An Approach to High-Level Behavioral Program Documentation Allowing Lightweight Verification

Coen De Roover, Isabel Michiels, Kim Gybels, Kris Gybels and Theo D'Hondt
Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
cderoove,imichiel,kgybels,kgijbels,tjdhondt@vub.ac.be

Abstract

Typically, multiple developers are involved in the various stages of the software development and maintenance process. To ensure an optimal transfer of knowledge between these different peers, a reliable human-readable model of the dynamics of a software artefact is needed. Once these models become machine-verifiable, they can be used throughout an application's lifetime to check whether the documented behavioral properties continue to hold as the application evolves. Unfortunately, most existing modeling media are inadequate to express human-readable behavioral models which are at the same time machine-verifiable. We therefore propose a declarative platform wherein behavioral program models can be expressed in terms of user-defined high-level concepts and be automatically verified against an application's actual behavior. We demonstrate our approach by using it to both document and verify an interpreter for a garbage-collected programming language.

1. Introduction

From a software system's initial conception to its eventual maintenance, it is important for all involved to have a reliable mental model of the interactions between its components and the processes that govern the behavior that thus emerges. Behavioral program models are not only invaluable in order to locate and evaluate the impact of a necessary maintenance change, but also play an important role in the communication of knowledge between a developer's peers. Once written down, program models serve as knowledge transfer vehicles between successive developers, thus reducing the overall program understanding effort.

There are two important desirable properties for any model documenting a program's behavior. On the one hand, program models should be written down in a descriptive human-readable format in order to serve their communicative purposes. Ideally, to encourage daily use, expressing a model in such a human-readable format shouldn't be too tedious either, while the resulting models should convey as much relevant information as possible.

On the other hand, the second desirable property of a program model has to do with keeping the information it conveys as reliable as possible. As an application evolves, the corresponding behavioral model will have to be adapted accordingly. A model might have to be extended in order to accomodate for changes, but ideas expressed in the model might have to be revisited as well. Manually detecting discrepancies between a program's observed behavior and the behavior that was documented by the model is unfortunately time-consuming and error-prone. In order for this task to be automated, models should be expressed in a machine-verifiable format. Ideally, to encourage daily use, the automated verification shouldn't take too much time on larger programs either. The second desirable property is therefore lightweight machine-verifiability.

At first sight, human-readability and lightweight machine-verifiability seem to be conflicting properties of behavioral program models. In this paper, we will however show that these two desirable properties can be reconciled by introducing high-level concepts in the behavioral descriptions documented by a program model. Such concepts are of a higher semantic level than the individual programming constructs offered by a particular implementation language.

The introduction of high-level concepts in behavioral program models achieves the human-readability property by making it easier for developers to communicate and reason about a program's behavior at large, possibly even enabling concept reuse across models for applications in the same domain. It is therefore interesting to consider machine-verifiable behavioral program models that are built from such higher-level or even domain-level concepts. At the same time, we will explore how the verification of an application's actual behavior against models described in terms of high-level concepts can be made computationally less expensive than the verification against models described in terms of low-level programming constructs.

We have developed a platform, called BEHAVE, which supports this approach for programs written in C. We will start our paper by outlining our approach to documenting a program's behavior in high-level, machine-verifiable models. We will continue our discourse in section 3, throughout which we will use a naive stack implementation as the running example to introduce the BEHAVE platform supporting our approach. We will carefully outline the steps necessary to document and verify the stack's behavior in a high-level program model. In section 4, an experiment will be presented on a medium-sized but fairly complex C program: an interpreter for the Pico programming language, whose run-time behavior we have documented and verified using our platform. Related work will be reviewed in section 5 and we conclude with a discussion of our approach in section 6.

2. Machine-Verifiable High-level Program Documentation

In this section, we will outline our approach to documenting a program's dynamic behavior in high-level human-readable models which can be verified throughout the application's entire lifetime. Two important phases can be discerned in all machine-verifiable documentation approaches: one wherein developers document knowledge about the behavior of a program in a model and one wherein developers verify the consistency of the documented and actual program behavior. In case of inconsistencies, a third phase comprises the interpretation of the obtained verification results which might initiate another iteration through the documentation and verification phase. In our approach, the documentation consists of two parts: behavioral assertions documenting desired and non-desired program behavior and a description of the high-level concepts over which this knowledge is expressed.

2.1. Verifying Program Documentation

To better understand how our platform works, we will first explain how in the second phase the consistency of the documented and the application's actual behavior is verified. As we mentioned before, we wanted the verification phase to be lightweight. Heavyweight program verifiers [5, 6] check whether a program always exhibits the desired behavior regardless of any specific execution scenario. Although this feature seems desirable, the high computational cost associated with such an approach interferes with our daily usage requirement. We therefore opted for a dynamic analysis strategy wherein the program under investigation is executed along a well-defined scenario which prescribes user input and program arguments. Run-time events arising during the execution of the program are recorded in an execution trace.

