Automated Integrative Analysis of State-Based Requirements

Barbara J. Czerny
Delphi Delco Electronics Systems
One Corporate Center PO Box 9005
Kokomo, IN 46904-9005, USA
czerny@mail.delcoelect.com

Mats P.E. Heimdahl
University of Minnesota 4-192 EE/CS
Dept. of Computer Science
Minneapolis, MN 55455, USA
heimdahl@cs.umn.edu

Abstract

Statically analyzing requirements specifications to assure that they possess desirable properties is an important activity in any rigorous software development project. The analysis is performed on an abstraction of the original requirements specification. Abstractions in the model may lead to spurious errors in the analysis output. Spurious errors are conditions that are reported as errors, but information abstracted out of the model precludes the reported conditions from being satisfied. A high ratio of spurious errors to true errors in the analysis output makes it difficult, error-prone, and time consuming to find and correct the true errors. In this paper we describe an iterative and integrative approach for analyzing state-based requirements that capitalizes on the strengths of a symbolic analysis component and a reasoning component while circumventing their weaknesses. The resulting analysis method is fast enough and automated enough to be used on a day-to-day basis by practicing engineers, and generates analysis reports with a small ratio of spurious errors to true errors.

1. Introduction

Statically analyzing requirements specifications to assure that they possess desirable properties is an important activity in any rigorous software development project. Errors in the requirements that go undetected and propagate to later stages of development are the most costly to correct [15, 18]. Therefore, it is desirable to ensure that the requirements document satisfies certain properties before proceeding to later stages of the development process.

However, static analysis is performed on a formal model of the requirements that is an abstraction of the original requirements specification. Some degree of abstraction is necessary or the analysis becomes intractable. The output from the analysis is a report of the desired properties that the specification fails to satisfy. Often, abstractions in the analysis model lead to spurious errors in the analysis output. Spurious errors are conditions that are reported as errors, but information that was abstracted out of the analysis model precludes the reported conditions from being satisfied. For example, information about relational expressions is abstracted from the model and the analysis incorrectly reports that there is a problem if the condition $[x - y > 1200] \land [x \leq (1200 + y)]$ is satisfied. A high ratio of spurious errors to true errors in the analysis output makes it difficult, error-prone, and time consuming to find and correct the true errors in the specification.

Two desirable properties that certain requirements documents should satisfy (for example, the requirements for critical systems) are completeness (a behavior is specified for every possible input) and consistency (no conflicting behaviors are specified). Analyzing for completeness and consistency in state-based requirements generalizes to analyzing logical expressions for satisfiability and mutual exclusion. Two methods for analyzing logical expressions for satisfiability and mutual exclusion are symbolic methods such as those that rely on Binary Decision Diagrams (BDDs), and reasoning methods such as theorem proving. Symbolic methods are fast and fully automated, but generate output that may contain many spurious errors since the analysis model contains many abstractions. Reasoning methods tend to be slower and require more manual intervention, but generate more accurate output since the analysis model contains fewer abstractions.

We developed an iterative approach for analyzing state-based requirements that integrates a symbolic analysis component and a reasoning component. Our method is automated and easy to use. It capitalizes on the strengths of the individual components while circumventing their weaknesses, thus, resulting in an analysis process that is fast enough and automated enough to be used on a day-to-day basis by practicing engineers, and that generates analysis reports with a small ratio of spurious errors to true errors.

The remainder of this paper is organized as follows: Section 2 overviews related work in integrative analysis and describes how our approach is different, Section 3 provides some background information and describes in more detail the problem that we set out to solve, Section 4 describes the integrative and iterative analysis process we developed to analyze logical expressions for satisfiability and mutual exclusion, and Section 5 describes the analysis of output and the iteration options available to the analyst. In Section 6 we discuss the results we obtained from applying our method to a large real-world avionics specification. Finally, in Section 7 we present our conclusions.

2. Related work

The integrative approach in itself is not unique. There are several others who have also developed integrative analysis techniques. Most of these techniques have been applied in the area of hardware verification, or analysis of concurrent programs. Joyce and Seger developed an integrated approach to formal hardware verification that combines BDD-based symbolic simulation techniques with interactive theorem proving [16]. Young, Taylor, Forester, and Brodbeck have integrated static concurrency analysis with symbolic execution to detect anomalous synchronization patterns in concurrent Ada programs [22, 23, 24]. In [10], Havelund and Shankar describe a series of protocol verification experiments that combine theorem proving and model checking. It is clear that an integrative approach to analysis is essential, since all individual analysis techniques suffer from limitations of one form or another. The integrative approaches represent attempts to capitalize on the strengths of the individual techniques and circumvent their weaknesses.

