Replace this file with prentcsmacro.sty for your meeting,
or with entcsmacro.sty for your meeting. Both can be
found at the ENTCS Macro Home Page.

Verifying LTL Properties of Bytecode with Symbolic Execution [1]

Pietro Braione, Giovanni Denaro[2,3]

Dipartimento di Informatica Sistemistica e Comunicazione (DISCO)
Università degli Studi di Milano-Bicocca, Milano, Italy

Bohuslav Křena[4]

Faculty of Information Technology (FIT)
Brno University of Technology, Brno, Czech Republic

Mauro Pezzè[5]

Dipartimento di Informatica Sistemistica e Comunicazione (DISCO)
Università degli Studi di Milano-Bicocca, Milano, Italy
and
Facoltà di Scienze Informatiche
Università della Svizzera Italiana, Lugano, Switzerland

Abstract

Bytecode languages are at a very desirable degree of abstraction for performing formal analysis of programs, but at the same time pose new challenges when compared with traditional languages. This paper proposes a methodology for bytecode analysis which harmonizes two well-known formal verification techniques, *model checking* and *symbolic execution*. Model checking is a property-guided exploration of the system state space until the property is proved or disproved, producing in the latter case a counterexample execution trace. Symbolic execution emulates program execution by replacing concrete variable values with symbolic ones, so that the symbolic execution along a path represents the potentially infinite numeric executions that may occur along that path. We propose an approach where symbolic execution is used for building a possibly partial model of the program state space, and on-the-fly model checking is exploited for verifying temporal properties on it. The synergy of the two techniques yields considerable potential advantages: symbolic execution allows for modeling the state space of infinite-state software systems, limits the state explosion, and fosters modular verification; model checking provides fully automated verification of reachability properties of a program. To assess these potential advantages, we report our preliminary experience with the analysis of a safety-critical software system.

Keywords: Symbolic execution, code-based model checking of software.

[1] This work has been partially supported by the European Union through the Research Training Network *SeGraVis*, by the Czech Ministry of Education under the project *Security-Oriented Research in Information Technology*, contract CEZ MSM 0021630528, and by the Czech Science Foundation under the contracts 102/07/0322 and 102/06/P076.

[2] Email: braione@disco.unimib.it

[3] Email: denaro@disco.unimib.it

[4] Email: krena@fit.vutbr.cz

[5] Email: pezze@disco.unimib.it

©2008 Published by Elsevier Science B. V.

1 Introduction

One of the long-standing challenges of the software industry is ensuring that a software system does not contain fatal errors after it is shipped. To this end, software industry is considering *formal* software verification techniques (e.g. [2]). Formal verification aims to prove that a software artifact is compliant with user-provided specifications of correct behaviour, and can thus complement testing in increasing confidence in the software operation.

The success of bytecode languages opens new opportunities for the practical applicability of formal software verification techniques. Bytecode languages are at a very desirable degree of abstraction for formal analysis. Bytecode is unaware of the diverse, semantically rich constructs that high-level programming languages offer to developers, allowing their analysis by means of a common simpler language. Bytecode can be formally analyzed even in the partial or total absence of source code when third-party libraries and components are used. Yet, it is at a sufficiently high level of abstraction to avoid issues like memory management and pointers that complicate analysis at the binary level.

In this paper we propose a formal analysis methodology for Java bytecode which harmonizes two well-known techniques, *model checking* and *symbolic execution*. Model checking arose as an automatic approach for verifying properties expressed as temporal logic formula on finite state systems. Model checking does a property-guided exploration of the system state space until the property is proved or disproved, producing in the latter case a counterexample execution trace. Symbolic execution executes the program replacing concrete variable values with symbolic ones, and builds a representation of the state space suitable for analysis. We propose an approach that uses symbolic execution for building a model of the program state space, and model checking for verifying temporal properties on it. To reduce the portion of the state space model explored by the checker, we build it on-the-fly as model checking of the temporal property proceeds.

The synergy of the two techniques brings considerable advantages [14]. Model checking provides an expressive language for specifying software properties, and efficient algorithms for verifying them automatically. Symbolic execution brings two key advantages. First, it produces a compact representation of the program state space where large set of numeric values are represented by symbols, potentially limiting state explosion during analysis. Second, by initializing software modules with symbolic values it allows to analyze them in isolation, thus fostering modular verification. However, while recently there has been a renewed interest in using symbolic execution for software verification [14,21,1,8,19], we are not aware of previous research works on this issue in the context of formal verification based on symbolic execution.