Automatic verification thus amounts to checking whether the obtained execution trace exhibits the wanted and not the unwanted behavior documented in the program model. Our verification results are however always relative to the user input prescribed by the execution scenario. A well-chosen execution scenario moreover offers a convenient way to focus the verification on specific parts of a larger program. Our approach is herein similar to unit testing [3] advocated by the Extreme Programming community.

2.2. Specifying Program Documentation

Now that we have explained the nature of our verification phase, it is time to focus on the most distinctive feature of our platform: its ability to document a program's behavior in high-level behavioral program models. Again, as we mentioned before, we wanted to encourage daily use and make sure documenting a program's behavior doesn't become too tedious. In contrast to many heavyweight program verification approaches, the focus in our approach to machine-verifiable program documentation is on the communicative aspect of the models documenting a program's behavior. We therefore do not require developers to exhaustively model every possible program state that may arise during an application's execution. On the contrary, our models only describe relevant desired and unwanted program behavior.

Developers have to specify desired and unwanted behavior as a set of assertions over the run-time events that are recorded during the application's execution. Most verification approaches relying on dynamic analysis demand these assertions to be expressed over a *rigid* set of *low-level* run-time events that are directly related to programming language constructs. Examples of such low-level events are for instance assignments to variables or calls to functions. An assertion might state that after every call to the function `bar`, the value of the variable `foo` has been increased. Expressing entire program models in terms of such assertions over low-level run-time events can however become awkward and the resulting models lose much of their ability to convey any semantic information at all. Moreover, there is little room for reuse of models across applications in the same domain. Given our knowledge transfer setting, we would therefore rather be able to document a program's behavior in terms of concepts that are of a higher semantic level instead. For a stack datastructure, concepts that immediately come to mind are the pop and push operations and the elements on the stack. A higher-level assertion about the behavior of a stack datastructure might for instance state that the size of the stack grows after every push operation. Such an assertion is immediately decoupled from a particular stack implementation and might be reused across many applications employing a stack datastructure.

At the very least, we thus require that the language used to express assertions in, allows us to compose higher-level events from low-level run-time events and allows us to decouple these events from concrete source code. This would for instance already allow us to specify that, for a particular program, the operation pop referenced by a high-level stack program model, is implemented by a call to the concrete function `pop`. However, the values associated with each low-level run-time event are predetermined too. Of a low-level assignment event, usually only the assigned value and modified variable are available. This limits the range of values that can be associated with composite higher-level events as well. In case we would like to know the concrete elements on the stack after each stack operation, we would be forced to compute this information from the low-level assignment events that preceded the operation. Such post-mortem computations could make the verification of behavioral program models expensive, which would conflict with our lightweight machine-verifiability desirable property.

We therefore decided to go one step further and chose to have the run-time events recorded during the execution of the application be high-level themselves. This allows behavioral assertions to be expressed directly over high-level run-time events. The identification of high-level events is completely left up to the developer. They can choose to model a program's behavior using high-level events that are shared between multiple applications. For a stack datastructure, these could for instance be the typical stack manipulating operations. A developer only has to specify *when* such a high-level event takes place and also *how* to obtain any additional run-time information that is associated with such an event. For the stack datastructure, the information associated with each stack operation could for instance indicate the elements on the stack after the operation. A developer using our platform is thus free to determine the high-level events that arise during an application's execution and the information about a program's run-time state that is associated with each event. With respect to the human-readability requirement, special care has been given to the design of the descriptive language in which program models and their meta models have to be specified. We will introduce this specification language in the following section using a running example.

Our decision to record high-level instead of low-level run-time events during the execution of a program, directly complies with our lightweight machine-verifiability requirement as well. As developers are now able to specify when and how events occur at run-time, we will no longer have to record every single low-level run-time event. The resulting high-level execution traces in general comprise fewer run-time events which allows our approach to be applied on larger programs. The introduction of high-level events in both the execution traces and the program models against which they are verified, therefore allows for the practical investigation of larger programs using succinct and highly descriptive models suitable for knowledge transfer.

3. Using BEHAVE to Document and Verify Program Behavior

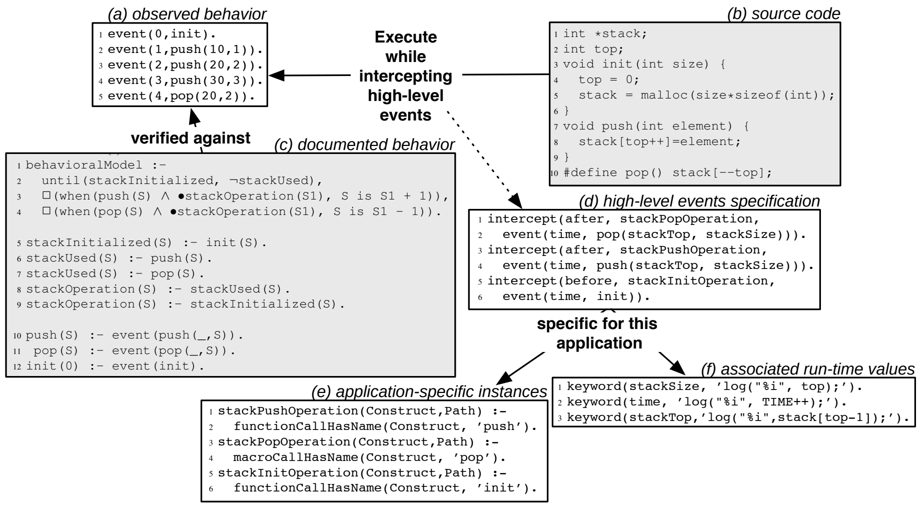
Using the BEHAVE platform to document a program's behavior, developers can freely determine the high-level events over which they want to express behavioral assertions in the program's model. Although such large degrees of freedom inherently require more developer involvement, the benefits of high-level program models with respect to the aforementioned desirable properties definitely merit an investigation.