Our integrative analysis approach is based on trying the simple, straightforward methods first and, if these methods fail, apply the more complex and computationally expensive methods. Our iterative analysis is based on identifying the information that was abstracted out of the model that is leading to spurious errors, feeding the information back into the model, and re-running the analysis. Our approach is unique in that it provides guidelines to help the analyst decide which analysis component to apply on the first iteration based on the input and it provides guidelines for which actions to take on subsequent iterations based on the output. In addition, we provide a method to help the analyst identify the missing information so the spurious errors can be eliminated and the true errors can be more readily identified. In this paper, we refer to the missing information as augmenting information since its inclusion into the analysis process augments the accuracy of the analysis output by eliminating spurious errors. The idea is that since the spurious errors are the result of missing information (abstraction) in the model, identifying the missing information leading to the spurious errors and adding it back into the analysis process, should make the spurious errors disappear. In other words, if all of the missing information is identified, and the original guarding conditions are consistent (complete), then augmenting the analysis process with the missing information will yield the correct output of FALSE (TRUE), showing that the guarding conditions are consistent (complete). In this paper, we do not discuss our technique for identifying the missing information; see [6] for a detailed description of this technique.

3. Background and problem

In state-based languages such as Statecharts [8, 9], SCR [14], and RSML [18], the transitions between states are guarded by conditions; the guarding condition must be true before the transition can be taken. In the definition of completeness and consistency provided in [12] the properties imply the following:

1. Every state must have a deterministic behavior (transition) defined for every possible input event,
2. The logical OR of the guarding conditions on every transition out of any state must form a tautology; for any condition, there is always a transition that can be taken, and
3. The logical AND between the guarding conditions on two transitions out of a state must form a contradiction; for each possible condition, there is only one feasible transition out of every state.

Thus, verifying consistency and completeness in state-based requirements primarily involves calculating the AND and OR of the guarding conditions on the transitions to see if they form contradictions and tautologies.

3.1. Analysis procedures

In our work, we have investigated two main approaches for manipulating the guarding conditions to check for contradictions and tautologies: (1) theorem proving (reasoning) and (2) symbolic manipulation using Binary Decision Diagrams (BDDs). BDDs are directed acyclic graphs that represent Boolean formulas in a canonical form. Algorithms for manipulating BDDs, for example, ANDing and ORing Boolean formulas, are efficient and provide good average performance [2]. For the theorem proving component of our method, we chose the Prototype Verification System (PVS). PVS is a specification and verification system that provides an interactive environment for the development and analysis of formal specifications [5, 20, 21].

3.2. Spurious errors

Since all analysis techniques rely on abstraction to generate a system model that can be analyzed in a computationally tractable manner (i.e., avoid the state explosion problem), spurious errors may be reported that would be eliminated if certain abstractions were not made. Thus, the problem of spurious errors is common to all analysis techniques. Currently, it is left up to the analyst to determine which error reports represent true errors and which error reports represent spurious errors.

In essence, in each spurious error report, there is some undetected contradiction that exists between the constituent components of the reported expression that actually precludes the reported error from being a true error. After numerous case studies and experiments, we identified four classes of undetected contradictions that lead to spurious errors. We classified the spurious errors according to the undetected contradictions that cause them.

1. Spurious errors involving simple and obvious contradictions between predicates such as enumerated type predicates and predicates involving simple arithmetic expressions. For example, an error report that requires the expression $[(x - y) > 1200] \land [x \leq (1200 + y)]$ to be TRUE.

2. Spurious errors involving three or more predicates containing related linear arithmetic expressions. For example, an error report that requires the expression $(i > j) \land (j > k) \land (i - k < 0)$ to be TRUE.

3. Spurious errors involving non-linear expressions. For example, an error report requiring $(z^2 < x/y) \land (z^2 \geq x/y)$ to be TRUE.

4. Spurious errors related to the structure of the state machine, or spurious errors related to information about the environment in which the system will operate.

Different analysis techniques are able to eliminate different classes of spurious errors, but no single analysis technique can eliminate all spurious errors. Spurious errors of the types involved in the first two classes listed above can be eliminated by augmenting the analysis process with decision procedures. Decision procedures are algorithms designed to reason about expressions and, if possible, to perform simplifications on those expressions. For example, we know that symbolic analysis using BDDs cannot reason about the components of the expressions and therefore cannot make decisions about whether or not two or more interdependent expressions contradict each other. This inability to reason about relationships between expressions may lead to many spurious error reports in the analysis output. Adding procedures to the symbolic analysis process that can perform some simple reasoning about the interdependencies between expressions will strengthen the overall analysis process and result in fewer numbers of spurious error reports.

