Integration of Formal Analysis into a Model-Based Software Development Process

Michael Whalen[1], Darren Cofer[1], Steven Miller[1], Bruce H. Krogh[2], Walter Storm[3]

[1] Rockwell Collins Inc., Advanced Technology Center
400 Collins Rd, Cedar Rapids, IA 52498
[2] Carnegie Mellon University, Dept. of Electical & Computer Engineering
5000 Forbes Ave., Pittsburgh, PA 15123
[3] Lockheed Martin Aeronautics Company, Flight Control Advanced Development
P. O. Box 748, Ft. Worth, TX 76101
{mwwhalen, ddcofer, spmiller}@rockwellcollins.com, krogh@ece.cmu.edu,
walter.a.storm@lmco.com

Abstract. The next generation of military aerospace systems will include advanced control systems whose size and complexity will challenge current verification and validation approaches. The recent adoption by the aerospace industry of model-based development tools such as Simulink® and SCADE Suite™ is removing barriers to the use of formal methods for the verification of critical avionics software. Formal methods use mathematics to *prove* that software design models meet their requirements, and so can greatly increase confidence in the safety and correctness of software. Recent advances in formal analysis tools have made it practical to formally verify important properties of these models to ensure that design defects are identified and corrected early in the lifecycle. This paper describes how formal analysis tools can be inserted into a model-based development process to decrease costs and increase quality of critical avionics software.

Keywords: Model checking, Model-based development, Flight control, software verification

1 Introduction

Emerging military aerospace system operational goals will require advanced safety-critical control systems with more demanding requirements and novel system architectures, software algorithms, and hardware implementations. These emerging control systems will significantly challenge current verification tools, methods, and processes. Ultimately, transition of advanced control systems to operational military systems will be possible only when there are affordable V&V strategies that reduce costs and compress schedules. The AFRL VVIACS program documented these challenges in detail [1].

Current software validation and verification for critical systems centers on testing of English-language requirements. While testing is currently the only way to examine

the behavior of a system in its final operational environment, it is incomplete and resource intensive. The incompleteness of testing is due to the extremely large *state space* of even small control systems.

To illustrate, the number of possible states of a program with ten 32-bit integers is 10^{96}, which exceeds the number of atoms in the universe (around 10^{80}). To exhaustively test such systems is clearly impractical. Extremely large numbers of tests must be run to gain confidence in the correctness of programs, and these test suites are still insufficient to determine whether or not a system meets its requirements.

Further complicating the issue is that the requirements for the system are usually specified in English. It is often the case that these requirements are ambiguous, incomplete, and inconsistent, meaning that developers may legitimately disagree as to whether the system meets its requirements, or even that it is not possible to implement a program that meets all of the requirements.

While the benefits of formal methods have been understood for over twenty years, their use has been hampered by the lack of specification languages acceptable to practicing engineers and the level of expertise required to effectively use formal verification tools such as theorem provers. Over the last few years these hurdles have been greatly reduced by two trends: 1) the growing adoption of model-based development for safety-critical systems; and 2) the development of powerful verification tools that are easier for practicing engineers to use. The result will be a revolution in how safety-critical software is developed.

Lockheed Martin, Rockwell Collins, and Carnegie Mellon University are working together under AFRL's Certification Technologies for Advanced Flight Critical Systems (CerTA FCS) program. Our team is tasked with determining the applicability of formal methods to avionics verification concerns for next-generation control systems. Rockwell Collins has built a set of tools that translate Simulink models into the languages of several formal analysis tools, allowing "push button" analysis of Simulink models using model checkers and theorem provers. The project is split into two phases which analyze *finite* and *infinite* state models, respectively.

This paper describes the process used and the results obtained in the first phase of the project, in which we successfully and cost-effectively analyzed large finite-state subsystems within a prototype UAV controller modeled in Simulink. During the analysis, over 60 formal properties were verified and 10 model errors and 2 requirements errors were found in relatively mature models. These results are similar to previous applications of this technology on large avionics models at Rockwell Collins [2][3][10].

To use formal methods most effectively, some changes must be made to the traditional development cycle, and formal analysis should be considered when creating requirements and designing models. This paper focuses on processes and techniques for using formal methods effectively within the design cycle for critical avionics applications.

2 Formal Methods in a Model-Based Development Process

Model-Based Development (MBD) refers to the use of domain-specific modeling notations such as Simulink or SCADE that can be analyzed for desired behavior before a digital system is built. The use of such modeling languages allows a system engineer to create a model of the desired system early in the lifecycle that can be executed on the desktop, analyzed for desired behaviors, and then used to automatically generate code and test cases. Also known as *correct-by-construction* development, the emphasis in model-based development is to focus the engineering effort on the early lifecycle activities of modeling, simulation, and analysis, and to automate the late life-cycle activities of coding and testing. This reduces development costs by finding defects early in the lifecycle, avoiding rework that is necessary when errors are discovered during integration testing, and by automating coding and the creation of test cases. In this way, model-based development significantly reduces costs while also improving quality.

