

Invariant Checking – Java Examples Handbook

July 29, 2008

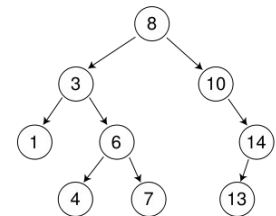
These artifacts are Java examples targeted to evaluation of Java symbolic execution and test case generation tools such as Kiasan, JPF, and Jcute. Each example is typically one class (in some cases, auxiliary classes are included) with invariant specifications given in the form of executable Java methods (boolean functions).

If you use these examples in your research, we would appreciate it if any resulting papers could cite one or both of the papers listed in the **References** below – in addition to citing the SIR repository.

I. Motivation

I.1 Contract Checking in Object-Oriented Languages

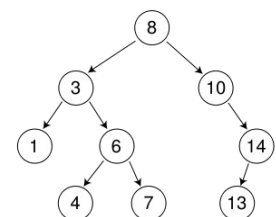
In development contexts that emphasize reusable components, it is important for development methodologies and processes to be supported by the following capabilities: (1) specification notations such as those used in the Design-by-Contract (DBC) paradigm that provides “software contracts” to specify the assumptions that a component makes about its context and the behavior/functionality of the services that the component guarantees to provide to clients, and (2) tools for automatically checking that clients conform to component contract assumptions and that a component's implementation provides functionality that satisfies what its contract guarantees. These capabilities can be difficult to provide in OOP languages due to the extensive use of dynamically created heap objects, where one has to be able to precisely reason about objects, their data, and the relationships between them. We refer to invariants and functional behavioral specifications on complex heap data structures as *strong* properties as they are hard to analyze due to aliasing issues (e.g., equivalence of object structures); *lightweight* properties such as simple relationships between scalar values and variable null-ness are a counterpart of strong properties that are also important to specify and enforce as an integral part of the development process.



Tools such as ESC/Java that are founded on theorem-proving techniques have made significant contributions in the area of automated contract checking. However, ESC/Java and related tools have significant difficulties in supporting checking of strong properties of heap-manipulating programs; they provide weak support for generation of informative counter-examples, and they lack integration with existing quality assurance mechanisms such as testing.

I.2 Unit Testing in Object-Oriented Languages

Unit testing -- a quality assurance method in which individual software system modules (i.e., *units*) are tested in isolation, is an integral part of many development processes. Successful unit testing requires several important steps.



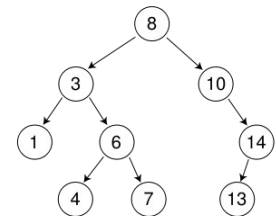
First, a suite of *unit tests* must be created that call the public methods of the unit with method input parameter values that drive the execution of the unit down particular execution paths. The number of unit tests and choice of parameter values are usually driven by multiple factors including: (1) the need to cover all the different combinations of values that might get passed into the unit when it is deployed within the context of a complete system (i.e., *context value coverage*), and (2) the need to obtain a certain level of *code coverage* for the unit (e.g., certain levels of statement or branch coverage). Factor (1) may be aided by documentation about a unit's required behavior or by formal or informal unit interface contract specifications (e.g., such as those that result for a Design-By-Contract methodology). In the presence of such specifications, the test suite construction goals associated with (1) can be presented as aiming to achieve a certain level of specification coverage.

Second, one must design a collection of *test oracles* that can be invoked to automatically determine if the output of a test is correct. As with (1) above, the construction of oracles can be guided by the presence of documentation or contract specifications (e.g., the *post-condition* of a method contract can often be directly translated into a test oracle).