Tools using reasoning components augmented with decision procedures, such as PVS, can easily eliminate all types of class 1 and class 2 spurious errors. PVS can also eliminate some spurious errors involving simple non-linear inequalities.

In general, no analysis technique can eliminate class 4 spurious errors, since this class of spurious errors results because information that is required by the analysis process has been abstracted from the model, or is related to some environmental constraints that were not modeled. To eliminate spurious errors of class 4 requires one to identify the relevant information that is missing from the analysis process and to augment the analysis process with this information.

From our experiments we discovered that no one individual static technique for performing the analysis is sufficient to satisfy the desired goals of analysis [24]; speed, automation, and accuracy. There are trade-offs between the amount of abstraction an analysis method relies on, the degree of automation, the speed with which the analysis completes, and the level of accuracy in the analysis output. When more details are included in the analysis process, the analysis often requires costly and time consuming user intervention.

Furthermore, we know that even the most powerful analysis techniques will generate spurious errors when information required for the analysis has been abstracted out of the model.

To summarize, symbolic methods are fully automated, but may generate inaccurate analysis reports because many functions are not interpreted; i.e., the semantics of the functions are abstracted away. Reasoning methods can generate more accurate analysis reports, but are more costly to use. If static analysis techniques are going to be used more frequently in industrial applications, they must be automated [7, 13, 19]. We want an analysis method that is automated, fast, and that generates accurate analysis reports so it is feasible to use in industrial settings. This paper specifically addresses the need for automation and speed by describing an automated process to check logical expressions for satisfiability and mutual exclusion that is feasible in industrial settings. In [6] we also address the issue of accuracy in the analysis output.

4. Integrative analysis

In this section we describe the integrative and iterative analysis approach we developed to analyze disjunctive and conjunctive expressions for satisfiability and mutual exclusion. An application of the process developed in this research is in the analysis of state-based requirements for completeness and consistency. To demonstrate the scalability of our analysis process to industrial problems, we applied our method to a large real-world avionics specification, specified in RSML, to check parts of the specification for completeness and consistency. The results of this application are reported in Section 6.

4.1. General analysis process

The inputs to the analysis process for mutual exclusion are two logical expressions. If the expressions are mutually exclusive, the conjunction will reduce to FALSE. If the expressions are not mutually exclusive, i.e., the conjunction of the expressions does not reduce to FALSE, the analysis will report the logical expressions that are satisfiable by both of the disjuncts at the same time. The analyst is then left with the task of determining how to modify the two original logical expressions so they are mutually exclusive.

The input to the analysis process for satisfiability may be one or more logical expressions. The analysis process forms the disjunction of the logical expressions. If the disjunction of the expressions is satisfiable, then the disjunction reduces to TRUE. If the disjunction of the expressions is not satisfiable, the analysis will report the logical expressions to the analyst that are not satisfiable by any of the original logical expressions. The analyst can then determine the logical expressions that need to be added to make the disjunction of the expressions a tautology. We do not address the issue of how the analyst corrects the problems once they are identified; this is a difficult problem and requires further research.

Figure 1 shows the general analysis process and the integration of the symbolic and reasoning components. The logical expressions to be analyzed are stored in some machine readable form; for example, a machine readable RSML specification. The analysis process begins by first parsing the logical expressions and converting them into an internal representation. If the analyst chooses to perform symbolic analysis, then the tool performs symbolic analysis using BDDs and generates two outputs: a BDD node profile[1], and an analysis report in disjunctive normal form in a tabular format (an AND/OR table). The latter output of the symbolic analysis component may be samples of the resulting disjunctive normal form expression; the analyst may specify samples if the resulting disjunctive expression is too large to report in its entirety; for example, we encountered expressions that contained thousands or millions of disjuncts.

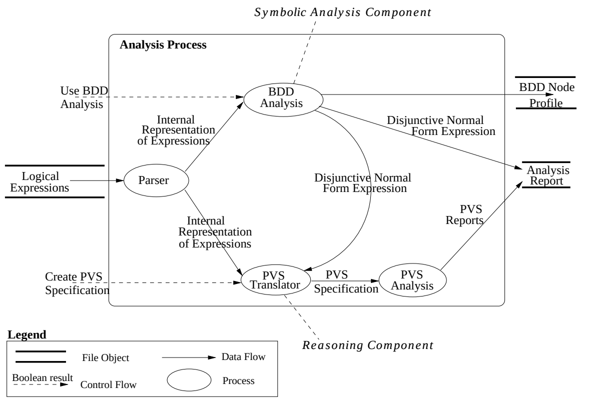

Figure 1. General analysis process and integration of symbolic and reasoning components.