In order to optimally exploit the platform during an application's life-cycle, a developer can therefore adhere to the following proven recipe. For the documentation phase of our approach, developers have to identify the high-level events that are to be used in the program model (1), document the current understanding of a program's internal workings in a model specifying desired and unwanted behavior (2) and specify the application-specific instances of high-level events (3). For the verification phase of our approach, developers can have the consistency of the program's documented behavior and its actual behavior verified by our platform (4). In the subsequent sections, we will detail each of the recipe's steps using a simple stack implementation as the running example.

3.1. Identifying High-Level Run-Time Events

The first step in our recipe to document a program's behavior in a machine verifiable format, is to identify the high-level events over which behavioral assertions will be expressed in the program model. Inadvertently, when thinking about stack datastructures, the operations *pop*, *push* and *initialize* immediately spring to mind. Independent from a concrete implementation, the *size* of the stack and the element on *top* are other important stack-related concepts. As the concrete values of these concepts will vary over time as the stack is being manipulated, we will consider these as additional information about a program's state associated with each of the high-level events representing stack operations.

Considering the concrete stack implementation as shown in Figure 1b, we could as well have described our program model in terms of calls to the function `push` or array manipulations originating from an expansion of the macro `pop`.


(a) observed behavior
Execute while intercepting high-level events
(b) source code
verified against
(c) documented behavior
(d) high-level events specification
specific for this application
(f) associated run-time values
(e) application-specific instances

(a) observed behavior

```
event(0,init).
event(1,push(10,1)).
event(2,push(20,2)).
event(3,push(30,3)).
event(4,pop(20,2)).
```

(b) source code

```
int *stack;
int top;
void init(int size) {
  top = 0;
  stack = malloc(size*sizeof(int));
}
void push(int element) {
  stack[top++]=element;
}
#define pop() stack[--top];
```

(c) documented behavior

```
behavioralModel :-
  until(stackInitialized, ¬stackUsed),
  □(when(push(S) ∧ •stackOperation(S1), S is S1 + 1)),
  □(when(pop(S) ∧ •stackOperation(S1), S is S1 - 1)).

stackInitialized(S) :- init(S).
stackUsed(S) :- push(S).
stackUsed(S) :- pop(S).
stackOperation(S) :- stackUsed(S).
stackOperation(S) :- stackInitialized(S).

push(S) :- event(push(_,S)).
 pop(S) :- event(pop(_,S)).
init(0) :- event(init).
```

(d) high-level events specification

```
intercept(after, stackPopOperation,
  event(time, pop(stackTop, stackSize))).
intercept(after, stackPushOperation,
  event(time, push(stackTop, stackSize))).
intercept(before, stackInitOperation,
  event(time, init)).
```

(e) application-specific instances

```
stackPushOperation(Construct,Path) :-
  functionCallHasName(Construct, 'push').
stackPopOperation(Construct,Path) :-
  macroCallHasName(Construct, 'pop').
stackInitOperation(Construct,Path) :-
  functionCallHasName(Construct, 'init').
```

(f) associated run-time values

```
keyword(stackSize, 'log("%i", top);').
keyword(time, 'log("%i", TIME++);').
keyword(stackTop,'log("%i",stack[top-1]);').
```

Figure 1: Source code and corresponding behavioral documentation of a C stack implementation.

We have however already argued in Section 2.2 that program models consisting of assertions over such low-level run-time events do not adhere to our human-readability desirable property nor do they allow the reuse of high-level events across documentation for applications in the same domain.

3.2. Documenting Program Behavior

The next step in our recipe comprises documenting a program's behavior in a machine-verifiable, but human-readable model. Our most basic understanding of the behavior of a typical stack is that it grows in size whenever new elements are pushed on it and shrinks whenever an existing element is removed through the pop operation. The second perception that springs to mind is that failures might follow when the stack hasn't been correctly initialized. We could also specify assertions about the top of the stack with regard to the push and pop operation, but we rather opt for a small model as our objective here is to introduce the platform.

In order to have our insights checked automatically, we have to specify them in a machine-verifiable language. To facilitate this task as much as possible, the specification language should be highly expressive. An expressive, declarative language additionally ensures that the resulting specifications are descriptive and convey as much information as possible. Machine-executable logic languages, whereof Prolog [9] is the well-known archetypical representant, are good contenders.