Third, because units are tested in isolation, some sort of executable behavior must be defined for any method invocations that the unit makes into the enclosing context (we view such units as having incomplete computation structures -- open systems). Such behavior is often implemented in what are termed *mock objects* -- dummy objects with method stubs that implement fragments of the functionality that exists in the *real objects* that the unit interacts with when deployed in a system context. The partial functionality of the mock object might: (a) use assertions to check for properties of parameter values flowing from the unit into the mock object, (b) simulate side effects that might occur during the execution of the context method, (c) generate simulated method return values that get passed from the context back into the unit. Methodologies for constructing mock objects are very ad-hoc (e.g., there is hardly ever an attempt to relate the behaviors of mock and real objects as over/under-approximations), and the degree to which the functionality of real object is captured by the mock object varies widely.

I.3 Symbolic Execution for Contract Checking and Unit Test Case Generation

Recent efforts have demonstrated that symbolic execution can serve as a basis for checking strong contract properties and invariants of complex heap-based data structures and for supporting automated test case generation. One key advantage of symbolic execution over real/concrete execution (e.g., traditional testing) is that one can avoid the burden of constructing numerous concrete input parameter values and instead use symbolic values and constraints to compactly represent sets of possible input values. Once the symbolic execution algorithm has traversed a path in a method and accumulated symbolic constraints over a set of symbolic values, the constraints can be solved to find a concrete *representative* -- a binding of symbolic values to concrete values that satisfies the accompanying constraints, and this representative can be turned into a concrete test case for the method that drives execution along a particular path (the path walked by the symbolic execution algorithm when constructing the current collection of symbolic values and constraints).

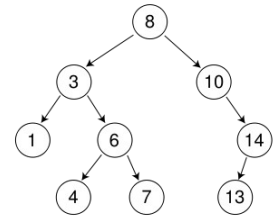


Because symbolic execution is typically implemented as a *path-sensitive analysis*, it can require significant computational resources. Thus, it is important to empirically validate the effectiveness of

symbolic execution for both contract checking and test case generation.

II. An Overview of the Artifacts

The artifacts in this collection were used to evaluate the Kiasan symbolic execution and unit test case generation algorithms presented in the references below. Some of the examples come from earlier papers on symbolic execution and test case generation; others come from Java collections library or from data structures textbooks.



The examples are organized into the following groups (corresponding to directories in the distribution files):

- Array Partition (Quicksort)
- Binary Heap
- Binary Search Tree
- Disjoint Set
- DLL (doublely-linked list)
- Red Black Tree
- Sort (a collection of five sorting algorithms)
- Stack
- Vector

Each of the examples above has two documentation files associated with it.

- Read Me file -- Describes the origin of the example as well as important features of the example.
- Code Samples file -- contains the "specifications" given in the form of code fragments with annotations describing the attachment of invariants.

The “specification” in each file are typically stated in terms of a “RepOK” method that encodes the invariant for data structure, along with some additional annotations recognized by the Kiasan tool. The Kiasan-specific annotations can easily be modified to match the input of another symbolic execution tool. We plan to eventually re-release these examples with the specifications encoded in the Java Modeling Language (JML)

III. Issues in Evaluation

The references below contain details regarding particular issues investigated in experiments with the artifacts. Generally the experiments aim to assess the computational cost (memory, time, number of internal states in the checking algorithm) associated with achieving a certain level of quality of the analysis (usually measured in some form of coverage – typically, branch coverage). Symbolic execution is usually presented as an *under-approximation* in which the analysis is bounded in some way by the depth of execution path explored or size of statespace (e.g., number of objects of a particular type).

We have enclosed in this distribution the extended version of the two papers below that give the details of the experiments carried out using these artifacts.

Acknowledgements:

The following people contributed to the construction of the code examples and documentation in this distribution: William Deng, Robby, Andrew King, Sam Procter, John Hatcliff.

References:

1. Xianghua (William) Deng, Robby, John Hatcliff. "Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems", Proceedings of the 2007 International Conference on Testing: Academic and Industrial Conference: Practice and Research Techniques, IEEE Press, September 2007.
2. Xianghua Deng, Robby, and John Hatcliff. "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs", Proceedings of the 2007 International Conference on Software Engineering and Formal Methods, September 2007.