Program Testing Using High-Level Property-Driven Models

Isabel Michiels[1], Coen De Roover[1], Johan Brichau[1,2], Elisa Gonzalez Boix[1], Theo D'Hondt[1]

[1]Programming Technology Lab
Vrije Universiteit Brussel, Belgium

[2]Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille, France

Abstract

Testing is a crucial part of the software development life cycle that necessitates adequate techniques and tools. On the one hand, unit or function testing techniques are particularly easy and lightweight to use but are restricted to the testing of the external behaviour of a program. On the other hand, testing techniques that use property-driven models of the software are able to test behavioural properties of the entire execution of a program. However these models are often specified in terms of low-level execution events of the program. In this paper, we present the lightweight property-driven testing platform called BEHAVE*, where tests are written as property-driven behavioural models over high-level run-time events. We validate our approach by testing behavioural properties related to the automatic memory management in the interpreter of the Pico programming language.*

1. Introduction

The increasing complexity and size of today's software systems provides ample opportunities for developers to introduce faulty and erroneous behaviour in the system's implementation. The presence of such *bugs* in many finished software products emphasizes the need for integrated testing in the development process. However, software testing is a laborious activity that is often still conducted without any adequate tools and thus consumes more than 50% of the total software development effort [4].

Model-oriented verification techniques [5] permit us to accurately verify the entire program's behaviour against its specification, but they are also time-consuming, complex and do not scale very well. This is because the entire behaviour of a software system must be specified in a complete and precise behavioural model of all execution states of that system.

On the other end of the spectrum, unit or function testing techniques like the XUnit open source testing frameworks [8](e.g. SUnit for Smalltalk, CUnit for C, etc.) are particularly lightweight. Using these techniques, developers focus on testing small parts of the system's implementation incrementally, based on particular usage scenarios [3]. Although these incremental approaches address the scalability problem and are relatively easy to use, they do not verify all aspects of the behaviour of the executing system. Instead, they can merely test the external behaviour of particular parts of a system.

Using property-driven models for testing combines some of the advantages of model-oriented verification and unit-based testing. They are lightweight in the sense that they only model particular *properties* of the program's execution. Moreover, in contrast to unit-based testing, they also test non-externally verifiable behaviour of the executing program with respect to the modeled properties. Nevertheless, property-driven testing does require the specification of behavioural models of these properties and these are often specified in terms of low-level execution events of the program.

In this paper, we present the property-driven testing platform BEHAVE and apply it to test behavioural properties of memory management in the Pico language interpreter. The particular contribution of BEHAVE is that the behavioural models are specified using a declarative formalism which renders the models machine-verifiable and, at the same time, understandable to the developers [11]. More specifically, BEHAVE offers support for testing particular behaviour of an application by specifying property-driven models *over* high-level run-time events *using* temporal logic programming.

We will start our paper by clarifying our case study of the Pico language interpreter and its automatic memory management in section 2. In section 3 we will introduce our platform called BEHAVE together with its underlying approach and we will demonstrate the use of our platform by testing garbage collection in Pico. We will end with related work in section 4 and a conclusion in section 5.

2. The Pico Interpreter and its Memory Model

Pico [10] is an interpreted programming language developed at the Vrije Universiteit Brussel. While it was originally conceived to teach programming concepts to students outside the realm of computer science, its interpreter implementation (which totals about 8000 lines of condense C code) is nowadays also used for teaching about programming languages, interpreters and automatic memory management. In this paper, we use the Pico interpreter as a case study to illustrate our testing approach. In particular we will test for erroneous behaviour related to automatic memory management.