If the analyst chooses to create a PVS specification, the tool converts the internal representation of the logical expressions into a PVS specification. The output of the PVS analysis is either a finished proof (no unprovable subgoals), or a report of unprovable subgoals. The analyst can also choose to create a PVS specification for the disjunctive normal form expression output from the symbolic analysis component (this option is shown in the figure by the data flow from the *BDD analysis* process to the *PVS translator* process).

4.2. Tools and tool integration

The specific tools we use in our analysis process and the flow of data between the tools are shown in Figure 2. The tools described in this section are specific to the application of our analysis process to the analysis of state-based requirements (namely, RSML requirements) for completeness and consistency. The symbolic analysis component sits on top of a BDD library created by Long at Carnegie Mellon University [1].

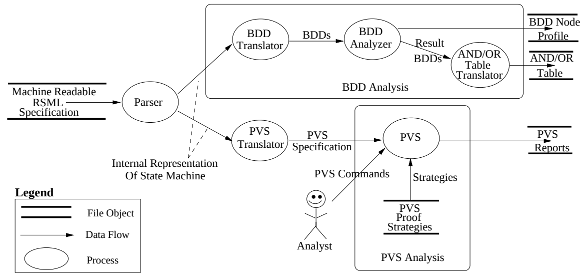

Figure 2. Overview of analysis tools and data flow between the tools.

The symbolic analysis component consists of three subprocesses: a *BDD translator* that translates the RSML AND/OR table[2] representation of guarding conditions to Binary Decision Diagrams, a *BDD analyzer* that manipulates

[1] A BDD node profile is simply a histogram showing the number of nodes at each level in a BDD.

[2] An AND/OR table is a disjunctive normal form tabular representation of logical expressions.

the BDD representation of the guarding conditions to check for tautologies and contradictions, and an *AND/OR table translator* that converts the result BDDs output from the BDD analyzer process to AND/OR table format to present to the analyst.

The *PVS translator* converts the internal representation of the state machine to a PVS specification. We do not discuss the translation process from an RSML specification to a PVS specification in this paper. For details regarding this translation process see [6, 11]. Once the translation process is complete, the analyst initiates PVS and automatically parses and typechecks the theories and declarations generated from the RSML specification. When parsing and typechecking is finished, the analyst invokes the PVS prover on the conjecture to be proved.

We developed a set of proof strategies that allow the PVS analysis to be more automated, and largely free the analyst from having intimate knowledge of the PVS prover commands. When the analyst invokes the prover, the set of strategies developed during this research is loaded into PVS and the strategies become available to the PVS prover. We developed strategies for proving that the pairwise conjunction of two logical expressions is a contradiction (i.e., the logical expressions are mutually exclusive, or consistent), and we developed strategies for proving that the disjunction of logical expressions is a tautology (i.e., the disjunction of the logical expressions is satisfiable). In most cases, the strategies allow the analyst to perform proofs of completeness and consistency with a single command (the strategy name). An example strategy is shown in Figure 3; this strategy is used to check guarding conditions for completeness.

```
(defstep complete2
 (apply (then (skolem!)
   (rewrite-msg-off)
   (auto-rewrite-defs$)
   (do-rewrite$)
   (repeat*
     (try (bddsimp)
       (try (record) (assert) (postpone))
               (skip))))))
```

Figure 3. PVS strategy for proving guarding conditions complete.

In addition, our tool provides command line options that allow the analyst to choose which analysis she wants to perform. For example, the analyst might choose to run PVS or BDD analysis on the original expressions, or she might choose to translate the BDD analysis output to a PVS specification.

The outcome of any analysis is an error report of some form that is presented to the analyst. Error reports from our analysis process are presented to the analyst as Boolean expressions in disjunctive normal form (AND/OR table format). In terms of analyzing requirements for completeness and consistency, each report represents either (1) an incompleteness, that is, a condition the requirements do not handle, or (2) an inconsistency, that is, a condition where two or more responses are specified. We now describe the iteration options available to the analyst, and the analysis of output.

5 Iteration options and analysis of output

In this section we describe the analysis of output and the iteration options available to the analyst. Again, in this section we apply our method to RSML requirements to check the requirements for completeness and consistency, but the approach could be generalized to apply to any analysis that requires the manipulation of large Boolean expressions. Figure 4 shows the possible outputs from the analysis process, the possible outcomes of the output analysis process, and the options available to the analyst based on the outcome of the *Analyze Output* process.