Formal methods may be applied in a MBD process to prevent and eliminate requirements, design and code errors, and should be viewed as complementary to testing. While testing shows that functional requirements are satisfied for specific input sequences and detects some errors, formal methods can be used to increase confidence that a system will *always* comply with particular requirements when specific conditions hold. Informally we can say that testing shows that the software *does* work for *certain test cases* and formal, analytical methods show that it *should* work for *all* cases. It follows that some verification objectives may be better met by formal, analytical means and others might be better met by testing.

Although formal methods have significant technical advantages over testing for software verification, their use has been limited in industry. The additional cost and effort of creating and reasoning about formal models in a traditional development process has been a significant barrier. Manually creating models solely for the purpose of formal analysis is labor intensive, requires significant knowledge of formal methods notations, and requires that models and code be kept tightly synchronized to justify the results of the analysis.

The value proposition for formal methods changes dramatically with the introduction of MBD and the use of completely automated analysis tools. Many of the notations in MBD have straightforward formal semantics. This means that it is possible to use models written in these languages as the basis for formal analysis, removing the incremental cost for constructing verification models. Also, model checkers are now sufficiently powerful to allow "push-button" analysis of interesting properties over large models, removing the manual analysis cost. If a property is violated, the model checker generates a counterexample, which is simply a test case that shows a scenario that violates the property. The counterexamples generated by model checkers are often better for localizing and correcting failures than discovering failures from testing and simulation because they tend to be very short (under 10 input steps) and tailored towards the specific requirement in question.

The Rockwell Collins translation framework is illustrated in Figure 1. Under a five year project sponsored in part by the NASA Langley Research Center, Rockwell Collins developed highly optimizing translators from MATLAB Simulink and SCADE Suite™ models to a variety of implicit state model checkers and theorem

provers. These automated tools allow us to quickly and easily generate models for verification directly from the design models produced by the MBD process. The counterexamples generated by model checking tools can be translated back to the MBD environment for simulation. This tool infrastructure provides the means for integration of formal methods directly and efficiently into the MBD process.

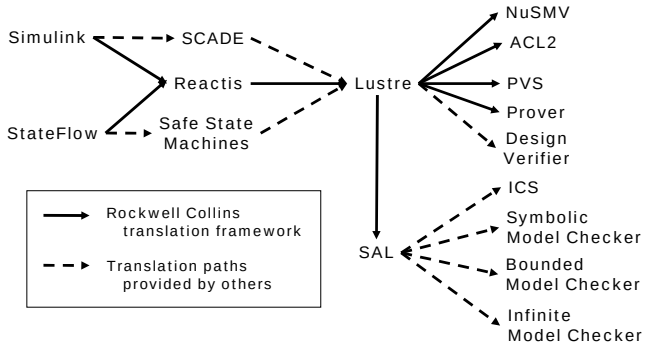

Fig. 1. Rockwell Collins model translation framework.

There are at least two different ways that model checking can be integrated into a MBD process. First, it can be performed as part of the traditional verification process in a traditional waterfall model in addition to testing. This was the approach used in the first phase of the CerTA FCS project. In this approach, the model checker simply provides a significantly more rigorous verification step to ensure that the model works as intended. However, if this step is performed late in the development cycle, much of the benefit of early detection and quick removal of defects is lost.

A better approach for integrating model checking technology is to include formal analysis as an extension of a spiral development process. In an MBD process, it is common during the model design phase to use simulation as a "sanity check" to make sure that the model is performing as intended with respect to some system requirements of interest. When performed at the subsystem level, model checking allows a much more rigorous analysis based directly on the requirements of the system. If the subsystem requirements have been captured as "shall" statements, it is usually the case that these statements can be easily re-written as formal properties. Although model checking is a rigorous application of formal methods, for many kinds of models it does not require a significant amount of manual effort.

The spiral approach was used in a previous effort during the model development process for a complex cockpit displays application [2]. After each modification of the design, Simulink models were re-analyzed against a large set of requirements in a matter of minutes. By the end of the project, the model had been proven correct against all of their requirements (573 formal properties) and 98 errors had been corrected.

The guidance in this paper focuses on the use of *implicit state model checkers*, because this is the most mature of the "push-button" analysis tools, and these tools were the focus of Phase I of the CerTA FCS project. In order to reap the maximum

benefit of formal analysis, models must be designed for analysis, much as they are designed for autocode or test case generation in current processes. The rest of this section provides guidelines for determining whether implicit state model checking is an appropriate technique for the model being constructed, and for using model checking successfully within the development process.

Implicit state model checkers are designed to analyze models with discrete variables that have relatively small domains: Boolean and enumerated types, or relatively small subranges of integers. The performance of the tools is primarily determined by four things: 1) the number of inputs to the model, 2) the number of latches (delays) in the model, 3) the size of each variable (number of bits), and 4) the complexity of the assignment equations for the variables. Implicit state model checkers do not have the ability to analyze models with real or floating point variables.