The Pico memory consists of a heap that is managed by an automatic garbage collector. The garbage collection algorithm can be triggered every time a chunk of memory is requested. If no sufficient memory is available, the garbage collector will traverse the Pico environment and possibly defragment it. As a result, chunks can be moved to a completely different address in the Pico heap. This change of location happens transparently because the garbage collector also updates any references to that chunk. However, this requires that all references to that chunk are also stored on the heap, which is not always the case. In many parts of the Pico implementation, references to chunks of memory on the Pico heap need to be stored in a temporary variable inside a C function. More particularly this happens in so called 'continuation functions' which are C functions that represent a part of program execution in Pico. They can be recognized as functions with no return type and no parameters. If a garbage collect occurs during the execution of such a continuation function, those references to the Pico memory might become invalid and the Pico interpreter will crash.

Obviously, we want to detect such an unwanted behaviour through careful testing. This means that we need to detect the occurrence of a garbage collect in between the assignment and the use of a temporary variable that holds a reference to the Pico memory. Unit testing is insufficient because such a test only fails if a garbage collect actually defragments memory. Depending on the actual memory consumption and organisation, such tests might thus fail or not, although the erroneous behaviour of a *possible* garbage collect is always present. Therefore, we need to test the *possible* occurrences of a garbage collect in between the assignments and the uses of a temporary variable inside a C function. To this extent, we developed a property-driven model that is verified using the BEHAVE platform.

3. Property-Driven Testing with BEHAVE

In general, three important phases can be distinguished in application testing: specification of a model, verifying the model against the actual behaviour (running the test) and evaluating the test results. Using BEHAVE, developers specify property-driven models using a declarative programming language. These models express certain behavioural properties that must hold throughout an application's lifetime and they are automatically verified with respect to a high-level execution-trace of the program. This execution trace is *high-level* because it contains the application-specific events that are required to verify the model. These high-level events are also specified by the developer. Figure 1 provides a general overview of all constituents of this approach when applied to the testing of the garbage collector in the Pico interpreter. The Pico implementation consists of many continuation functions, of which an excerpt is shown in figure 1b. The execution of this function results in the creation of an execution-trace (shown in figure 1a) that only contains the observed behaviour in terms of high-level events. These high-level events are used to verify the property-driven model specified in figure 1c. Finally, figure 1d, 1e and 1f specify how the high-level events need to be recorded during the execution of the program. We will further explain the details of each of these parts throughout the remainder of the paper.

To summarize, the use of the BEHAVE platform for property-driven testing of software applications amounts to the following 4-step recipe:

- Identify the high-level run-time events of the property to be tested,
- Specify a property-driven model,
- Specify the application-specific instances of high-level run-time events,
- Run the test by verifying the property-driven model against the actual behaviour and evaluate the results.

In the following sections we will explain each of these steps through the application of BEHAVE for testing the garbage collector of the Pico language interpreter.

3.1. Identifying the High-Level Run-Time Events

The first step of our recipe comprises the identification of the high-level run-time events we need to verify the garbage collection property mentioned in section 2. Having analysed that description, we need to specify the following high-level run-time events: possible garbage collect (`possibleGC`), temporary variable used (`tempUsed`) and temporary variable assigned (update or initialisation) (`tempUpdated`). Indeed, we want to detect occurrences of possible garbage collection events *in between* the assignment and use of temporary variables holding a reference to