The original contribution reported in this paper are:

- A framework for the analysis of temporal formula with arbitrary predicates on program variables;
- A technology demonstrator composed by an on-the-fly linear-time temporal logic model checker and a prototype virtual machine which does symbolic execution of bytecode without relying on code translation or instrumentation;

- A preliminary investigation of the computational impact of the proposed analysis by applying our demonstrator to the analysis of a safety-critical software system.

Our research is grounded on previous research performed by some of the authors on symbolic execution based software verification [6].

The paper is organized as follows. Section 2 describes the overall framework for symbolically executing bytecode. Section 3 describes how to adapt model checking to bytecode verification. Section 4 reports the architecture of the technology demonstrator used for assessing the proposed approach and some preliminary experience on the verification of the TCAS aircraft collision avoidance system. Section 5 grounds our research effort in the current literature. Section 6 concludes the paper with some final remarks and an overview of our current and future research endeavours.

2 Symbolic execution for bytecode analysis

Symbolic execution [15,5] is a well known technique to execute programs with *symbolic* input values. The execution state stores a predicate, called *path condition*, that keeps track of the constraints on symbolic values and determines the control flow path yielding the state. For example, when a conditional jump instruction is executed, a decision procedure is invoked to determine if the jump condition and its negation are satisfiable under the current path condition. If both are satisfiable, the current state is “split” into two next states, and the path condition of each is augmented with the clause determining which branch is executed.

Symbolic execution has been traditionally applied to languages with numeric types. When considering languages that allow references and dynamic memory, symbolic execution can be extended with *lazy initialization* [14]. Lazy initialization calculates the possible initial values of a symbolic reference only when it is first used during execution, then it creates a state for each possible concrete value: `null`, a reference to a fresh object (assumed to be initially available in the heap but not yet used), a reference to each type-compatible object introduced by previous lazy initialization of other symbolic references. Considering all possible values on first use may generate a high number of states in later phases of the symbolic execution, thus variants have been proposed to further delay the set of values to be considered in order to reduce state explosion [9].

Intuitively, symbolically executing a method is a way to analyze the method invocation from a generic execution state. The initial state of symbolic execution represents the assumptions on the context of the method invocation. In our approach, the least assumption is an initial state containing only the *root* object, i.e., the receiver (`this`) of the method invocation, with all fields and method invocation parameters initialized to symbolic values and the path condition set to `true`. Preconditions on scalar variables, if any, are included in the path condition as predicates on scalar symbols. Preconditions on references are represented by adding objects to the heap and replacing symbolic references with concrete ones.

The symbolic initialization of static class members requires particular care. According to the Java Virtual Machine specification, initialization of static class members occurs when classes are first loaded, and causes the execution of the class’ static initializer. As a consequence, the initial state of symbolic execution depends on the assumption on whether or not each class is already loaded. By default we assume all classes as pre-loaded (i.e., we assume that static members have already been initialized and potentially modified before the current symbolic execution), which leads to conservatively initialize static members to fresh symbolic values. Users may configure the set of classes that must not be assumed as pre-loaded: such classes will be loaded on the first use, and their static class initializer will be then executed. For the sake of consistency, we always assume that the classes of the objects initially present in the heap are pre-loaded.

Our symbolic execution resolves symbolic references by lazy initialization when they are loaded on the operand stack. To ensure soundness, our implementation of lazy initialization resolves a symbolic reference only with type-compatible concrete references, and enforces the exclusion of non pre-loaded classes.

Our symbolic executor offers full support to a generic verification procedure[6] for exploring the symbolic state space in an arbitrary visiting order, and for evaluating user-specified predicates on the explored states.

3 Verifying LTL Properties

Temporal logics have been widely used for specifying properties of reactive systems for model checking [4,12]. *Linear-time temporal logic* (LTL [17]) is generally recommended for model checking frameworks based on explicit exploration of the state space. In this paper, we use LTL to specify properties of Java programs.

Informally, an LTL formula is built up from atomic propositions (which evaluate to either *true* or *false* in a given state), logic connectives ($\neg$, $\vee$, $\wedge$, $\rightarrow$), and *temporal operators* (**X** for next, **U** for until, **R** for release, **F** for eventually, **G** for globally). Temporal operators model relations between states of a linear temporal sequence (aka a path). A system satisfies a property described as an LTL formula if the property holds for all possible paths in the system. The full semantics of LTL is out of the scope of this paper; the interested reader is referred to [17] for details. Here we only notice that checking an LTL formula by incrementally evaluating its sub-formulae is possible, but it is more convenient to translate the formula to an automaton (namely a Büchi automaton) that is then used as an operational model to check whether the paths in the state space satisfy the formula [20].