After the automated portion of the analysis is finished and the analysis report is generated, the analyst inspects the output to determine the next course of action; the *Analyze Output* process in the figure. If the report shows the guarding conditions complete or consistent, the process is finished and no further iterations are required. If the output size from either analysis process is manageable (for example, 30 or fewer errors), the analyst manually inspects the output for the true errors. For the purposes of this discussion we assume that symbolic analysis using BDDs is applied first. If the output size from the symbolic analysis is not manageable for an analyst, but is manageable for PVS

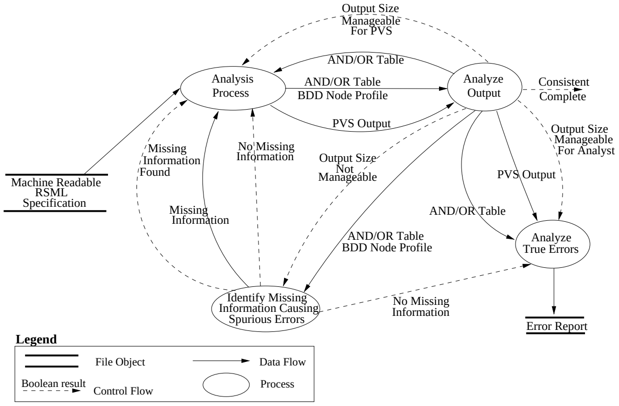

Figure 4. Analysis of output and iteration options of the analysis.

(as determined by the analyst), it is fed back into the analysis process, a PVS specification is generated, and the PVS output is reported to the analyst.

If the output size from either analysis process is not manageable, we assume that most of the error reports are spurious and exist because there is some information missing from the model. We can make this assumption since the initial specification was intended to be consistent (complete) and we assume the analyst produced a specification that does not stray too far from this intention. The outputs from the symbolic analysis, the BDD node profile and the AND/OR table, are used to identify the information that is missing; the process labeled *Identify Missing Information Causing Spurious Errors* in Figure 4 (the details of this process are beyond the scope of this paper; see [6] for details). If no missing information can be found, and both analysis processes have been tried, then the errors must be true errors, and the analyst must examine them manually. If only symbolic analysis has been tried and no missing information can be found, either the symbolic analysis output is converted to a PVS specification and PVS analysis is run, or PVS analysis is applied to the original expressions. If missing information is found, it is added to either the machine readable RSML specification or the PVS specification generated from the machine readable RSML specification, and either symbolic analysis or PVS analysis is run on the augmented specification. Adding augmenting information to the analysis process is discussed in Section 5.1.

Note that it is not necessary to start the analysis process using symbolic analysis. The analyst may choose to start with PVS. A situation where it would be useful to start with PVS is when the predicates contain many arithmetic linear expressions, for example. Should PVS result in too many error reports or fail to halt, then the analyst can apply symbolic analysis to the original specification and attempt to identify missing information using the process described in [6]. We developed a set of heuristics to help the analyst choose which analysis process to apply. The heuristics are based on what the constituent components of the logical expressions are comprised of, and what the results of an application of an analysis are. We learned the heuristics during application of our method.

First, for guarding conditions that contain a large number of relational type predicates with a large number of possible multi-way interdependencies, it is best to apply PVS on the first iteration. Symbolic analysis (without some additional decision procedures) will not be effective for such types of expressions. For example, expressions such as $(x < c1, x >= c1)$, $(x <= c2, x > c2)$ where x is an integer or integer function and $c1$ and $c2$ are integer constants. And expressions such as $(x < c3, x >= c4)$ and $(x >= c4, x <= c5)$ where x is an integer or an integer function and $c3$, $c4$, and $c5$ are constants subject to the constraint: $c3 < c4, c5 < c4$. In all of these cases the predicates cannot be both TRUE or both FALSE at the same time.

Second, for guarding conditions that contain a large number of enumerated type predicates, either symbolic analysis with our decision procedures, or PVS can be applied first. However, since PVS has more decision procedures available, applying PVS on the first iteration may yield fewer spurious or redundant error reports.

Third, if the guarding conditions being analyzed are very large, it is best to apply symbolic analysis on the first iteration. The symbolic analysis report may show that some of the guarding conditions are consistent (complete). When the report from the symbolic analysis shows many inconsistencies (incompletenesses) then apply our process described in [6] to identify any missing information. Once the missing information has been identified, it is best to augment the PVS specification with the missing information, and use PVS on the augmented specification. Since PVS has more decision procedures available, the output may report fewer spurious errors and fewer redundant errors when true errors exist. We do not start with PVS for complex guarding conditions since, as our results showed, there are some guarding conditions that PVS fails to halt on; this failure is generally because some information is lacking from the specification and the proof process. Once the PVS specification and proof process are augmented with the missing information, PVS is effective.