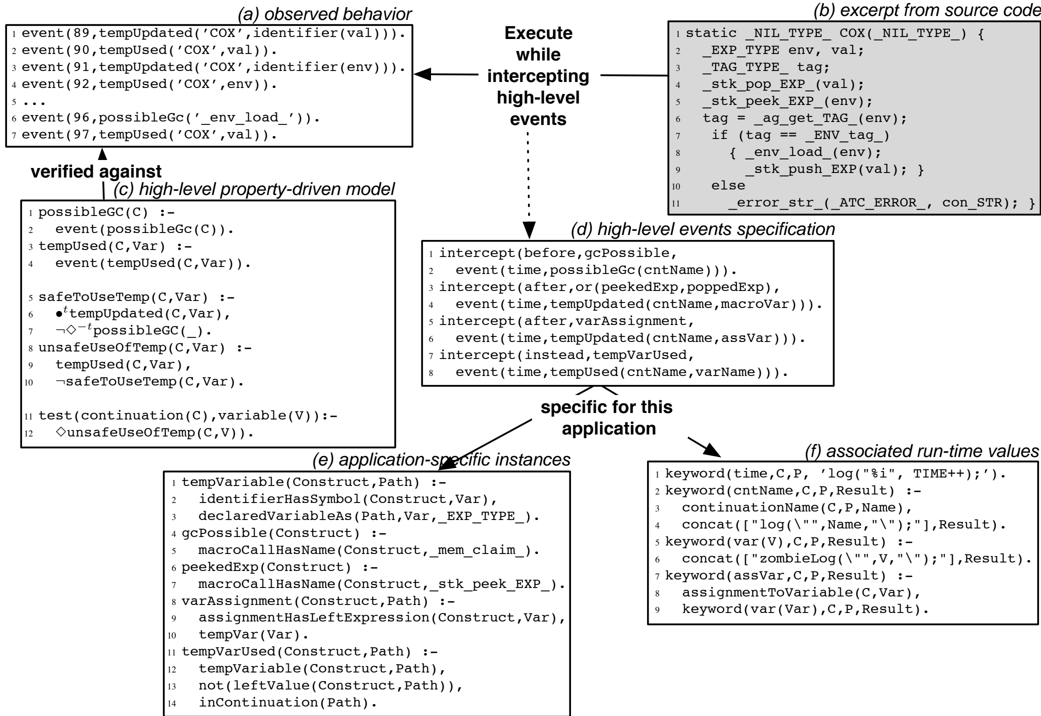

Figure 1: Using the BEHAVE platform for testing Pico memory management

the Pico memory. Figure 1a shows instances of these events in an execution trace using logic (Prolog) facts. Each event also declares additional information associated to the event. For example, each event contains a time-stamp, which is the first value that is shown in each event instance in figure 1a. The `tempUsed` event on line 2 also declares the name of the temporary variable (`val`) and the name of the continuation function (`COX`) in which it is defined.

The identification of these high-level run-time events allows the property-driven model to be specified at a high level of abstraction, i.e. in terms of these high-level events instead of low-level execution traces. Furthermore, because a developer can specify which high-level events to intercept, this approach also achieves a selective and compact execution trace only containing the needed run-time information. Note that the identification of these high-level events is completely up to the developer; he has to specify *when* such a high-level event occurs and also *how* to obtain any additional run-time information. We will explain in section 3.3 how these high-level events are specified such that they are recorded during the execution of the program.

3.2. Specifying a Property-Driven Model

In the second recipe step of BEHAVE, we specify a property-driven model using a temporal logic programming language based on Prolog. We chose a logic language as specification language because of its declarative nature. This ensures that a model will be human-readable and at a high level of abstraction, which is needed for specifying a property in terms of concepts rather than low-level programming constructs, such as a call to a function. In Prolog, a program consists of logic clauses representing knowledge about a particular problem at hand. As we have explained in the previous section, the high-level execution trace of the C program also consists of a set of logic facts. This means that we can implement the property-driven model as a set of high-level assertions *over* the execution trace, expressed using logic clauses.

The choice for a temporal variant of Prolog is particularly useful because temporal logic programming comprises all functionality of logic programming and provides some additional logic operators for expressing temporal constraints. Such temporal constraints are of extreme importance when reasoning about events representing run-time behaviour. The temporal operators which we will further use in our property-driven model are: $\Box$ (always), $\diamond$ (sometimes), $\bullet$ (previous) and $\circ$ (next). Using these operators, we can easily describe temporal relations between events contained in the execution trace as opposed to when you use traditional logic programming

Finally, since we use our platform for testing an application's behaviour, we are always looking for occurrences of *unwanted* behaviour. Testing is an activity that tries to find

out where the behaviour of a program is broken. Therefore, our property-driven models will need to specify unwanted behaviour and *where* in the source code this unwanted behaviour occurs. This approach is of importance for correcting the source code of the application in case a test fails, as we will explain in section 3.4.