To deal with symbolic states we must adapt traditional model checking algorithms. While evaluating atomic propositions in concrete states always results in either *true* or *false*, evaluating atomic propositions in symbolic states may lead to ambiguity, since symbolic states can stand for sets of concrete states including states with different evaluations. Le us consider for instance the evaluation of the atomic proposition $x > 5$ in the symbolic state characterized by the path condition $x > 0$. Without further information $x > 0$ does not lead to a single truth value for $x > 5$, which can be either *false* or *true* for different states represented by the path condition. To deal with this case, we split the symbolic states according to the atomic propositions incorporated in the formula under verification: with reference

[6] not necessarily a model checker

to the above example, $x > 5$ evaluates to $false$ for the states $0 < x \leq 5$, while it evaluates to $true$ for $x > 5$. Splitting symbolic states assigns a unique truth value to atomic propositions in each new symbolic state, albeit increasing the size of symbolic state spaces.

Our on-the-fly model checker translates all LTL properties to corresponding Büchi automata, extracts the set of atomic propositions of the properties, configures the symbolic execution engine to split states according to these propositions, and then it starts the symbolic execution. The symbolic execution engine notifies changes in the evaluation of the atomic propositions that occur while traversing the program state space. Based on the notifications, the model checker checks on-the-fly compliance with the property automaton, guides the symbolic execution engine through the states to be explored, and terminates when it either identifies a counterexample (the property is refuted) or completes the exploration of the state space without finding counterexamples (the property is proved).

Model checking based on symbolic execution needs to determine the equivalence of symbolic states, which is an undecidable problem. We overcome this problem by avoiding state matching, and producing tree-shaped state spaces. In this way, we simplify model checking, but we introduce multiple computations of the same symbolic states (as many times as they are reached), and we may produce infinite state spaces in presence of loops. We avoid infinite state spaces by limiting the depth of the tree to be explored, thus admitting unsound verification (that is, property violations cannot be excluded even if the analysis terminates without producing a counterexample). Repeating model checking of a property with increasing depth of explored state space until the examined property is refuted or proved, would eliminate unsoundness but would not guarantee that the model checking process terminates.

We notice that, in absence of state matching, we can only check a subset of LTL, which includes safety properties, but does not include liveness properties [20]. Visser et al. propose subsumption checking to compare symbolic states regardless of the general undecidability of their equivalence [21].

4 Preliminary Evaluation

We evaluated the approach with a prototype tool that analyzes temporal properties on Java programs. Here we draft the architecture of the prototype and report some preliminary experience on verifying a component of TCAS, an aircraft collision avoidance software system.

4.1 Prototype

Figure 1 illustrates the logical structure of the prototype, which consists of three software modules: a symbolic executor for Java bytecode, a linear temporal logic translator, and a model checker.

The Java Bytecode Symbolic Executor (JBSE from hence) implements the Java Virtual Machine Specification v.2. JBSE supports symbolic values without the need to instrument the source code. It can be customized to explore the symbolic