In such programming languages, a program consists of logic clauses representing a developer's knowledge about a particular problem. In our program verification platform, the logic program consists of a logic representation of the source code and an execution trace of the C program under investigation. Our behavioral assertions over the run-time events from the execution trace, are expressed as logic formulas. To determine whether such a formula is a logical consequence of the logic program, logic programming languages rely on a proof procedure. While behavioral program models could be expressed in regular Prolog, our BEHAVE platform offers an extended Prolog variant which is especially suited to model the temporal relations between run-time events.

Formulae in temporal logics comprise the classical logic formulae possibly qualified by temporal operators such as $\Box$ (always), $\diamond$ (sometimes), $\bullet$ (previous) and $\circ$ (next). The truth value of a formula depends on an implicit temporal context: a formula can be true at a certain moment in time, while it might be false at the next moment. While different time models are supported in temporal logics, a finite linear time model suffices for our approach. In this model, informally, the temporal formula $\Box\phi$ is true if ϕ is true at all moments in time. Similarly, we have that $\diamond\phi$ is true when ϕ is true at some moment in time.

Temporal logic programming languages [11] are based on a subset of a temporal logic such that programs written in this subset are machine-executable. MTL [4] is a temporal logic programming language based on metric temporal logic. Metric temporal logics incorporate an additional quantitative aspect into the temporal operators. The $\diamond_t$ (sometimes within t time points) operator is for instance available. We propose the use of a variant of MTL[1] to document a program's behavior in behavioral assertions over

[1]Regular MTL rules may not contain applications of $\Box$-operators in their bodies, or $\diamond$-operators in their heads. These limitations however do not apply in our context where the boundaries of time coincide with the start and end of the execution of a program.

high-level events.

While temporal logic formulae have been successfully applied in the program verification domain, they are sometimes hard to understand [7]. Users unacquainted with temporal logic are therefore free to document their programs using assertions expressed in plain Prolog. Temporal operators, however, allow the delimitation of temporal contexts in a very descriptive manner without having to explicitly manipulate integers representing points in time. Program models expressed in Prolog might thus be less concise, but can still be expressed at a conceptual instead of an implementation level thanks to the platform's high-level run-time events. Moreover, our BEHAVE platform employs a temporal logic programming language as the specification language instead of a plain temporal logic. As users can combine temporal operators into reusable higher-order logic rules, they are able to express often recurring temporal patterns without having to remember their idiomatic expression in plain temporal logic. Examples are various occurrence and ordering patterns which were identified as appearing most often in specifications for verification systems [7].

Given MTL as a specification language, we can easily express our understanding of the behavior of a stack in the machine-verifiable, human-readable model depicted in Figure 1c. The second line of the extract states that until the stack is initialized, it may not be used. The third line states that it must always be the case that whenever a push operation left the stack in a state with size S, any previous stack operation should have left the stack in a state with size $S-1$.

Lines 5–9 define the logic predicates used within these assertions. Lines 8 and 9 define that a `stackOperation` comprises either the initialization or manipulation of the stack. The `push` and `pop` operations are considered stack manipulators, which is expressed by the `stackUsed` predicate in lines 6 and 7.

The final 3 lines of the extract link the predicates used in the behavioral model to the high-level events observed during the execution of the program. We can see that the high-level push and pop events in the execution trace record more information about the state of the stack than is actually needed by this model specification. The first recorded value is ignored as we are only interested in the second value which records the size of the stack. By altering the definition of the `push`, `pop` and `init` predicates, our behavioral model specification can be reused even when different run-time values are associated with the high-level events in the execution trace. In the next step of our recipe, developers have to specify *how* to observe these high-level events during the execution of the program.

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

At this point in our 4-step recipe, we have identified the high-level run-time events typically associated with a stack datastructure. We have also specified a model of its behavior using assertions over these events expressed as temporal logic formulae. The high-level run-time events *push*, *pop* and *initialize* will be the constituents of the execution traces against which we will verify the program's behavioral model shown in Figure 1c. An example of such an execution trace is shown in Figure 1a.

Since the recorded execution traces consist of user-defined high-level run-time events, developers have to specify *which* events can be intercepted during an application's execution and also *how* each event is recorded. For our running example, this specification is shown in Figure 1d. It consists of a set of `intercept(When, What, RecordAs)` declarations. On the first line of the specification, we declare that all occurrences of a high-level *pop* run-time event have to be intercepted. We also declare that these events must be recorded in the execution trace as facts of the form `event(time, pop(stackTop, stackSize))`. Instead of merely logging the occurrence of this high-level event, we also record the top of the stack and its size after the event occurred. These will be the run-time values associated with the *pop* event. Occurrences of the *push* and *initialize* events are logged analogously.