There are four primary questions in determining the applicability of implicit state model checkers in an MBD process.

- Does tool support exist (or can it be created) to automatically translate the MBD notation to the notation of the analysis tool? A handful of tools have model checking support built into the tool (e.g., Esterel Technologies SCADE, i-Logix StateMate), and several more academic and commercial projects support translation into analysis tools from Simulink and Stateflow.
- If the model contains large-domain integers or floating point numbers, can these be abstracted or restructured away from the "core" of the model? Implicit state model checkers cannot reason about floating point numbers, and do not scale well with large-domain integers. However, it is often the case that there is a complex mode logic "core" that can be analyzed separately via model checking, while the surrounding code that manages the floating point or large-domain integers can be analyzed using other means.
- Can the model be partitioned into subsystems that have intrinsically interesting properties and that are of reasonable size? Model checking has been shown to be very effective at verification and validation of large software models in a model-development process. However, there are scalability limits for implicit state tools that limit the size of models that can be analyzed effectively. In Section 5, we describe strategies for structuring requirements such that requirements over the entire model are entailed by simpler obligations over subsystems within the model.
- Can the requirements be formalized? Traditional English requirements documents are often well-suited to formalization [3], so this may not be a significant a barrier to use. Also, designers tend to have an intuitive notion of the expected behavior of a subsystem, and when formalized, these properties can form excellent documentation about the behavior of a model.

If the answers to each of these questions is 'yes', then implicit state model checking is an efficient and low-cost approach for analyzing the behavior of models.

3 Changes to the Verification Process

In our experience, the introduction of model checking changes the nature of the verification process. Instead of focusing on the creation of test vectors, the focus is on the creation of *properties* and *environmental assumptions*. The properties are translations of natural language requirements into a formal notation, and the environmental assumptions are constraints on the inputs of the model that describe the intended operating environment for the model.

Figure 2 illustrates the difference between a test-based process and analysis-based verfication. In a test-based verification process, test cases must be developed for each requirement. Each test case defines a combination of input values (a test vector) or a sequence of inputs (a test sequence) that specifies the operating condition(s) under which the requirement must hold. The test case must also define the output to be produced by the system under test in response to the input test sequence.

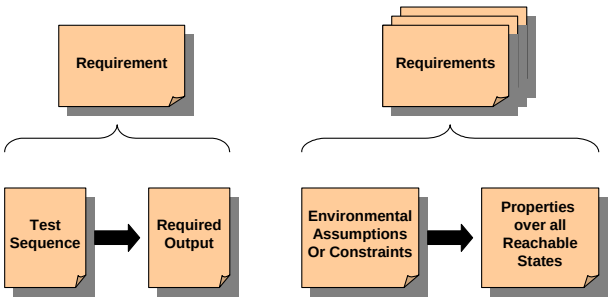

Fig. 2. Test-based verification (left) vs. Analysis-based verification (right).

An analysis-based verification process may be thought of in the same way. We normally consider a group of requirements, with related functionality for a particular subsystem. The environmental assumptions or constraints specify the operating conditions under which the requirements must hold. The properties define subsystem behaviors (values of outputs or state variables) that must hold for all system states reachable under the specified environmental assumptions.

The essential difference is one of precision: model checking requires the specification of *exactly* what is meant by specific requirements and determines *all* possible violations of those requirements at the subsystem level. This precision can be challenging, because an engineer is no longer allowed to rely on an intuitive understanding to create test vectors. Also, in some cases, the notation used for properties (such as CTL and LTL [4]) can be confusing, though there are a variety of notations (including the MBD languages themselves!) that can be used to mitigate this difficulty. Also, *precise* is not the same as *correct.* If a property is incorrectly written, then obviously a formal analysis tool may be unable to uncover incorrect behavior within a model. Therefore, it is very important that properties are carefully written and reviewed to ensure that they match the intuitive understanding of the requirement.

The fact that a model checker generates a counterexample from the set of *all* possible violations of a property often leads to 'nonsensical' counterexamples in which the model inputs change in ways that would be impossible in the real environment. In order to remove these counterexamples that will not occur in the real system, it is sometimes necessary to describe environmental constraints that describe how the inputs to the model are allowed to evolve. On the bright side, these constraints serve as a precise description of the environmental assumptions required by the component to meet its requirements.

We next describe specific changes to the verification process to facilitate the use of model checking tools.

Creating Formalizable Requirements

There are many different notations and tools used for capturing requirements in the avionics domain. These notations include traditional structured English "shall" statements, use cases, SCR specifications [5], CoRE documents [6], and others. Most avionics systems still use "shall" statements as the basis of their requirements. In our experience, shall statements are actually a good starting place for creating formalized requirements. Their prevalence indicates they are a natural and intuitive way for designers to put their first thoughts on paper.