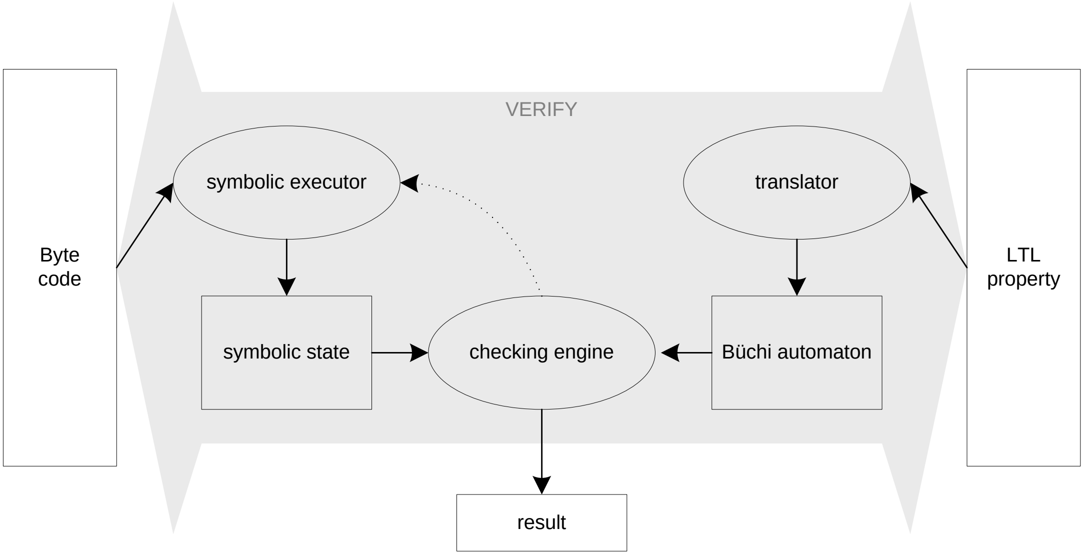

Fig. 1. The logical structure of the prototype

state space with different visiting orders (currently we use depth-first visit), and can backtrack to a previous saved state. Users can provide assumptions to drive symbolic execution, and can specify the concrete classes that must be used to lazily initialize symbolic references by means of regular expressions. JBSE relies on external decision procedures to decide the satisfiability of the path condition and prune unfeasible states (currently the prototype supports CVC3 and the Sicstus CLP environment). Among the features for supporting formal bytecode verification, JBSE offers a special "don't care" value, which is always expanded to both true and false each time it is evaluated, and can be used to model nondeterminism. JBSE offers methods to set backtrack points at any time of the execution, to register listeners to variable value changes, and to directly access the execution state and observe all the variables.

Translation from LTL formulae to Büchi automata is a well studied problem and several translators are available. We exploit the SPIN model checker v. 4.2.9 [12] run with the option `-f` or `-F` and the LTL2BA v. 1.1 tool [10]. Both tools produce Büchi automata described in Promela, the input language of the SPIN model checker.

The model checking engine guides the on-the-fly state space generation performed by the symbolic executor and at the same time steps the Büchi automaton. As soon as a counterexample is found this is returned and model checking is stopped. If no counterexample is found, the model checker reports whether the explored portion of the state space was sufficient or not to refute the property—i.e., whether the analysis was or was not sound. In the latter case it is possible that a greater exploration depth bound would make possible deciding the property. The model checker therefore outputs the path conditions of the branches abandoned for having reached the maximum exploration depth. This allows the user to repeat the analysis along these branches alone, with less effort than having to analyze the whole state space again.

The current implementation of JBSE requires users to manually inject atomic predicates as variables in the source code of the program under analysis. These

$$(\mathtt{\_UPSEPADQ} \wedge \neg\, \mathtt{\_DOWNSEPADQ}) \wedge \mathbf{F}(\mathtt{\_ATASTEN} \wedge \mathbf{F}\,\mathtt{\_ATASTDOWNRA}) \qquad \text{(PN1.1)}$$

$$\vee$$

$$(\neg\, \mathtt{\_UPSEPADQ} \wedge \mathtt{\_DOWNSEPADQ}) \wedge \mathbf{F}(\mathtt{\_ATASTEN} \wedge \mathbf{F}\,\mathtt{\_ATASTUPRA}) \qquad \text{(PN1.2)}$$

Fig. 2. PN1 and its subformulas

variables can be observed by the state observation mechanisms provided by JBSE.

4.2 Case study: TCAS

TCAS is an on-board aircraft conflict detection and resolution system used by US commercial aircraft [18], which has been widely studied in academia as a benchmark for safety critical applications [3,16,6]. The experimental work described in this section focuses on the component of TCAS that is responsible for finding the best *Resolution Advisory* (RA) suggesting to the pilot to either climb or descend, according to the relative position and speed of a nearby aircraft. The component, originally written in C, has been the subject of previous software engineering experiments [13,6]. We considered the Java equivalent of this component and replicated the analysis of the properties referred in [6].

Our analysis of TCAS considered the safety properties described in [6], and which we fully report in Appendix A. All the properties state some desirable features of the verified TCAS component in terms of the possible occurrence of a maneuver action under some conditions on the input data. As an example, we consider the first property described in [6] stating that, whenever one maneuver does not provide adequate separation and the other does, the former is never selected. The model checker has been fed on the negation of this property, which we will indicate as PN1 (for the sake of simplicity also available in Figure A.2). This formula is composed by two subformulas, stating the occurrence of an erroneous descend (PN1.1) or climb (PN1.2) maneuver. Both subformulas, in turn, share a common structure where a precondition (left subexpression) is conjoined with a temporal formula stating the eventual execution of the program statement performing a climb or descend maneuver [7]. The temporal part has been kept intentionally similar to the original formulation in [6]. All the safety properties are identically structured and differ only for the precondition parts.

