Journal of Logical and Algebraic Methods in Programming 85 (2016) 707–736

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Debugging Maude programs via runtime assertion checking and trace slicing ☆

María Alpuente [a], Demis Ballis [b], Francisco Frechina [a], Julia Sapiña [a,*]

[a] DSIC-ELP, Universitat Politècnica de València, Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
[b] DIMI, University of Udine, Via delle Scienze, 206, 33100, Udine, Italy

ARTICLE INFO

Article history:
Received 26 March 2015
Received in revised form 10 March 2016
Accepted 10 March 2016
Available online 16 March 2016

Keywords:
Trace slicing
Runtime checking
Dynamic program slicing
Program diagnosis and debugging
Rewriting logic
Maude

ABSTRACT

In this paper we propose a dynamic analysis methodology for improving the diagnosis of erroneous Maude programs. The key idea is to combine runtime checking and dynamic trace slicing for automatically catching errors at runtime while reducing the size and complexity of the erroneous traces to be analyzed (i.e., those leading to states failing to satisfy some of the assertions). First, we formalize a technique that is aimed at automatically detecting deviations of the program behavior (symptoms) with respect to two types of user-defined assertions: functional assertions and system assertions. The proposed dynamic checking is provably sound in the sense that all errors flagged are definitely violations of the specifications. Then, upon eventual assertion violations we generate accurate trace slices that help identify the cause of the error. Our methodology is based on (i) a logical notation for specifying assertions that are imposed on execution runs; (ii) a runtime checking technique that dynamically tests the assertions; and (iii) a mechanism based on (equational) least general generalization that automatically derives accurate criteria for slicing from falsified assertions. Finally, we report on an implementation of the proposed technique in the assertion-based, dynamic analyzer ABETS and show how the forward and backward tracking of asserted program properties leads to a thorough trace analysis algorithm that can be used for program diagnosis and debugging.

1. Introduction

Program debugging is crucial to reliable software development because the size and complexity of modern software systems make it almost impossible to avoid errors in their (requirements and design) specifications. Unfortunately, debugging is generally a burdensome process that takes up a large portion of the software development effort, with developers painfully going through volumes of execution traces to locate the actual cause of observable misbehaviors. In order to mitigate the costs of debugging, automated tools and techniques are required to help identify the root cause of (anticipated) errors. In this paper, we propose a general approach for the debugging of programs

☆ This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN2015-69175-C4-1-R and TIN2013-45732-C4-1-P, and by Generalitat Valenciana Ref. PROMETEOII/2015/013. F. Frechina was supported by FPU-ME grant AP2010-5681, and J. Sapiña was supported by FPI-UPV grant SP2013-0083 and mobility grant VIIT-3946.

* Corresponding author.
E-mail addresses: alpuente@dsic.upv.es (M. Alpuente), demis.ballis@uniud.it (D. Ballis), ffrechina@dsic.upv.es (F. Frechina), jsapina@dsic.upv.es (J. Sapiña).

http://dx.doi.org/10.1016/j.jlamp.2016.03.001
2352-2208/

that is based on systematically combining runtime assertion checking and automated trace (and program) simplification.

Assertion checking is one of the most useful automated techniques available for detecting program faults. In runtime assertion checking, assertions are traditionally used to express conditions that should hold at runtime. By finding inconsistencies between specified properties and the program code, dynamic assertion checking can prove that the code is incorrect. Moreover, since an assertion failure usually reports an error, the user can direct his attention to the location at which the logical inconsistency is detected and (hopefully) trace the errors back to their sources more easily. Runtime assertion checking can also be useful in finding problems in the specifications themselves, which is important for keeping the specifications accurate and up-to-date. Although not universally used, assertions seem to have widely infiltrated common programming practice, primarily for finding bugs in the later stages of development. A brief history of the research ideas that have contributed to the assertion capabilities of modern programming languages and development tools can be found in [1].

Program slicing [2,3] is another well-established activity in software engineering with increasing recognition in error diagnosis and program comprehension since it allows one to focus on the code fragment that is relevant to the piece of information (known as slicing criterion) that we want to track from a given program point. The basic idea of program slicing is to isolate a subset of program statements that either (i) contribute to the values of a set of variables at a given point or (ii) are influenced by the values of a given set of variables. The first approach corresponds to forms of backward slicing, whereas the second one corresponds to forward slicing. Work in this area has focused on the development of progressively more effective, useful, and powerful slicing techniques, which have been transferred to many application areas including program testing, software maintenance, and software reuse.

In order to cope with very complex distributed systems, tools and methods that can improve the early specification are key to the system development effort. Maude [4] is a high-performance language and system that provides a powerful variety of correctness tools and techniques including prototyping, state space exploration, and model checking of temporal formulas. Maude programs correspond to specifications in rewriting logic (RWL) [5], which is an extension of membership equational logic [6] that, besides supporting equations and allowing the elements of a type or *sort* to be characterized by means of membership axioms, adds rewrite rules that can be non-deterministic in order to represent transitions in a concurrent system. Thanks to its reflective design and meta-level capabilities, the Maude system provides powerful and highly efficient meta-programming facilities. This has further contributed to its success, giving support to the development of sophisticated tools and techniques for the modeling and analysis of Maude specifications, such as LTLR model checking [7], abstract certification [8], Web verification [9,10], narrowing-based code-carrying theory [11], *etc.* (for a survey of the related literature, see [12]).

The use of slicing for debugging Maude programs is discussed in [13], and relies on a rich and highly dynamic parameterized scheme for exploring rewriting logic computations defined in [14,15] that can significantly reduce the size and complexity of the runs under examination by automatically slicing both programs and computation traces. However, Maude does not currently provide general support for asserting properties that are dynamically-checked. Hence, the aim of this work is to provide Maude with runtime assertion-checking capabilities by first introducing a simple assertion language that suffices for the purpose of improving error diagnosis and debugging in the context of rewriting logic. We follow the approach of modern specification and verification systems such as Spec$^\sharp$ or the Java Modeling Language (JML) where the specification language is typically an extension of the underlying programming language and specifications are used as *contracts* that guarantee certain properties to hold at a number of execution states (e.g., before or after a given function call [16]). We believe that this choice of a language is of practical interest because it facilitates the job of programmers. Even if Maude is a highly declarative language that supports a programming style where no conceptual difference exists between programs and high-level specifications, there can be good reasons not to use the code itself as a contract. Assertions can be seen as a form of lightweight, possibly incomplete or weaker specification embedded in the program text that may help developers identify program properties or behaviors to be preserved when modifying code. Independent assertions can also improve the effectiveness of tests, can be used as contracts to check the conformance of an implementation to its formal specification, and are key for static verification and automated test case generation. During the design process, they can simulate a design, allowing one to explore its properties before committing to the long development process. The advantages of equipping software with assertions are extensively discussed in [17].

In our framework, if an assertion evaluates to false at runtime, an assertion failure results, which typically causes execution to abort while delivering a huge execution trace. By automatically inferring deft slicing criteria from falsified assertions, we derive a self-initiating, enhanced dynamic slicing technique that automatically starts slicing the trace backwards at the time the assertion violation occurs, without having to manually determine the slicing criterion in advance. As a by-product of the trace slicing process, we also derive a dynamic program slice that preserves the program behavior for the considered program inputs [2]. In the proposed approach, assertions are *external* and evaluated at runtime whenever the state associated with the assertion is reached during execution. This use of assertions involves checking *individual* (finite) program executions as well as non-deterministic execution trees (up to a finite depth), rather than proving (or disproving) the correctness of every program execution.