The problem with shall statements has been that inconsistencies, incompleteness, and ambiguities are not found until the later phases of the project. The process of formalizing the requirements into properties helps remove the problem of ambiguity. When formalizing a property, by necessity, one must write an unambiguous statement. The issue then becomes whether the formalization matches the intention of the original English requirement.

Inconsistencies can be detected in several ways. First, if all requirements are formalized, then it is not possible to simultaneously prove all properties over a model if the set of properties are inconsistent. With additional translation support, it is also possible to query a model checker to determine whether *any* model can satisfy all of the properties simultaneously. There are also current research projects to define metrics for requirements completeness over a given formal model using model checking tools [7], but this research is not yet usable on an industrial scale.

Testable requirements are also analyzable, so this is a good starting point for determining whether requirements are suitable for analysis. On the other hand, there are classes of requirements that are not testable but are, in fact, analyzable. For example, requirements such as:

- the system shall *never* allow behavior *x*,
- given *y,* the system shall *always eventually* do *z*

can be analyzed formally, but are not suitable for testing as they require an unbounded number of test cases.

Other system requirement techniques such as use cases are also possible sources of properties. While more structured than shall statements, as practiced today use cases normally lack a precise formal semantics and suffer from the same problems of

inconsistency, incompleteness, and ambiguity as shall statements. While not part of this experiment, it seems reasonable that it should be possible to express use cases as a sequence of properties describing how the system responds to its stimuli, and to verify these sequences through simulation and formal analysis. In this way, the consistency and completeness of use cases could be improved in the same manner as was done for shall statements.

Creating Environmental Assumptions

One significant change when moving from a testing-based verification process to a formal process is that much more attention must be focused on environmental assumptions for the system being analyzed. Often, there are a significant number of environmental assumptions that are built into the design of the control software that cause it to fail when those assumptions are violated, and these assumptions are often not well documented. In testing, it is usually the case that the tester has an intuitive understanding of the system under test and is unlikely to create test scenarios where the plane is "flying upside-down and backwards". The model checker, on the other hand, will often find requirements violations that occur under such scenarios if environmental constraints that rule out impossible conditions are not stated explicitly.

It is often not possible to verify interesting safety properties on a large model in a completely unconstrained environment. As part of the analysis process, we examine the environmental assumptions in the requirements document to create constraints on the possible values of inputs into the system. Each of the model checking tools that we have examined supports *invariants* that allow engineers to specify constraints on the behavior of the environment. Here, "environment" means any inputs or parameters that can affect the behavior of the model being verified, and invariants are restrictions on these environmental variables. These invariants should be as simple as possible so as to not impact unduly the efficiency of the verification algorithm, but they must be sufficiently complex to assure that the specification is being evaluated for the relevant conditions. For example, for specifications for a controller model that are related to the closed-loop behavior of the system, the appropriate invariant may require the creation of a "plant model" representing a reactive environment that responds dynamically to the controller outputs.

Although invariants are necessary to prove "interesting" properties over subsystems, they are also dangerous to the soundness and applicability of the analysis. If conflicting invariants are specified, then there are *no states that satisfy the invariants*, so all properties are trivially true. Similarly, if invariants restrict the set of allowed inputs so that it is a subset of the possible inputs to the real system, then our analysis will be incomplete. Finally, just because constraints are specified in the requirements document does not mean that the environment, which can include other subsystems, will actually obey these constraints.

Therefore, although we formalize the invariants in this step *we do not use them in our initial model checking analysis*. If the initial subsystem analyses return counterexamples, we analyze the counterexamples to see whether they are due to violations of our invariants or due to incorrect behavior within the model. Even if counterexamples are due to invariant violations, we prefer to strengthen the model

behavior, when possible, to deal with abnormal environments rather than use system invariants. If it is determined that there is no good way to handle abnormal environments within the model, then we finally begin to use the invariants derived from the environmental assumptions.

It is worth noting that such environmental assumptions were precisely the cause of the Arianne V disaster [8], when an assumption about the lateral velocity of the rocket shortly after liftoff was violated when the control software was reused from the Arianne IV, causing it to fail catastrophically. By requiring developers to make their assumptions about the operating environment explicit and precise, a formal analysis process can help to eliminate this type of error.

Interpreting Counterexamples

One of the benefits of using a model checker in the verification process is the generation of counterexamples that illustrate how a property has been violated. However, for large systems it can be difficult and time consuming to determine the root cause of the violation by examining only the model checker output. Instead, the simulation capabilities of the MBD tools should be utilized to allow playback of a counterexample.

Both Simulink and SCADE have sophisticated simulation capabilities that allow single-step playback of tests and easy "drill down/drill up" through the structure of the model. These capabilities can be used to quickly localize the cause of failure for a counterexample. Third-party tools such as Reactis [11] for Simulink also allow a "step back" function so that it is possible to rewind and step through a sequence of steps within a counterexample, adding to the explanatory power of the tool.