Using the high-level events identified in section 3.1, we can now specify a property-driven model using temporal logic clauses that describes the desired as well as the undesired behaviour of the program with respect to these events. In section 2, we described that a possible garbage collect event may not occur between the assignment (which is either an update or an initialisation) and the use of a temporary variable. In other words, *after a possible garbage collection event you should not use temporary variables unless they have been updated in between.* Expressed as unwanted behaviour, we will look for all uses of temporary variables that do not obey this last saying. The property-driven model that specifies this behaviour is shown in figure 1c. Lines 1-4 introduce additional abstractions over the events in the execution trace, i.e. `possibleGC` and `tempUsed` [1]. On lines 5-10 the concepts of unsafe and safe uses of temporary variables are defined as the logic assertions `unsafeUseOfTemp` and `safeToUseTemp` respectively. This last assertion (defined on lines 5-7) states that it is safe to use a temporary variable `Var` within a continuation function `C` if within `t` timesteps in the past from now, the variable `Var` has been updated (or given an initial value if it didn't have a value before) and if within that time frame no possible garbage collection could have occurred. The `unsafeUseOfTemp(C,Var)` assertion on lines 8-10 then captures the unsafe use of a temporary variable that states the complement. The actual test can be seen on lines 11-12 and expresses the wish of finding a variable `V` within a continuation function `C` that is used in an unsafe way, as it is defined above.

3.3. Specifying Application-Specific Instances of High-Level Events

In the previous sections, the high-level events that we need for verifying garbage collection were identified and the property-driven model was expressed as assertions over these high-level events using temporal logic programming. In this subsection, we describe how a developer that uses the BEHAVE platform can specify *which* events have to be intercepted and *how* they should be recorded. This consists of specifying *what* source-code constructs raise these high-level events (figure 1e), how associated run-time values need to be extracted in order to be recorded with the event (figure 1f) and the specification of the events themselves (figure 1d). Once again, logic programs are used to describe these actions.

The declarations in figure 1d describe the high-level events that need to be recorded. All declarations are of the form `intercept(When,What,RecordAs)`. We will explain these declarations by example. Consider lines 1-2 where we declare that the high-level event of a possible garbage collect is recorded in the execution trace as `event(time,possibleGc(cntName))`. More precisely, this event will be recorded right `before` the execution of the source code construct that is identified by the `gcPossible` assertion. This assertion is defined by the logic clause on lines 4-5 in figure 1e. This clause specifies that the high-level event of a possible garbage collect is triggered by the execution of the source-code construct `Construct` if it is a C macrocall that is named `_mem_claim_`. There are three other macrocalls in the Pico implementation besides _mem_claim_ that can trigger the garbage collection property. However, due to space limitations we did not include all of them here but there exists a very similar logic clause for each of them. Furthermore, instead of merely logging the occurrence of a possible garbage collect event, we also log the `time` at which the event occurred and the name of the continuation function `cntName` [2] in which the possible garbage collect event occurs. We will explain how the run-time values of these so-called *keywords* `time` and `cntName` are computed later on.

In lines 3-6, the same high-level event (`event(time,tempUpdated(cntName,varName))`) is described by two different logic declarations. This is because the high-level event of a temporary variable being (re)assigned a value can be manifested in the source code in various ways. More specifically, this event is triggered by the execution of the source code construct that is specified by the `peekedExp`, the `poppedExp` or the`varAssignment` assertions. Another difference between the two clauses has to do with the different run-time values that are needed. These are represented by the keywords (`macroVar` and `assVar`.