In the PN1 formula the elementary predicates _UPSEPADQ and _DOWNSEPADQ state that the separation resulting from, respectively, climbing or descending are adequate. The remaining predicates state that the symbolic execution is at the beginning of the escape maneuver computation (_ATASTEN), at the point where a climb maneuver is selected (_ATASTUPRA), or at the point where a descend maneuver is selected (_ATASTDOWNRA). Since none of the variables in TCAS express any of these predicates, they have been implemented in TCAS as observable boolean variables, one for each predicate, and are evaluated in all the program points where they may change their truth value. The most critical predicates are _UPSEPADQ and _DOWNSEPADQ, whose evaluation requires state splitting.

[7] A LTL formula $\mathbf{F}p$ states that a state is reachable where the p predicate is true.

Property	Outcome	Number of states		$\frac{\#MCSt}{\#SESt}$
		Classic symbolic execution ($\#SESt$)	Symbolic execution driven by Model checking ($\#MCSt$)	
PN1	True	5701	2156	37.8%
PN2	True	8617	2239	26.0%
PN3	False	10729	308	2.9%
PN4	False	4021	284	7.1%
PN5	True	4453	2981	66.9%

Table 1
Number of traversed states for TCAS verification

4.3 Results

We analyzed TCAS to estimate the potential saving of the combination of symbolic execution with model checking in terms of number of traversed states. We verified the five safety properties for TCAS proposed in [6]. The TCAS code that we analyzed does not contain loops, thus symbolic execution builds a finite representation of the state space. We analyzed each property both with classic symbolic execution and with symbolic execution combined with model checking. When analyzing the property with classic symbolic execution we inserted assertions in the code, to check if the property is verified, when analyzing the property with the combination of symbolic execution and model checking, we verified the LTL property as discussed in this paper.

Table 1 reports the results of our experiments for each property. The first column indicates the property, described in the appendix. The second column shows the verification result on the implementation of TCAS considered in our experiments[8]. The third and fourth columns give the number of states traversed when analyzing the property with symbolic execution only and with a combination of symbolic execution and model checking, respectively. The fifth column quantifies the advantages of the combination of symbolic execution and model checking over classic symbolic execution as the ratio between the number of states explored in the two cases, that represents the portion of the symbolic space state that must be explored by the model checker to prove or disprove the property.

As we can see, on-the-fly model checking explores a subset of the state space. The saving is higher when properties are not verified (properties PN2 and PN3) than when they are (properties PN1, PN4 and PN5), since in the former case the model checker stops early with a counterexample. When properties are not verified the model checker explored less than 10% of the state space. When properties are verified the model checker explores between 26% and 66.9%, still with a relevant

[8] In [6] the property PN3 was incorrectly reported as verified for TCAS.

saving.

We finally give some comments about performance. The analysis time is dominated by the time spent in the external satisfiability decision procedure (about the 96–98% of the total). This is in line with the fact that the analyzed TCAS code is essentially a sequence of conditional branches. With the Sicstus decision procedure the analysis speed ranges from 114 to 133 states/sec, yielding overall analysis times not exceeding 2 min. With the CVC3 decision procedure the analysis is considerably slower, exceeding a 60 min time budget with any of the properties. Currently we have no definitive explanation for this phenomenon. We conjecture that a meaningful performance improvement can be achieved by enhancing the JBSE-to-CVC3 interface module, which currently does not produce optimized CVC3 input. All the experiments were performed on a 1.6 GHz Centrino laptop with 512 MBytes of RAM.

5 Related Work

Automatic verification of programs is currently the subject of several research efforts. This section reports on related work distinguishing, for presentation purposes, between approaches that rely on symbolic execution to model the execution of a program, and approaches that extract finite models from source code and then rely on techniques for finite state verification.

Symbolic execution-based verification

Several research prototypes for testing and verification of Java programs use symbolic execution to explore program state spaces, for different purposes: JPF-SE checks program states against logic assertions and can generate test cases for the traversed paths [14,21,1]. Bogor/Kiasan checks class methods against JML specifications [8]. Tomb, Brat and Visser's prototype checks for occurrence of unhandled runtime exceptions [19].

None of the above prototypes symbolically executes bytecode directly. They either translate bytecode to some checkable language (for instance, Bogor/Kiasan translates bytecode to BIR, which is then symbolically executed), or rely on the user to handle symbolic values and heap analysis at the source code level (as in JPF-SE). Our prototype executes the bytecode through a suitable JVM that provides native support for symbolic values and heap analysis at the bytecode level. Thus, our approach does not face any semantic mismatch that may derive from translation between non semantically equivalent languages, and avoids the burden on programmers of instrumenting the code and handling lazy initializations.