When a counterexample is discovered, it is classified by its underlying cause and appropriate corrective action taken. The cause may be one or more of the following:

- Modeling error
- Property formalization error
- Incorrect/missing invariants for the subsystem
- High-Level requirements error

4 Changes to the Modeling Process

Flight control models, such as the Lockheed Martin operational flight program (OFP) model analyzed in our CerTA FCS project, are too large to be efficiently analyzed by current model checkers. There are several development practices that should be adopted within a MBD process to create models that are suitable for analysis. These practices will yield models that will be simpler to analyze.

Partitioning the System

The first step in analyzing the model is to divide the requirements and model into subsystems that can be automatically analyzed. *Analysis partitions* are created by splitting the original model into different subsystems and assigning a set of system requirements that will be analyzed on the subsystem (Figure 3). After the subsystems have been created, each subsystem is separately analyzed. The result of the analysis process may require changes to the subsystem under analysis, to another subsystem, or to the system-level requirements or environmental assumptions.

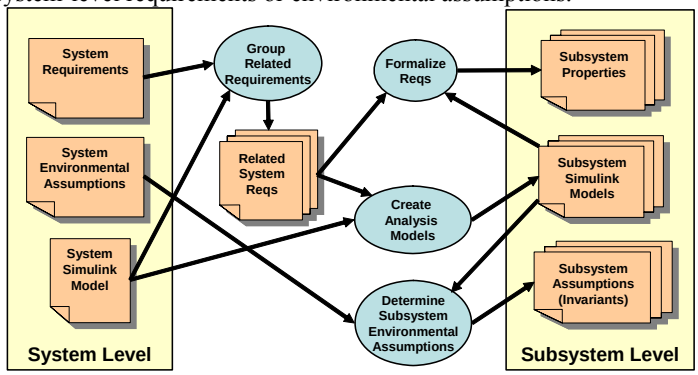

Fig. 3. Process for creating analysis partitions.

There are several steps necessary to create the analysis partitions.

Group Related Requirements. To create analysis partitions, we first try to group system requirements into sets that can be checked against a portion of the system Simulink model. In our experience with the WM and the FCS 5000 [3], it is usually the case that the properties naturally partition into sets that are functionally related to one another, and that the truth or falsehood of these property sets can be determined by examining a relatively small portion of the entire Simulink model.

Create Analysis Models. After grouping the properties, we split the system model into reasonably-sized analysis models that are sufficient to check one or more of the requirements groups. We would like to make each subsystem small enough that it is quick to analyze using our BDD-based model checking tools.

Formalizing Requirements. The next step in analyzing the model involves formalizing the functional and safety requirements as properties. For a synchronous system where the requirements are specified as "shall" statements over system inputs and outputs, this process is often straightforward. In [2], [3], and [10], we described the process of translating these informal statements into safety properties in more detail.

The system requirements document is not the only source of properties to be analyzed. Properties also emerge from discussions with developers about the functionality of different subsystems, or even from a careful review of a particular implementation detail of the Simulink model. In some cases, these properties can be

thought of as validity checks for particular implementation choices, but on occasion they lead to additions to the system requirements document.

Using Libraries

The construction of analysis partitions can be simplified by splitting the original model into *libraries*. Both Simulink and SCADE support packaging of subsystems into libraries, which are really just additional "source" files for the model. Just as it makes sense to construct a large C program using several source files (for various reasons, including version control), it makes sense to construct models using library files.

If a Simulink or SCADE model is created from a set of libraries, it is possible to generate the analysis models with very little effort. A benefit of this approach is that the subsystems within the libraries can evolve without requiring changes to the analysis models, as long as the subsystem interfaces remain stable. Therefore, once the analysis models are created, they can be used for regression testing without any additional effort.

Using Supported Blocks

Most MBD environments were originally constructed for the purpose of modeling and simulation, or for autogeneration of source code, and not for design analysis. It is usually necessary to restrict the use of certain constructs within a MBD language that complicate the semantics of the language, or that have potentially undefined behavior outside of the simulation environment. Some languages, such as SCADE, were built for formal analysis, and so almost all features of the SCADE environment (i.e., all aspects that do not involve use of a 'host' language, such as C, to implement functionality) can be formally analyzed. Simulink contains an extremely wide range of block sets with varying levels of formality. None of the current model checking tools for Simulink/StateFlow support all of the block sets that can be used within the language.

The Rockwell Collins translation tools support a wide range of Simulink/StateFlow constructs. This toolset is tailored for critical avionics software, and is able to analyze all of the blocks used in the OFP model.

Structuring for Analysis

Design choices that lead to code-bloat or poorly cohesive systems also affect the performance of the model checker. A rule of thumb is that the larger the number of blocks within a model, the longer it will require to analyze. Therefore, model re-factoring is not only a useful design activity, but often necessary to successfully analyze large subsystem models.