Fourth, for guarding conditions that contain a significant number of linear arithmetic predicates, non-linear arithmetic predicates with constants, or expressions involving division by constants when the expressions are structurally equivalent, it is best to apply PVS to the guarding conditions on the first iteration; symbolic analysis using BDDs cannot effectively manage any of the aforementioned problems unless all of the predicates are structurally equivalent and there are no multi-way interdependencies between predicates (for example, two or more arithmetic predicates that cannot be satisfied at the same time).

5.1. Adding augmenting information to the analysis process

To add augmenting information to our symbolic analysis component, we create a disjunctive normal form expression (in the form of an AND/OR table) representing the augmenting information and add it to the RSML specification. To add augmenting information to the PVS analysis we create an axiom representing the augmenting information, add it to the PVS specification, and introduce the axiom into the proof process when the PVS prover is invoked.

5.2. Future work in automating the process

We can further automate our process by creating a shell script and applying some heuristics we learned when we applied our method to the application described in Section 6. Since our analysis is driven by command line options, we can create a shell script to initiate the analysis, examine the output for some specific situations, and based on the output, decide on the next course of action. For example, our script could start the analysis process to use BDDs to check a specification for consistency. Once the analysis is complete, the script would check to see how many error reports were generated. If the number of error reports is between 50 and 1000, for example, the script could take the error reports (in AND/OR table format) and start the analysis process with the option to translate the tabular output into a PVS specification. PVS is equipped with a batch mode, so the script could also initiate PVS once the translation is complete. If the number of error reports from the symbolic analysis is under 50, for example, the script could present the symbolic analysis results to the analyst for review.

It is also possible for us to incorporate our heuristics into the shell script and provide guidance to the analyst in what the best options are for the analyst to try next. This is useful for parts of our analysis process that require the analysts input. For example, if the number of error reports is large, say on the order of thousands, the script could inform the analyst that information may be missing from the model and print out some hints to help the analyst identify the missing information. The analyst can then use the hints provided by the script to locate the missing information in the original specification.

6. Application of method and results

We applied our iterative and integrative analysis process to the TCAS (Traffic alert and Collision Avoidance System) II requirements specification [17]. TCAS II is a complex avionics system that is required on all commercial aircraft carrying 30 or more passengers through US Airspace. The system monitors the airspace around the aircraft for other aircraft that may be on a collision course, and takes evasive action if a collision is imminent. The entire specification for TCAS II (version 6.04A) comprises a little over 400 pages [3]. The requirements consist of two main parts, *Own-Aircraft* and *Other-Aircraft* [4]. We concentrated our efforts on several transitions with some of the most complex guarding conditions in one of the most complex portions of the TCAS II requirements specification.

One of the states within the state *Other-Aircraft*, *Intruder-Status*, tracks the status of aircraft within the proximity of the tracking aircraft (own aircraft). Four states within the state *Intruder-Status* are *Other-Traffic*, *Proximate-Traffic*, *Potential-Threat*, and *Threat*. A transition from *Proximate-Traffic*, *Potential-Threat*, or *Threat* to the state *Other-Traffic* means that the intruder aircraft is no longer in the airspace close to own aircraft but is still being monitored; in other words, the status of the intruder in relation to the monitoring aircraft has been downgraded. A transition from either *Proximate-Traffic*, *Potential-Threat*, or *Other-Traffic* to *Threat* means a potential collision is imminent and own aircraft is directed to take evasive action to avoid a collision.

Figures 5 and 6 show the guarding conditions for the transitions from state *Proximate-Traffic* to *Other-Traffic* and from state *Proximate-Traffic* to *Threat*. The guarding conditions are expressed as AND/OR tables. The AND/OR table shows the predicates to the left, and the columns of truth values represent different truth assignments the predicates can have. The table is interpreted as the disjunction of the conjunction of the truth values in the individual columns. In other words, if any of the columns is TRUE, the entire table is TRUE. The predicates sub-scripted with an *m* represent macro predicates. The *Threat-Condition* macro is shown in Figure 7. As the *Threat-Condition* macro shows, a macro may contain other macros. Thus, there may be several levels of indirection within the guarding conditions. The *Threat* macro when fully expanded, is one of the most complex macros in the TCAS II requirements specification. The macros included in the *Threat-Condition* macro range in size from the smallest, a one column two row table, to the largest, a six column ten row table.