Extraction of finite state models from source code

The Bandera toolkit supports LTL model checking of Java programs [7]. Bandera provides a set of analysis and transformation components that extract finite state models from the program source code, leverages finite-state model checkers (such as SPIN and SMV) to perform LTL verification on the models, and maps the results back to the original source code. The main disadvantage of Bandera comes from the semantic gap between the modeling languages of the model checkers and the Java programming language, which in several cases hinders the possibility of automatically extracting the finite state models.

Henzinger et al. and Ball and Rajamani describe approaches to verifying temporal properties of programs by exploiting predicate abstractions of programs [11,2]. Predicate abstraction works very well for abstracting program control flows and can deal with infinite paths better than symbolic execution. However, there is not a really convenient abstraction that accounts for all possible dependencies between program references and heap structures. Symbolic execution in combination with lazy initialization allows for a thorough analysis of the dependencies between the heap and the program state space, which are extremely important when analyzing bytecode.

6 Conclusions

Model checking can provide formal analysis of critical properties of programs, and successfully complement program testing for systems with high quality software requirements. Model checking of programs is challenging because programs are usually defined over infinite input domains, which cannot be directly dealt by traditional model checking procedures for finite state systems. Symbolic execution has the potential to contrast this problem by providing finite representations of potentially infinite sets of the numeric executions of a program.

In this paper, we have presented our approach to LTL model checking of Java programs based on symbolically executing the programs at the bytecode level. We see clear advantages in addressing the analysis at the bytecode level: bytecode provides a nicer level of abstraction than both source and binary code, is available even in partial absence of source code, and brings in the well known advantages of platform- and language-independency. We have described a prototype that implements our approach, and we have reported the results of a preliminary experience with the analysis of a safety-critical application. The initial results support our belief that the interplay of model checking and symbolic execution can provide suitable tradeoffs for the portion of program state space that needs to be explored to analyze LTL properties.

Our approach differs from the other existing approaches that address verification by means of the combination of model checking and symbolic execution. First, the other approaches that we are aware of, do not target verification of LTL properties: they either target generic properties (such as absence of runtime exceptions), or verify pre- and post-conditions at the boundaries of program modules. LTL allows for specifying and checking interprocedural relations between modules that involve relations on the program variables, and thus can capture the critical properties that are specific to a software system. Furthermore, most existing approaches do not symbolically execute bytecode directly: they either rely on translations to checking-oriented languages (which can be the source of semantic mismatches) or require the manual instrumentation of symbolic values and lazy initialization in the source code (which can be burdensome and error prone).

Our future research agenda includes several still open issues. Coping with infinity that arise from loop unfolding and from recursive data structures is a classical problem for symbolic execution, and strategies must be defined so that in most practical cases the analysis is able to produce an answer. Issues of analysis costs deserve further investigation: compared to other abstractions, symbolic execution builds a very precise model of the state space, but leads to techniques with higher computational cost than methods that analyze abstractions of the state space. Finally, the presence of symbolic values mandates the use of automatic decision procedures whose impact needs to be further assessed. Our long-term goal is gaining a better understanding on the assumptions and tradeoffs on the above issues, such as to devise a practical methodology based on the approach proposed in this paper.

References

[1] Anand, S., C. S. Pasareanu and W. Visser, *JPF-SE: A symbolic execution extension to java pathfinder*, in: *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, Lecture Notes in Computer Science **4424** (2007), pp. 134–138.

[2] Ball, T. and S. K. Rajamani, *Automatically validating temporal safety properties of interfaces*, Lecture Notes in Computer Science **2057** (2001), pp. 103–22.

[3] Chan, W., R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin and J. Reese, *Model checking large software specifications*, ACM SIGSOFT Software Engineering Notes **21** (1996), pp. 156–166.

[4] Clarke, E. N., E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 244–263.

[5] Clarke, L., *A system to generate test data and symbolically execute programs*, IEEE Transactions on Software Engineering **2** (1976), pp. 215–222.

[6] Coen-Porisini, A., G. Denaro, C. Ghezzi and M. Pezzè, *Using symbolic execution for verifying safety-critical systems*, in: *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2001)*, 2001, pp. 142–151.

[7] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zheng, *Bandera: extracting finite-state models from java source code*, in: *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, 2000, pp. 439–448.

[8] Deng, X., J. Lee and Robby, *Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems*, in: *Proceedings of the 21st International Conference on Automated Software Engineering (ASE 2006)*, 2006, pp. 157–166.

[9] Deng, X., Robby and J. Hatcliff, *Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs*, in: *5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, 2007, pp. 273–282.

[10] Gastin, P. and D. Oddoux, *Fast LTL to Büchi automata translation*, in: G. Berry, H. Comon and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, Lecture Notes in Computer Science **2102** (2001), pp. 53–65.
URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/Cav01go.ps

[11] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in: *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)* (2002), pp. 58–70.