In our experience, we have re-factored models in which some piece of functionality (e.g., display application placement) is replicated (e.g., left-side and

right-side display application placement) by "copy and paste reuse". By properly packaging the functionality into subsystems, we can split the analysis task into independent parts, leading to much faster analysis.

Similarly, when creating the analysis models, it is possible to indirectly analyze subsystem coupling by examining the complexity of invariants between the outputs of one subsystem and the inputs of another subsystem. If complex invariants are required to prove properties about a subsystem, then it is likely that the subsystem is tightly coupled to the subsystem that generates the outputs. These cases should be examined to determine if it is possible to re-factor the design to simplify the analysis invariants.

Structuring for Predicate Abstraction

If models contain several large-domain integers and/or real numbers, they will not be analyzable by current tools. However, it is often the case that these variables can be factored out of modules that contain the complex behavior that would benefit most from formal analysis. The idea is to either abstract the conditions that involve numeric constraints or the ranges of the constants and variables involved in the conditions.

Subsystems that compute system modes often contain a handful of large-domain integers that are used for comparisons in conditions within the mode computation, e.g., *Altitude > PreSelectAlt + AltCapBias.* If the ranges of these integers are large, e.g., zero to 50000 feet, analysis may become intractable, even though they only influence a few conditions within the logic. In this case, it is much simpler for formal analysis if the original comparisons in the mode logic are replaced with Boolean inputs representing the result of the comparison (e.g., *Altitude_Gt_PreSelect_Plus_AltCapBias)*. This input is then computed by an external subsystem which can be separately (and usually trivially) checked for correctness. This kind of model factoring is called *predicate abstraction* [9], and can reduce the analysis time required from hours to seconds in the original subsystem.

If the model contains a significant number of variables and the constraints involving those variables are related, or if it uses the variables to compute numeric outputs, predicate abstraction is less useful. In these cases, it is often possible to perform domain reductions in order to scale the ranges so as to be able to analyze the models successfully.

Reducing State Through Type Replacement

A primary limiting factor when using the model checker is the size of the state space. In this section, we describe strategies to reduce the size of the model state space in order to apply implicit state model checking technology.

Using Generic Types. The implicit state model-checking tools that we use are unable to reason about real numbers. Fortunately, it is often the case that the interesting safety-related behavior is preserved by replacing real-valued variables by integers for the purpose of analysis [9]. We have used a simplified version of

predicate abstraction, which attempts to reduce the domain of a variable while preserving the interesting traces of the system behavior, i.e., the ones that can lead to a counterexample. The idea is to preserve enough values such that all conditions involving real numbers will be completely exercised.

From a design-for-analysis perspective, both Simulink and SCADE support a notion of generic types that allow models to be constructed that can use either integers or reals. The only place where the types must be specified is at the "top-level" inputs. If models are constructed using library blocks, then very little effort is required to derive analysis models from the original models.

Limiting Integer Ranges. To efficiently model-check a specification, we would like to determine the minimal range necessary to represent the behavior of each variable in the model. This is because the performance of BDD-based model checkers is directly correlated to the ranges of the variables in the model. The Rockwell Collins translation tools currently allow a high degree of control over the integer range of each variable within the model. It is possible for the user to specify both the default range of all integer variables within the model, and also to set the ranges for individual variables within the model. This allows us to trim unreachable values of variables and reduce the system state space. If we inadvertently eliminate a reachable value, the model checker will detect this and the variable range can be corrected.

5 Analysis Results

In this section, we discuss the application of the process described here to the analysis of finite-state models from the Lockheed Martin OFP Simulink model. In this analysis we focused on the Redundancy Manager (RM) component of the OFP.

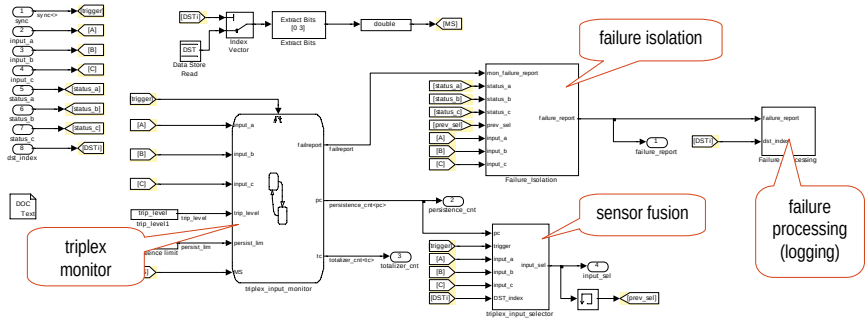

Fig. 4. Simulink model for triplex voter subsystem of the Redundancy Manager.

