which means this specific system may be expected to suffer 1 failure in 1,000 demands. In recent work by Littlewood and Rushby,[18] we show that in suitable architectures, reliability of one channel and possible perfection of another are conditionally independent; hence one channel that is 0.999 reliable and another that has 0.999 probability of perfection can together yield a system that 0.999999 reliable. However, our analysis shows that in architectures where a monitor can trigger alternative (e.g., adaptive) behavior, perfection of the monitor is a dominant factor. Since the monitor is derived from the safety case, we need unusually high confidence in the case. The approach outlined here can contribute to that confidence. Furthermore, the formal methods employed in the approach render it feasible that monitors could be derived directly from the formal statements of assumptions that are generated in applying the approach.

B. Conclusion

We have described how argument-based safety cases could provide a framework in which to certify novel systems, such as those that employ adaptive controls. We focused on the circumstance where a switch to adaptive mode is triggered by runtime monitoring of assertions derived from the safety case. This circumstance demands strong assurance that the assumptions and other claims of the safety case, and its supporting argument, are sound and complete. We argued that modern formal methods employing SMT solvers can provide this assurance and we illustrated the approach with a simple example.

This example is part of a larger case study under development to examine the architecture of the fuel control and monitoring computers of the A340-600, based on the report of an incident in February 2005.[13] In future work, we plan to examine an explicitly adaptive system.

Acknowledgments

Discussions with Kelly Hayhurst of NASA Langley Research Center, and funding through NASA Cooperative Agreements NNX08AC64A and NNX08AY53A are gratefully acknowledged. I also appreciate enlightening interactions on these topics with Robin Bloomfield and Bev Littlewood of City University, and discussions on the example with Rance DeLong.

References

[1]Requirements and Technical Concepts for Aviation, Washington, DC, *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec. 1992, This document is known as EUROCAE ED-12B in Europe.

[2]*Argument Map Wiki page*, `http://en.wikipedia.org/wiki/Argument_map`.

[3]Kelly, T. P. and Weaver, R. A., "The Goal Structuring Notation—A Safety Argument Notation," *DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, Florence, Italy, July 2004.

[4]Bishop, P., Bloomfield, R., and Guerra, S., "The Future of Goal-Based Assurance Cases," *DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, Florence, Italy, July 2004.

[5]*ASCE home page*, `http://www.adelard.com/web/hnav/ASCE/index.html`.

[6]Toulmin, S. E., *The Uses of Argument*, Cambridge University Press, 2003, Updated edition (the original is dated 1958).

[7]*Simulink Design Verifier home page*, `http://www.mathworks.com/sldesignverifier/`.

[8]*AADL home page*, `http://www.aadl.info/`.

[9]WW Technology Group, "EDICT Tool Suite," Available at `http://wwtechnology.com/products/EdictCore.htm`, March 2008.

[10]Rushby, J., "Harnessing Disruptive Innovation in Formal Verification," *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, edited by D. V. Hung and P. Pandya, IEEE Computer Society, Pune, India, Sept. 2006, pp. 21–28.

[11]*SAL home page*, `http://sal.csl.sri.com/`.

[12]*Yices home page*, `http://yices.csl.sri.com/`.

[13]"Report on the incident to Airbus A340-642, registration G-VATL en-route from Hong Kong to London Heathrow on 8 February 2005," Report 4/2007, UK Air Investigations Branch, 2007, Available at `http://www.aaib.gov.uk/publications/formal_reports/4_2007_g_vatl.cfm`.

[14]Driscoll, K., Hall, B., Sivencrona, H., and Zumsteg, P., "Byzantine Fault Tolerance, from Theory to Reality," SafeComp *2003: Proceedings of the 22nd International Conference on Computer Safety, Reliability, and Security*, edited by S. Anderson, M. Felici, and B. Littlewood, No. 2788 in Lecture Notes in Computer Science, Springer-Verlag, Edinburgh, Scotland, Sept. 2003, pp. 235–248.

[15]Rushby, J., "Just-in-Time Certification," *12th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS)*, IEEE Computer Society, Auckland, New Zealand, July 2007, pp. 15–24.

[16]Rushby, J., "How Do We Certify For The Unexpected?" *AIAA Guidance, Navigation and Control Conference*, American Institute of Aeronautics and Astronautics, Honolulu HI, Aug. 2008, AIAA paper 2008-6799.

[17]Rushby, J., "Runtime Certification," *Eighth Workshop on Runtime Verification: RV08*, edited by M. Leucker, Vol. 5289 of *Lecture Notes in Computer Science*, Springer-Verlag, Budapest, Hungary, April 2008, pp. 21–35.

[18]Littlewood, B. and Rushby, J., *Reasoning about the Reliability of Fault-Tolerant Systems in Which One Component is "Possibly Perfect"*, City University UK and SRI International USA, 2009, In preparation.