[12] Holzmann, G. J., *The model checker spin*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[13] Hutchins, M., H. Foster, T. Goradia and T. Ostrand, *Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria*, in: *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy*, 1994, pp. 191–200.

[14] Khurshid, S., C. S. Pasareanu and W. Visser, *Generalized symbolic execution for model checking and testing*, in: *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, Lecture Notes in Computer Science **2619** (2003), pp. 553–568.

[15] King, J., *Symbolic execution and program testing*, Communications of the ACM **19** (1976), pp. 385–394.

[16] Leveson, N., M. Heimdahl, H. Hildreth and J. Reese, *Requirements specification for process-control systems*, IEEE Transactions on Software Engineering **20** (1994), pp. 684–707.

[17] Pnueli, A., *The temporal logic of programs*, in: *FOCS*, 1977, pp. 46–57.

[18] RTCA, *Minimum operational performance standards for traffic alert and collision avoidance system (TCAS) airborne equipment*, Technical Report DO-185, Radio Technical Commission for Aeronautics (1990).

[19] Tomb, A., G. P. Brat and W. Visser, *Variably interprocedural program analysis for runtime error detection*, in: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)* (2007).

[20] Vardi, M. Y., *An automata-theoretic approach to linear temporal logic*, in: *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata* (1996), pp. 238–266.

[21] Visser, W., C. S. Pasareanu and R. Pelanek, *Test input generation for java containers using state matching*, in: *Proceedings of the International symposium on Software testing and analysis (ISSTA 2006)* (2006), pp. 37–48.

A TCAS proprerties

A.1 The TCAS code

The experimental work focuses on the component of TCAS that is responsible for finding the best RA. The component, which consists of 120 lines of code, comes from a set of programs used by Hutchins et al. in a previous experiment [13]. Figure A.1 shows the code of procedure `alt_sep_test()`, which is referred to in all the LTL properties taken into account.

```
      public int alt_sep_test() {
         boolean enabled, tcas_equipped, intent_not_known;
         boolean need_upward_RA, need_downward_RA;
         int alt_sep;

ASTBeg:  enabled          =    High_Confidence
                            && (Own_Tracked_Alt_Rate <= OLEV)
                            && (Cur_Vertical_Sep > MAXALTDIFF);
         tcas_equipped    =    (Other_Capability == TCAS_TA);
         intent_not_known =    Two_of_Three_Reports_Valid
                            && (Other_RAC == NO_INTENT);
         alt_sep          = UNRESOLVED;
         if (   enabled
             && ((tcas_equipped && intent_not_known) || !tcas_equipped)) {
ASTEn:     need_upward_RA   =    Non_Crossing_Biased_Climb()
                              && Own_Below_Threat();
           need_downward_RA =    Non_Crossing_Biased_Descend()
                              && Own_Above_Threat();
           if      (need_upward_RA)
ASTUpRA:       alt_sep = UPWARD_RA;
           else if (need_downward_RA)
ASTDownRA:     alt_sep = DOWNWARD_RA;
           else
ASTUnresRA:    alt_sep = UNRESOLVED;
         }

         return alt_sep;
      }
```

Fig. A.1. The code of `alt_sep_test()`

The vertical separation between the two airplanes is represented by the global variable `Cur_Vertical_Sep`, while `Up_Separation` and `Down_Separation` represent the estimated vertical separation after a climbing maneuver and a descending maneuver, respectively. `Own_Tracked_Alt` and `Other_Tracked_Alt` represent the altitudes of the airplanes. These variables can be regarded as input data for the analyzed sub-system. The vertical separation at the closest point of approach is considered to be *adequate* if it is greater than a threshold value (`Positive_RA_Alt_Thresh`), which can be viewed as a system constant. The code contains the following five labels:

- `ASTBeg`, which identifies the beginning of the code;
- `ASTEn`, which identifies the first statement used for compute the best escape maneuver;
- `ASTUpRA`, which identifies the statement where a climbing RA is selected;
- `ASTDownRA`, which identifies the statement where descending RA is selected;
- `ASTUnresRA`, which identifies the statement where no RA is selected.

A.2 Atomic predicates

This subsection defines the atomic predicates necessary for defining the LTL verification formulas.