The behavioral assertions and *which* high-level events to intercept is documentation that can be shared by different applications. For each specific program, we only need to specify *how* to intercept the high-level events. Applied to our running example, this for instance amounts to identifying the constructs in the application's source code that give rise to the high-level *pop* event. From the code depicted in Figure 1b, it is clear that this operation is implemented by code resulting from an expansion of the `pop` C function macro on line 10. We can express this knowledge in the `stackPopOperation` rule whose concrete implementation was left open in the aforementioned `intercept` declaration. As shown on line 3 of Figure 1e, this rule states that the *pop* event is caused by calls to the macro named `pop`.

To support developers in the identification of constructs that give rise to a high-level event, BEHAVE makes an entire application's parse tree available[2]. The `stackPopOperation(Construct, Path)` rule is presented each parse tree node through the `Construct` variable, while the `Path` variable represents the path from the tree's root that leads to that node. Although the identification rules for

[2]The original macro calls can be accessed in this parse tree since they are not expanded by our parser. However, using macros, one can write programs that break the C parsing rules. In that case, developers currently have to use the C pre-processor and lose all macro information as we do not address this issue in our prototype implementation.

our running example only need to access attributes from the parse tree nodes, Prolog's full declarative reasoning power will be needed for the experiment in Section 4.

At this time, we still have to declare how the run-time information associated with each high-level event can be retrieved from the specific stack implementation whose behavior has to be verified. On lines 1–2 of Figure 1d, using the keywords `stackTop` and `stackSize` we declared that the size of the stack and the element on top should be recorded after each pop operation. These run-time values will have to be obtained by the execution of application-specific source code. For the stack implementation of the running example, the C code associated with each keyword is shown in Figure 1f. For the use case in Section 4, somewhat more elaborate code will have to be provided in order to obtain the correct run-time values for each high-level event.

3.4. Lightweight Consistency Verification

In the last step of our recipe, developers can verify the consistency of a program's actual behavior with its documented behavior by launching logic queries against a recorded execution trace. BEHAVE instruments the source code of the application under investigation in order to record all occurrences of the high-level run-time events specified in the behavior program model. To intercept occurrences of the high-level *pop* event, the platform relies on the `stackPopOperation` logic rule to identify those source code constructs which give rise to the *pop* event. The platform also relies on the definition of the `stackTop` and `stackSize` keywords to obtain the run-time values associated with this event.

To verify the behavioral model specified in Figure 1d, the logic query `?- behavioralModel` has to be launched. In case of a verification failure, our temporal logic interpreter prints the last event that was used in an attempt to prove the query. This information can be used to either adapt the application to the model or the model to the application. The generated execution traces consist of high-level events which renders manual inspection in case of verification failures somewhat more feasible. As discussed in Section 2, the generated execution traces contain in general fewer events as well since not every low-level event needs to be recorded.

4. Experiment: Documenting and Verifying the Behavior of the Pico Interpreter

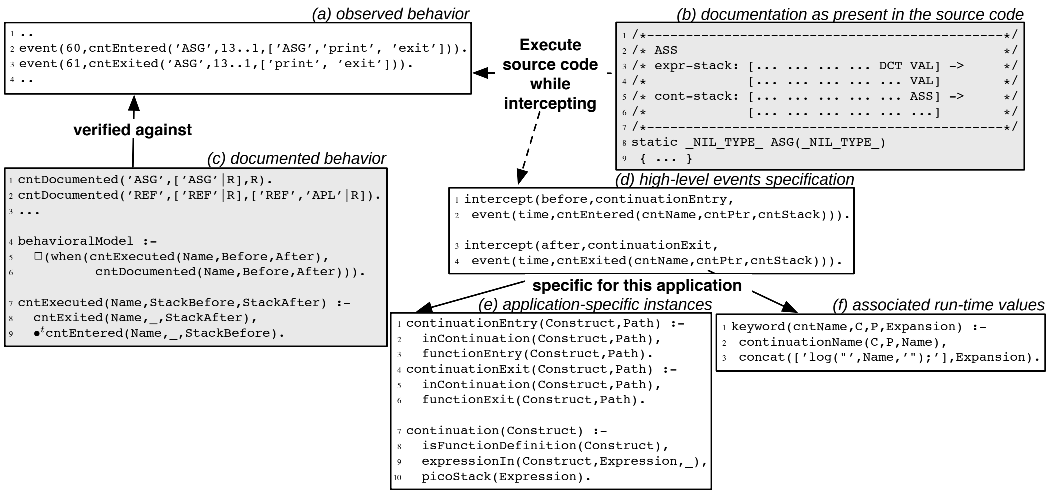
In this section, we will present the results of an experiment performed using the BEHAVE platform. In this experiment, we verified the actual behavior of an interpreter for a programming language against the available documentation. This interpreter presented a unique opportunity to test our approach: the original developer documented its behavior thoroughly in a non machine-verifiable format, while many changes have been made to the interpreter over time. As this interpreter is nowadays used to introduce computer science students to the foundations of interpretation, it is important to have reliable documentation conveying its dynamics in a concise but descriptive manner. In this experiment, we therefore formalised the existing documentation and verified whether it was still loyal to the actual behavior of the interpreter. The experiment demonstrated that the introduction of high-level events in the interpreter's behavioral program model resulted in machine-verifiable documentation that was as human-readable as the original. At the same time, the introduction of carefully selected high-level events in the execution traces resulted in relatively compact traces allowing for a more lightweight verification.