Redundancy Manager Verification Results

The redundancy manager model originally consisted of two main subsystems: *triplex_voter,* which implements sensor fusion and failure detection for a triply

redundant sensor, and *reset_manager*, which implements the pilot and global failure reset functionality for the sensors and control surfaces for the aircraft. The triplex_voter (see Figure 4) contains a fault monitor that detects failed sensors, failure isolation logic to prevent failed sensors from influencing the output, and a sensor fusion function to synthesize the correct sensor output. It also contains a fault logging function called the fault history table (FHT) that introduces a significant amount of state but is functionally isolated from the rest of the voter. Therefore, we factored this FHT functionality into a third subsystem, *failure processing*.

These models contained a mix of Simulink and StateFlow subsystems, and initially the triplex voter model contained floating-point inputs and outputs. Some of the more complex model features used were data stores with multiple reads/writes within a step, triggered and enabled subsystems with merge blocks, boundary-crossing and directed acyclic transitions through junctions, variables that were used both as integers and as bit flags, bit-level operations (shifts, masks, and bit-level ANDs and ORs), and StateFlow truth tables and functions. As shown in Table 1, during the course of our analysis we derived three analysis models from the RM model, checked 62 properties and found 12 errors. The complete analysis of all the properties using the NuSMV model checker takes approximately 7 minutes.

Table 1. Model size and analysis results for Redundancy Manager.

Subsystem	Number of Simulink subsystems / blocks	Reachable State Space	Properties	Confirmed Errors
Triplex voter without FHT	10 / 96	$6.0 * 10^{13}$	48	5
Failure processing	7 / 42	$2.1 * 10^{4}$	6	3
Reset manager	6 / 31	$1.32 * 10^{11}$	8	4
Totals	23 / 169	N/A	62	12

As an illustration of the properties analyzed for the Redundancy Manager, one requirement states that:

A single frame miscompare shall not cause a sensor to be declared failed.

A miscompare occurs when one of the three sensors disagrees with the other two sensors by more than a predefined tolerance level. This requirement states that a transient error on one of the sensors will not cause the sensor to be declared failed.

In the RM model, failures are recorded in the *device status table* (DST), and the sensor values are input to the model as `input_a, input_b, input_c`. From the requirements, we create variables representing when a sensor value miscompares with the other sensor values:

```
DEFINE
    a_miscompare :=
          (abs(input_a – input_b) > trip_level) &
          (abs(input_a – input_c) > trip_level) &
          (abs(input_b – input_c) <= trip_level);
    b_miscompare := ...
    c_miscompare := ...
```

These variables state that a sensor miscompares if it is outside of tolerance (`trip_level`) with the other two sensors and the other two sensors are within tolerance of each other. In a single frame miscompare, the sensor does not miscompare in the current frame but does miscompare in the next frame. In this case, the sensor must not be marked failed in the next frame.

Given these definitions, we can encode the property in CTL as follows:

```
AG((!a_miscompare) ->
      AX(failure_report != a_failed));
AG((!b_miscompare) ->
      AX(failure_report != b_failed));
AG((!c_miscompare) ->
      AX(failure_report != c_failed));
```

This property was violated in the original triplex voter model. The root cause of this error is that the model used a single counter to record the number of consecutive miscompares to determine whether to fail a sensor. If one sensor miscompares for several frames and then another sensor miscompares for a single frame at the failure threshold, then the second sensor will be declared failed.

This error was corrected by creating separate persistence counters for each input so that miscompares for one sensor will not cause another sensor to be declared failed.

Effort Required

The total effort required to perform the formal analysis was 399.8 hours. As shown in Figure 4, we broke down the analysis time along two axes: the phases of the analysis process and the type of effort. The three main phases of the analysis process are:

- **Preparation:** This task described the effort necessary to extend the analysis tools and condition the models for analysis
- **Initial Verification:** This task described the effort necessary to perform the initial formal analysis of the models
- **Rework:** This task described the effort necessary to fix the models and complete the analysis

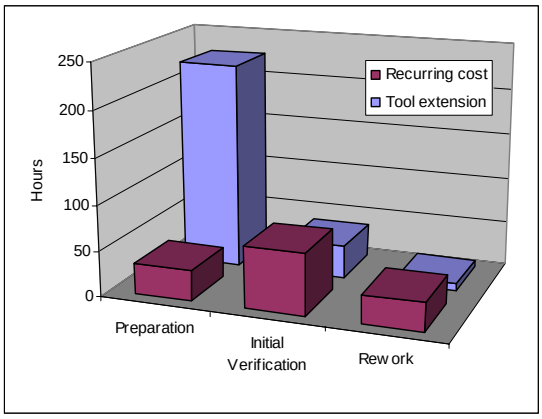

Fig. 4. Categorization of verification effort.

We identified two types of effort: tool modification (one-time tasks extending the capabilities of the tools for this project) and verification activites (tasks that would be carried out for each application). The largest effort for this project was tool modification, extending the Rockwell Collins translators to handle the subset of Simulink / StateFlow used by Lockheed Martin in the CerTA FCS models. This is a non-recurring cost that can be amortized in future analysis projects. This tool modification effort occurred both during the preparation phase (the initial tool up) and in the initial verification phase (where additional tool optimizations were discovered to speed the analysis).