- `_ATASTEN`: the next statement is `ASTBeg`;
- `_ASTUPRA`: the next statement is `ASTUpRA`;
- `_ASTDOWNRA`: the next statement is `ASTDownRA`;
- `_UPSEPADQ` $\overset{\text{def}}{=}$ (`Up_Separation` >= `Positive_RA_Alt_Thresh`):
  `Up_Separation` is adequate;
- `_DOWNSEPADQ` $\overset{\text{def}}{=}$ (`Down_Separation` >= `Positive_RA_Alt_Thresh`):
  `Down_Separation` is adequate;
- `_UPSEPBEST` $\overset{\text{def}}{=}$ (`Up_Separation` > `Down_Separation`):
  the estimated separation for a climbing maneuver is better than that for a descend maneuver;
- `_DOWNSEPBEST` $\overset{\text{def}}{=}$ (`Up_Separation` < `Down_Separation`):
  the estimated separation for a descend maneuver is better than that for a climb maneuver;
- `_OWNOVER` $\overset{\text{def}}{=}$ (`Own_Tracked_Alt` > `Other_Tracked_Alt`):
  the maneuvering aircraft is above the approaching aircraft;
- `_OTHEROVER` $\overset{\text{def}}{=}$ (`Own_Tracked_Alt` < `Other_Tracked_Alt`):
  the maneuvering aircraft is below the approaching aircraft.

A.3 Verified properties

This section reports the five TCAS safety properties and the associated LTL formulas used for checking them.

Property 1: Safe Advisory Selection
Meaning of the property: If one maneuver produces adequate separation and the other does not, then the RA corresponding to the maneuver that does not produce adequate separation is not issued. Checked by LTL formula PN1.

$$(\texttt{\_UPSEPADQ} \land \neg\,\texttt{\_DOWNSEPADQ}) \land \mathbf{F}(\texttt{\_ATASTEN} \land \mathbf{F}\texttt{\_ATASTDOWNRA}) \qquad \text{(PN1.1)}$$

$$\lor$$

$$(\neg\,\texttt{\_UPSEPADQ} \land \texttt{\_DOWNSEPADQ}) \land \mathbf{F}(\texttt{\_ATASTEN} \land \mathbf{F}\texttt{\_ATASTUPRA}) \qquad \text{(PN1.2)}$$

Fig. A.2. PN1 and its subformulas

Property 2: Best Advisory Selection
Meaning of the property: Let neither climb nor descent maneuvers produce adequate separation. Then, the RA corresponding to the maneuver that produces less separation is never issued. Checked by LTL formula PN2.

$$(\neg\ \_\mathtt{UPSEPADQ} \wedge \neg\ \_\mathtt{DOWNSEPADQ} \wedge \_\mathtt{UPSEPBEST}) \wedge$$

$$\mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTDOWNRA}) \tag{PN2.1}$$

$$\vee$$

$$(\neg\ \_\mathtt{UPSEPADQ} \wedge \_\mathtt{DOWNSEPADQ} \wedge \_\mathtt{DOWNSEPBEST}) \wedge$$

$$\mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTUPRA}) \tag{PN2.2}$$

Fig. A.3. PN2 and its subformulas

Property 3: Avoid Unnecessary Crossing
Meaning of the property: If both climbing and descending produce adequate separation, then a crossing RA is never issued. Checked by LTL formula PN3.

$$(\_\mathtt{UPSEPADQ} \wedge \_\mathtt{DOWNSEPADQ} \wedge \_\mathtt{OWNOVER}) \wedge \mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTDOWNRA}) \tag{PN3.1}$$

$$\vee$$

$$(\_\mathtt{UPSEPADQ} \wedge \_\mathtt{DOWNSEPADQ} \wedge \_\mathtt{OTHEROVER}) \wedge \mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTUPRA}) \tag{PN3.2}$$

Fig. A.4. PN3 and its subformulas

Property 4: No Crossing Advisory Selection
Meaning of the property: A crossing RA is never issued. Checked by LTL formula PN4.

$$(\_\mathtt{OWNOVER}) \wedge \mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTDOWNRA}) \tag{PN4.1}$$

$$\vee$$

$$(\_\mathtt{OTHEROVER}) \wedge \mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTUPRA}) \tag{PN4.2}$$

Fig. A.5. PN4 and its subformulas

Property 5: Optimal Advisory Selection
Meaning of the property: The RA that produces less separation is never issued. Checked by LTL formula PN5.

$$(\_\mathtt{UPSEPBEST}) \wedge \mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTDOWNRA}) \tag{PN5.1}$$

$$\vee$$

$$(\_\mathtt{DOWNSEPBEST}) \wedge \mathbf{F}(\_\mathtt{ATASTEN} \wedge \mathbf{F}\_\mathtt{ATASTUPRA}) \tag{PN5.2}$$

Fig. A.6. PN5 and its subformulas