4.1. The Pico Interpreter

Pico [10] is an interpreted programming language developed at the Vrije Universiteit Brussel. Originally conceived to teach programming concepts to students outside the realm of computer science, its C implementation –which totals about 16K lines of condense C code– is nowadays also heavily used as a teaching vehicle in the computer science curriculum.

The Pico interpreter relies on the concept of a continuation to represent the subtasks a computation –such as the evaluation of an expression– comprises. Contrary to the conventional semantics, a Pico continuation does not denote the entire future of the computation at hand, but rather a piece of this computation. The Pico interpreter stores these pieces on a stack. The entire stack of continuations, to which we will refer as the continuation stack, therefore represents the complete future of the computation. A continuation may invoke other continuations by placing them on the continuation stack. Arguments can be passed by storing them on a separate stack referred to as the expression stack. The heart of the Pico interpreter is a loop which continuously executes the continuation located at the top of the continuation stack. Continuations are implemented as pointers to C functions which take no arguments nor return a value.

The internals of the Pico interpreter are documented in a very consistent manner. As a well-defined sequence of continuation and expression stack manipulations determines the operational semantics of each Pico expression, the interpreter's documentation conveys how these stacks evolve during the evaluation of a program. For each continuation, the documentation describes what the continuation stack and expression stack are expected to look like before and after the execution of the continuation. Consider for instance the documentation of the `ASG` continuation, which


(a) observed behavior
Execute source code while intercepting
(b) documentation as present in the source code
verified against
(c) documented behavior
(d) high-level events specification
specific for this application
(e) application-specific instances
(f) associated run-time values

(a) observed behavior

```
..
event(60,cntEntered('ASG',13..1,['ASG','print', 'exit'])).
event(61,cntExited('ASG',13..1,['print', 'exit'])).
..
```

(b) documentation as present in the source code

```
/*---------------------------------------------*/
/* ASS                                         */
/* expr-stack: [... ... ... ... DCT VAL] ->    */
/*             [... ... ... ... ... VAL]       */
/* cont-stack: [... ... ... ... ... ASS] ->    */
/*             [... ... ... ... ... ...]       */
/*---------------------------------------------*/
static _NIL_TYPE_ ASG(_NIL_TYPE_)
 { ... }
```

(c) documented behavior

```
1 cntDocumented('ASG',['ASG'|R],R).
2 cntDocumented('REF',['REF'|R],['REF','APL'|R]).
3 ...

4 behavioralModel :-
5   □(when(cntExecuted(Name,Before,After),
6            cntDocumented(Name,Before,After))).

7 cntExecuted(Name,StackBefore,StackAfter) :-
8   cntExited(Name,_,StackAfter),
9   ●^t cntEntered(Name,_,StackBefore).
```

(d) high-level events specification

```
intercept(before,continuationEntry,
 event(time,cntEntered(cntName,cntPtr,cntStack))).

intercept(after,continuationExit,
 event(time,cntExited(cntName,cntPtr,cntStack))).
```

(e) application-specific instances

```
continuationEntry(Construct,Path) :-
  inContinuation(Construct,Path),
  functionEntry(Construct,Path).
continuationExit(Construct,Path) :-
  inContinuation(Construct,Path),
  functionExit(Construct,Path).

continuation(Construct) :-
  isFunctionDefinition(Construct),
  expressionIn(Construct,Expression,_),
  picoStack(Expression).
```

(f) associated run-time values

```
keyword(cntName,C,P,Expansion) :-
 continuationName(C,P,Name),
 concat(['log("',Name,'");'],Expansion).
```

Figure 2: Source code and corresponding behavioral documentation extracts of the Pico interpreter.

implements the execution of an assignment expression. Its behavior is documented in terms of expression and continuation stack transformations shown in Figure 2b. The expected elements of the continuation and expression stack are written down between square brackets and separated by spaces. The top of the stacks are located on the right side. The dots represent possible elements on the stack that are of no importance to the assignment continuation as they are left untouched during its execution. The expected configuration of the stack before the execution of the continuation is located to the left of each arrow, while its configuration after the execution is located to the right.

This human-readable schema sufficiently documents the semantics of the assignment expression as implemented by the ASG continuation. At a certain point during the evaluation of a program, the Pico driver loop pops the assignment continuation (ASG) from the continuation stack and executes it. The ASG continuation in turn expects a variable environment (represented by the dictionary DCT) and the value that is to be assigned VAL) to be on the expression stack. They are both needed to store a variable-value binding in the current environment. At the end of its execution, the ASG continuation pushes the assigned valued VAL on the expression stack, as Pico assignment expressions evaluate to their right-hand side.