We used our symbolic analysis component to check these conditions for consistency. The symbolic analysis reported over four million potential inconsistencies between the two guarding conditions. With this many error reports we concluded that most of them were spurious and that there was some information missing from the symbolic model that was causing the spurious errors. We applied our technique described in [6] to identify the missing information. In this case, we found that there was a relation between a particular input variable, *Other-Alt-Reporting*, and a state machine named *Alt-Reporting* that consists of the three states *Yes*, *Lost*, and *No*. Whenever the variable *Other-Alt-Reporting* is TRUE, *Alt-Reporting* must be in the state *Yes*. We codified this information in the form of an AND/OR table (Figure 8) and augmented the analysis model with the information. We reran our symbolic analysis component including the augmenting information and with our decision procedures enabled. The results of this second iteration showed that the guarding conditions were consistent. We also translated the original guarding conditions to a PVS specification and initiated PVS analysis. PVS ran for over a day on a SPARCserver 1000 with 256 MB main memory and four 85MHz CPUs. We aborted the process. We then augmented the PVS specification with the information we iden-

Transition(s): Proximate-Traffic ⟶ Other-Traffic

Location: Other-Aircraft ▹ Intruder-Status$_{\text{s-136}}$

Trigger Event: Air-Status-Evaluated-Event$_{\text{e-279}}$
Condition:

		OR							
	Alt-Reporting$_{\text{s-101}}$ **in state** Lost	T	T	·	·	·	·	·	·
	RA-Mode-Canceled$_{\text{m-199}}$	·	·	T	T	·	·	·	·
	Alt-Reporting$_{\text{s-101}}$ **in state** No	·	·	T	T	T	T	·	·
A	Other-Bearing-Valid$_{\text{v-120}}$ = True	F	·	F	·	F	·	·	·
N	Other-Range-Valid$_{\text{v-117}}$ = True	·	F	·	F	·	F	·	·
D	Proximate-Traffic-Condition$_{\text{m-216}}$	·	·	·	·	·	·	F	·
	Potential-Threat-Range-Test$_{\text{m-214}}$	·	·	·	·	T	T	·	·
	Potential-Threat-Condition$_{\text{m-213}}$	·	·	·	·	·	·	F	·
	Threat-Condition$_{\text{m-8}}$	·	·	·	·	·	·	F	·
	Other-Air-Status$_{\text{s-101}}$ **in state** On-Ground	·	·	·	·	·	·	·	T

Output Action: Intruder-Status-Evaluated-Event$_{\text{e-279}}$

Figure 5. Transition from Proximate-Traffic to Other-Traffic.

Transition(s): Proximate-Traffic ⟶ Threat

Location: Other-Aircraft ▹ Intruder-Status$_{\text{s-136}}$

Trigger Event: Air-Status-Evaluated-Event$_{\text{e-279}}$
Condition:

Threat-Condition$_{\text{m-8}}$	T

Output Action: Intruder-Status-Evaluated-Event$_{\text{e-279}}$

Figure 6. Transition from Proximate-Traffic to Threat.

Macro: Threat-Condition
Definition:

		OR	
	RA-Inhibit$_{\text{m-217}}$	F	F
	Other-Air-Status$_{\text{s-101}}$ **in state** Airborne	T	T
	Threat-Range-Test$_{\text{m-224}}$	T	T
A	Threat-Alt-Test$_{\text{m-223}}$	T	T
N	Reply-Invalid-Test$_{\text{m-218}}$	F	F
D	TCAS-TCAS-Crossing-Test$_{\text{m-221}}$	F	·
	Level-Wait$_{\text{s-101}}$ **in state** 3	F	T
	Alt-Separation-Test$_{\text{m-196}}$	F	F
	Low-Firmness-Separation-Test$_{\text{m-207}}$	F	F

Figure 7. The Threat-Condition Macro.

Macro: Other-Alt-Reporting-Alt-Reporting-Assertion

Definition:

			OR	
AND	Other-Alt-Reporting$_{\text{v-113}}$ = True	T	F	F
	Alt-Reporting$_{\text{s-101}}$ **in state** Yes	T	F	F
	Alt-Reporting$_{\text{s-101}}$ **in state** Lost	F	T	F
	Alt-Reporting$_{\text{s-101}}$ **in state** No	F	F	T

Figure 8. Augmenting information in tabular form.

tified above, and reran the PVS analysis. PVS showed the guarding conditions consistent in 44.87 seconds.

For two other states in the TCAS II specification, we analyzed a total of fourteen pairs of guarding conditions. Of these fourteen, five proved consistent after a single iteration of both the symbolic analysis and the PVS analysis. Both the symbolic analysis and the PVS analysis required only a few seconds to report that the guarding conditions were consistent.