The majority of the one-time tool modification costs occurred during preparation, when we were extending the translation tools to handle the additional blocks used in the CerTA FCS models. The remaining tool modifications costs were due to a handful of bugs in the tool extensions that were found during the verification effort.

The verification activities, which represent recurring costs, were fairly evenly distributed between the preparation, initial analysis, and rework. A significant fraction of the verification time went towards model preparation because the models were not initially constructed for analysis, so several of the "design for analysis" steps detailed in Section 4 had to be performed. Had the formal analysis been integrated into the design cycle, much of this work would have been unnecessary.

After the initial verification and rework effort on the original model, Lockheed Martin provided a modified version of the triplex voter with 10 additional requirements. Since the model had already been structured for automatic translation and analysis, only minor changes were needed. There included addition of input and output ports, definition of appropriate type replacements, and specification of the new properties. In this case, six of the new properties failed due to a single logic error in the new design. The modifications, verification, and results analysis were accomplished in approximately eight hours. This further illustrates the potential for cost savings.

6 Conclusion

This paper describes how formal methods (model checking) can be successfully injected into an avionics software development cycle and how this can lead to early detection and removal of errors in software designs. As a demonstration, we applied this technology to one of the major subsystems of an existing Lockheed Martin Aeronautics Company operational flight plan model, analyzing 62 properties and discovering 12 errors. These results are similar to previous applications of this technology on large avionics models at Rockwell Collins.

In this effort, we performed model checking as an augmentation of the traditional verification process after the models had been developed. In this approach, the model checker provides a verification step that is significantly more rigorous than simulation to ensure that the model works as intended. The total (recurring) time required for analysis was approximately 130 hours, of which about 70 hours were required to prepare the models and perform the initial verification.

Although we were successful, we believe that formal verification can have an even greater impact if its use is anticipated from the outset in the design process. In this paper, we described how model checking can be integrated into the design cycle for models to yield additional benefits. The changes to the development process focused on designing models for analysis and regular use of the model checker during design. The former change significantly reduces the time required to prepare models for analysis, and the latter allows bugs to be found very early in the development cycle, when they are cheapest to fix.

In the next phase of the CerTA FCS project, we will attempt to analyze models that contain large-domain integers and reals. This will be a significant challenge, and will involve investigating new model checking algorithms and theorem provers. On the model checking side, we will be investigating tools that use two recent checking algorithms: k-induction and interpolation, which can be used to analyze the behavior of models containing large-domain integers and reals. Unfortunately, these model checking algorithms have a significant restriction in that they only analyze models containing linear arithmetic. Therefore, we will also be investigating the use of theorem provers that can analyze arbitrarily complex non-linear models, but require greater expertise on the part of users.

Acknowledgments. This work was supported in part by AFRL and Lockheed Martin Aeronautics Company under prime contract FA8650-05-C-3564.

References

1. J. M. Buffington, V. Crum, B. H. Krogh, C. Plaisted, R. Prasanth, P. Bose, T. Johnson, Validation & verification of intelligent and adaptive control systems (VVIACS), AIAA Guidance, Navigation and Control Conference, Aug. 2004.
2. M. W. Whalen, J. D. Innis, S. P. Miller, and L. G. Wagner, ADGS-2100 Adaptive Display & Guidance System Window Manager Analysis, NASA Contractor Report CR-2006-213952, February 2006.

3. S. Miller, M. P.E. Heimdahl, and A.C. Tribble, Proving the Shalls, Proceedings of FM 2003: the 12th International FME Symposium, Pisa, Italy, Sept. 8-14, 2003.
4. E. Clarke, O. Grumberg, and P. Peled, Model Checking, The MIT Press, Cambridge, Massachusetts, 2001.
5. C. Heitmeyer, R. Jeffords., and B. Labaw, Automated Consistency Checking of Requirements Specification, ACM Transactions on Software Engineering and Methodology (TOSEM), 5(3):231-261, July 1996.
6. S. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr, The CoRE Method for Real-Time Requirements, IEEE Software, 9(5):22-33, September 1992.
7. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. In 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, volume 2860 of Lecture Notes in Computer Science, pages 111--125. Springer-Verlag, October 2003.
8. J. L. Lions, Arianne 5 Flight 501 Failure Report by the Inquiry Board, ESA Technical Report No. 33-1996, July 1996.
9. Y. Choi, M. P.E. Heimdahl, and S. Rayadurgam, Domain reduction abstraction. Technical Report 02-013. University of Minnesota, April 2002
10. A. C. Tribble, David D. Lempia, and Steven P. Miller, Software Safety Analysis of a Flight Guidance System, Proceedings of the 21st Digital Avionics Systems Conference (DASC'02), Irvine, California, Oct. 27-31, 2002.
11. Reactive Systems, Inc, Reactis Home Page, http://www.reactive-systems.com.