4.2. Documenting Program Behavior

Over the course of some years, several modifications to the Pico source code have been made. We therefore wanted to verify whether the actual dynamics of the continuation stack still matched the documented behavior. Following the recipe outlined in Section 2, we first have to identify the high-level events in terms of which we will document the behavior. Following the original documentation, we chose to model the execution of a continuation as a high-level event whose associated run-time values are the configuration of the continuation stack before and after the execution. In this paper, we will only describe and verify the evolution of the continuation stack, but our approach can be easily applied to the expression stack as well.

We are now ready to specify our behavioral model as assertions over the high-level events we just identified. We will start by transforming the original documentation into a format readable by our platform. As the model specification abstract in Figure 2c shows, we use logic facts of the form `cntDocumented(CntName, StackBefore, StackAfter)`. The documented configuration of the continuation stack before and after the continuation's execution is represented by a list whose first element is the top of the stack. In Prolog, the partial list `['ASG'|R]` matches any list starting with the element `'ASG'` while the rest of the list is bound to the variable R. We use this feature to represent the ellipses from the source code comments. The first line of the model therefore specifies that at the start of the ASG continuation's execution, there should be an ASG assignment continuation on top of the stack. After its execution, the continuation has to be popped from the stack.

The assertions in our behavioral model need to state that whenever a continuation is executed, the configuration of the continuation stack before and after its execution should match the ones documented in the model. We accomplish this on lines 4–6 of Figure 2c by expressing a temporal relation between the `cntExecuted` and `cntDocumented` predicates. The `Before` and `After` variables used within these predicates match in order to enforce that the observed complete stack configurations agree with their partial specifications. As we represented these stacks as concrete Prolog lists and partial Prolog lists respectively, we are relying on Prolog's built-in unification algorithm to perform the actual matching.

As in our running example, the final 3 lines of the extract link the predicates used in the behavioral model to the high-level events that will be observed during the execution of the Pico interpreter. This time, instead of just omitting unwanted information from the recorded events, we are using the temporal operator $\bullet^t$ to express that a continuation has been completely executed once it is exited and was entered t time points ago in the past.

4.3. Application-Specific Instances of High-Level Events

The third step in our recipe consisted of a precise specification of the high-level events used in the program model. The model from the previous section completely relies on just two high-level run-time events: the start and conclusion of a continuation's execution. More importantly, we are interested in the configuration of the continuation stack at those moments in time. We will therefore associate these values with the `cntEntered` and `cntExited` run-time events. The high-level event specification shown in Figure 2d declares these run-time events to be recorded as facts of the form `cntEntered(cntName,cntPtr,cntStack)` and `cntExited(cntName,cntPtr,cntStack)`.

The definitions of the `continuationEntry` and `continuationExit` predicates, which are among the logic predicates the high-level event specification relies on, are depicted in Figure 2e. They are responsible for locating those constructs in the Pico interpreter's source code that represent the entry and exit points of a continuation. The `continuationEntry` rule states that a construct is the entry point of a continuation if the construct is part of a continuation and if it is the entry point of a function as well. As before, the `Path` variable represents the path from the program's parse tree root to the parse tree node bound to the `Construct` variable. It is used by the `inContinuation` clause to check whether the programming language construct is part of a continuation. We will discuss how to positively identify a continuation later on, but for the moment it suffices to recall that they are implemented as functions. The `functionEntry` clause therefore checks whether the construct is the first C statement in a function body.

We could try to base our definition of the `inContinuation` predicate on the C signature shown on line 9 of Figure 2b. This signature is common to all Pico continuations and states that continuations are pointers to functions taking no arguments as well as not returning anything. Such a definition would however result in many false positives. Recalling that continuations invoke other continuations and pass around arguments by respectively manipulating the continuation and expression stack, we have therefore opted to identify continuations based on a more semantical definition shown in lines 7–10 of Figure 2e. The `continuation(Construct)` rule states that a C code construct is a continuation if first of all it is a function and at least one expression in the body of that function manipulates either the continuation or the expression stack. Such expressions are identified in the Pico source code by the `picoStack(Expression)` clause.

We have however neglected to mention three important keywords our specification of the meta model relies on: the `cntName`, `cntPtr` and `cntStack` keywords. As we mentioned before, keywords are used to declare how the information associated with each high-level run-time event specified in the general meta model can be retrieved from an application's run-time state. The `cntStack` keyword is responsible for capturing the run-time configuration of the continuation stack. This process involves code to walk over the continuation stack, but since this code is tightly tied to Pico's internals, it is out of this paper's scope. The keywords `cntName` and `cntPtr`, on the other hand, differ somewhat from the keywords we have seen in the running example. Instead of simply declaring the code it expands to as a logic fact, these keywords are actually logic rules which are allowed to query the application's parse tree to obtain information that is to be incorporated in the expanded source code. The `cntName` keyword, shown in Figure 2f, obtains the name of a continuation from a program's parse tree since it is very hard to obtain a function's name at run-time given ANSI C's limited reflective capabilities.