For the remaining nine pairs, the first iteration of the symbolic analysis reported from four to 403 inconsistencies. We automatically translated all of the reported inconsistencies from the symbolic analysis to a PVS specification and ran PVS to see if any further reductions in the reported inconsistencies could be achieved. In all cases (except for the minimum output of four inconsistencies from symbolic analysis), the PVS output was significantly reduced. The number of inconsistencies reported by PVS ranged from one to 46.

Manual inspection of the reported inconsistencies seemed to confirm that the guarding conditions could both be satisfied at the same time (i.e., they were inconsistent). These potential inconsistencies were reported to the maintainers of the specification. The maintainers reviewed the reported inconsistencies and noted that in their definition of the semantics of RSML, transitions out of a higher level state take precedence over transitions out of a lower level state. Thus, both transitions cannot be satisfied at the same time; the transition from the higher level state overrides the transition from the lower level state. Therefore, all of the guarding conditions for transitions out of the two states, in their choice of semantics, are mutually exclusive.

We applied our analysis method to many other guarding conditions from the TCAS II specification and all of our results were promising (see [6] for additional case studies).

7. Conclusion

Statically analyzing requirements specifications to assure that they possess desirable properties is an important activity in any rigorous software development project. However, static analysis is performed on a formal model of the requirements that is an abstraction of the original requirements specification. In many cases, abstractions in the analysis model lead to spurious errors in the analysis output. A high ratio of spurious errors to true errors in the analysis output makes it difficult, error-prone, and time consuming to find and correct the true errors in the specification.

Three desirable criteria for any analysis technique to satisfy are efficiency (in terms of speed), accuracy (a small ratio of spurious errors to true errors in the analysis output), and automation. From our experience, it is clear that symbolic manipulation and reasoning methods applied individually cannot sufficiently satisfy all three criteria. Therefore, an approach that integrates the strengths of the individual components and circumvents their weaknesses is needed.

In this paper we described an iterative approach for analyzing state-based requirements for completeness and consistency that integrates a symbolic component using BDDs with a reasoning component using PVS. Our approach takes advantage of the strengths of the individual techniques while circumventing their weaknesses. The resulting analysis process is fast and automated enough to be used on a day-to-day basis by practicing engineers, and generates analysis reports with a small ratio of spurious errors to true errors.

Since our approach is simple to apply and is driven by command line options, we can further automate it by creating a shell script which incorporates the heuristics we learned from applying our method. This makes it more accessible to practicing engineers and easier to use.

We applied our method to several large real-world case studies and the results were promising.

References

[1] BDD(3). Manual page, June 1993. Manual page for BDD package obtained via ftp from Carnegie-Mellon University.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Computer Aided Verification; 9th International Conference, CAV '97 Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327, June 1997.

[4] W. Chan, R. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 102–112, March 1998.

[5] J. Crow, S. Owre, J. Rushby, et al. A tutorial introduction to PVS. Presented at WIFT 95: Workshop on Industrial-Strength Formal Specification Techniques, 1995. Available at http://www.csl.sri.com/fm-papers.html.

[6] B. J. Czerny. *Integrative Analysis of State-Based Requirements for Completeness and Consistency*. PhD thesis, Michigan State University, May 1998.

[7] S. Gerhart et al. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, Jan. 1994.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[9] D. Harel et al. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, pages 403–413, Apr. 1990.

[10] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe (FME '96)*, pages 662–681, 1996.

[11] M. P. Heimdahl and B. J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *Proceedings of the IEEE High Assurance Systems Engineering Workshop*, pages 252–262, 1996.

[12] M. P. Heimdahl and N. Leveson. Completeness and Consistency Analysis of State-Based Requirements. *IEEE Transactions on Software Engineering*, TSE-22(6):363–377, June 1996.

[13] C. Heitmeyer et al. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering*, March 1995.

[14] C. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[15] M. S. Jaffe et al. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, pages 241–257, Mar. 1991.

[16] J. J. Joyce and C.-J. H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. Technical Report TR-93-18, University of British Columbia, Department of Computer Science; Vancouver, B.C. V6T 1Z2 Canada, 1993.

[17] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. Reese. TCAS II requirements specification.

[18] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[19] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[20] S. Owre, N. Shankar, and J.M.Rushby. *The PVS Specification Language*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, April 1993. Available at http://www.csl.sri.com/pvs.html.

[21] S. Owre, N. Shankar, and J.M.Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993. Available at http://www.csl.sri.com/pvs.html.

[22] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[23] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analaysis in a software development environment. In *Proceedings of the 3rd International Workshop on Testing, Analysis, and Verification*, 1989.

[24] M. Young, R. N. Taylor, D. L. Levine, K. Forester, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. Technical Report TR-128-P, Software Engineering Research Center, November 1994.