The source code constructs can be identified because BEHAVE makes an entire application's parse tree available. Let's have a look at the logic clause on lines 1-3 in figure 1e that defines a temporary variable as the `tempVariable(Construct,Path)` assertion. This rule has access to each parse tree node through the `Construct` variable, while the `Path` variable represents the path from the tree's root that leads to that `Construct` node. This defines what construct is a temporary variable pointing to a Pico memory chunk in the Pico implementation. In essence, some source-code construct is such a temporary variable if it is an identifier with name `Var` and if it has been declared

[1]Note that we also need the same abstraction for `tempUpdated(C,V)`, but we didn't include it here due to space limitations.

[2]Note also that `cntName` and `time` are not logic variables here; in Prolog logic variables are denoted with a capital letter

as being of type _EXP_TYPE_. The tempVarUsed rule on lines 11-14 needs some more explanation. This rule states that a temporary variable (denoted by the variable Construct) has been used if first of all it is a temporary variable (line 12), if it is found to be used within a continuation function (line 14) and if it is not part of the left side of a variable assignment (line 13), because then it is not used but updated.

We still need to explain how the run-time information associated with each high-level event can be retrieved. In figure 1d on line 2 we denoted that for the high-level run-time event possibleGc we wanted to log the name of the continuation function where the possible garbage collection event could occur through the cntName keyword. These run-time values will have to be obtained by the execution of application-specific source code. The most important keywords used here together with the associated C code can be found in figure 1f. On line 1, the time keyword is defined as a call to a C log function that will merely write a number to the execution trace file and increment a time counter. The keyword cntName on lines 2-4 uses a continuationName rule, which is not included here, but this rule looks for the continuation function node in the Path of this Construct and takes its name. The two rules on lines 5-9 are used in combination to be able to log the name of a variable to which a variable has been assigned [3].

3.4. Running the Test and Evaluating the Results

Verifying the property-driven model against the actual behaviour and then evaluating the results forms the last step of our recipe. Our platform first instruments the source code of the application under test to be able to record all possible occurrences of the defined high-level run-time events. Second, the instrumented source code gets executed according to a particular scenario which creates the execution trace containing the high-level run-time events that occurred during execution. Afterwards, the property-driven model can be verified by launching the logic query :- test(Function,Variable). A fail of this query would mean that no variables can be found that are used in an unsafe way. However, if this query yields results, the results will describe which variables in which continuation functions are used in an unsafe way.

Applying our approach to the complete Pico implementation, we were able to find three occurrences of unsafe usage of variables in two different continuation functions. Let us consider one of the two results: continuation('COX'), variable(val);. This result means that in the COX continuation function the temporary variable val is used in an unsafe way. Depending on the state of the heap representing underlying Pico memory, if a garbage collect event occurred that triggers a heap defragmentation during the execution of the COX continuation function, the system would crash completely. The C code fragment representing this function is depicted in figure 1b: on line 9 the temporary variable val is used and apparently the statement on line 8, a call to the function _env_load triggered a possible garbage collection event (as can be seen in figure 1a on line 6).

[3] Note that we need two completely analogous rules to the one on lines 7-9 for the keywords macroVar and varName as used in figure 1d

4. Related Work

A lot of research has been done on dynamic analysis approaches, differing in the domain they are applied, analysis time, expressiveness of the medium used to express behavioral models, what run-time events can be intercepted, etc. We briefly discuss those most closely related to our approach.

CCI [12], is a general program monitor notification tool for C programs. At run-time, an execution monitor is notified of events through calls to a user-implemented macro which takes an integer indicating the type of event that occurred and an event-specific associated value. As CCI is a pure program monitor notification tool, it provides no language specifically tailored to reasoning about intercepted events, which our platform does provide. JMonitor [9] provides a Java API for specifying event patterns and associating them with user provided event monitors. JMonitor doesn't offer any mechanism either to analyse observed events.

Coca [6] performs a dynamic analysis on C programs by having them executed in a stepwise fashion by a debugger. Coca runs alongside the debugging process to steer the execution of the program. Analysis is done on-line, so considering alternative matches for a run-time event without advancing the application is not possible. This makes it difficult to express assertions about nested events. Coca is however suited to debugging purposes as queries can be expressed in a Prolog variant augmented with predicates to request future run-time events.

Auguston et al. [1, 2] present the interesting procedural assertion specification language FORMAN. Atomic low-level events occur at a time point, while composite events occupy an interval in time. An event grammar formally specifies the low-level constituents of composite events and their mutual ordering on the time line. This allows for automatic low-level event selection according to a given assertion over composite events. The information recorded about each composite event is however dependent on its atomic constituents. In this setting we prefer the more declarative nature of the property models resulting from our temporal logic programming specification language. Our experiments have also indicated the benefits of being able to freely determine the run-time values associated with high-

level events. Due to the lack of a formal event grammar, our approach however requires more user involvement.

TestLog [7] focuses on testing Smalltalk applications. It also uses a Prolog-like logic language, but to reason about whole-program execution traces comprising method invocations and the recursive state of each receiver before and after the invocation.

5. Conclusion

We presented the BEHAVE platform for testing behavioural properties of software applications using declarative property-driven behavioural models. These models are formulated as declarative high-level temporal assertions over high-level run-time events freely chosen by the application tester. Our approach features the following characteristics:

- Because our testing approach is property-driven, we can verify a specific part of program execution; this makes our approach applicable to larger programs as well;
- The most commonly used dynamic analysis approaches demand a model being written over a fixed set of low-level programming language constructs; using the BEHAVE platform an application tester can specify himself what run-time events should be recorded; consequently verification is again computationally less expensive because of the selectively chosen events in the execution of a program;
- Moreover, the high-level property-driven models are written in a declarative medium which makes them more human-readable; hence they can also serve as documentation artefacts between successive application testers during software evolution;

Furthermore, to cope with the large degrees of freedom our platform offers, we outlined a four-step recipe which guides a user of our platform for testing a specific system property. We validated this recipe through the testing of the garbage collector in the Pico language interpreter and were able to find some undiscovered bugs in the implementation. The BEHAVE platform has also been validated in the context of documenting and verifying high-level behavioral program documentation [11]. And we are planning experiments to use our platform in a teaching environment to aid a student in grasping the concepts of a particular program under study.

References

[1] M. Auguston. Building Program Behavior Models. In *Proc. of the European Conf. on Artificial Intelligence Worksh. on Spatial and Temporal Reasoning (ECAI98)*, pages 19–26, 1998.

[2] M. Auguston. Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 159–166, July 6-8 2000.

[3] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[4] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 2nd edition edition, 1995.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] M. Ducassé. Coca: An automated Debugger for C. In *Proc. of the 21st Intl. Conf. on Software engineering (ICSE99)*, pages 504–513, 1999.

[7] S. Ducasse, M. Freidig, and R. Wuyts. Logic and Trace-based Object-Oriented Application Testing. In *Proc. of the Intl. Worksh. on Object-Oriented Reengineering (WOOR04)*, 2004.

[8] P. Hamill. *Unit Test Frameworks*. O'Reilly, November 2004.

[9] M. Karaorman and J. Freeman. jMonitor: Java Runtime Event Specification and Monitoring Library. In *Proc. of the 4th Intl. Worksh. on Run-time Verification (RV04)*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 181–200, 2004.

[10] W. D. Meuter, T. D'Hondt, and J. Dedecker. Pico: Scheme for Mere Mortals. In *Online Proc. of the 1st European Lisp and Scheme Worksh.*, 2004.

[11] C. D. Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt. An Approach to High-Level Behavioral Program Documentation Allowing Lightweight Verification. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), Athens, June 14-16 (To be published)*, 2006.

[12] K. Templer and C. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. In *Proc. of the 13th IEEE Intl. Conf. on Automated Software Engineering (ASE98)*, page 249, 1998.