4.4. Lightweight Consistency Verification

To verify whether the Pico interpreter indeed behaves as indicated by its documentation, we used the interpreter to evaluate a program containing most of the allowed Pico expression types. This way, we obtained high-level execution traces containing information about the dynamics of the continuation stack. An example of such an execution trace is shown in Figure 2a. We verified whether these traces conform to our model specification by launching the logic query `?- behavioralModel`. By doing so, we found several interesting conflicts between Pico's documented behavior and the behavior we observed. One of them was located in the documentation of the `REA` continuation which is executed when an expression is read. The documentation indicated that this continuation just pops the top of the continuation stack during its execution, while in reality the `EXT` and `EXP` continuations were pushed on top of the stack. In addition, several minor naming inconsistencies were detected. The `_eval_exp_` continuation was for instance abbreviated in the documentation as `EXP`, but another continuation already had this name. Such inconsistencies are very likely to confuse programmers studying the documentation. Upon interpretation of our verification results, we were able to adapt the machine-verifiable behavioral documentation to the actual program behavior. Since this documentation is at least as descriptive and human-readable as the original documentation in the source code comments, we adopted the machine-verifiable documentation as the official documentation of the interpreter. This will allow us to keep the documentation and source code in sync as we can easily verify their consistency after future modifications to the interpreter.

5. Related Work

The BEHAVE platform's distinguishing property is that it allows behavioral program documentation to be expressed as temporal assertions over events at the conceptual level instead of a program's implementation level. Expressing behavioral assertions through a temporal logic language is, however, not new and quite common in heavyweight verification techniques exhaustively exploring a program's possible states.

TRIO [8], for instance, is an established executable first-order temporal logic specification language for real-time systems. Similar to the metric temporal logic we employ, it provides a metric to indicate distance in time between events and length of time intervals. Outside of the heavyweight program verification domain, it has also been applied to check whether an execution history of a real-time system satisfies its specification. Our BEHAVE platform employs a metric temporal logic programming language as the specification language which allows temporal operators to be combined into reusable higher-order logic rules. As these rules can express often recurring temporal patterns, users don't necessarily have to remember their idiomatic expression in temporal logic.

The BEHAVE platform performs lightweight verification through a dynamic analysis, while enabling developers to determine the kind of run-time events that are intercepted during a program's execution. In the following section, we will therefore compare some of the existing applications of dynamic analysis that are closest to our work.

5.1. Related Dynamic Analyses

C-Patrol [14] is an assertion insertion system for C. Assertions are expressed as C expressions in source comments which are textually converted into executable code by a preprocessor. The resulting code is inserted at locations identified by extensional directives in the comments. Assertions can therefore not be verified without altering the code of the application under investigation, nor can they be reused among different applications. There is no explicit notion of a run-time event and thus also no language to express assertions as computations over these events.

CCI [13] 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. The event meta model is thus procedural. Through a basic pattern matching language, composite events can be composed out of statically selected low-level events. The value associated with the higher-level events is however predetermined by its constituents. As CCI is a program monitor notification tool, it provides no language specifically tailored to reasoning about intercepted events.

Auguston et al. [2] present the interesting procedural assertion specification language FORMAN. It provides quantifiers, boolean and aggregate operations over events and target program values. 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 a program documentation setting, we prefer the more declarative nature of the 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 developer involvement.

5.2. Aspect-Oriented Programming

Aspect-Oriented programming languages feature a pointcut language as an extension to a base programming language which allows aspects to localize the implementation parts of a crosscutting concern. In our approach, we could have used a pointcut language for the specification of the high-level run-time events. However, mainstream pointcut languages aren't declarative, which would have conflicted with our human-readability requirement. Moreover, for our experiments, we needed to identify all continuations in the Pico interpreter's code. We didn't base the identification on a function's signature, but on a more semantic rule relying on the expressions in its body. Most AOP languages don't offer the ability to reason about a function's body. Aspicere's pointcut language [1], one of the most expressive for C, can not access macro's in their pre-expanded state which we needed for our experiments. Finally, we would still have needed an expressive medium to describe a program's behavior in and have the resulting documentation verified. To encourage daily use, it is preferable to have all elements of this documentation in the uniform expressive logic programming paradigm.

Stolz et al. [12] however present an interesting lightweight Java program verification approach based on AOP technology. Behavioral assertions are temporal logic formulae over AspectJ pointcuts and are checked on-line during a program's execution. These assertions are however less suited to document a program's behavior with. First of all, the logic formulae cannot assert anything about future events. Secondly, the formulae can only reason about the limited amount of events accessible by AspectJ pointcuts. As the verification is performed on-line, asserting temporal relations between nested events is problematic. Finally, there is little support for the incremental specification of assertions, requiring programmers to remember the idiomatic temporal logic expressions for commonly occurring temporal relations.

6. Conclusion and Future Work

In this paper, we presented the BEHAVE platform wherein behavioral documentation can be expressed as assertions over high-level events freely chosen by the developers. The platform verifies the consistency of the documented behavior and the program's actual behavior in a lightweight manner through dynamic analysis. More conventional dynamic analysis applications demand assertions to be expressed over a rigid set of low-level run-time events that are directly related to programming language constructs. In contrast, a developer using the BEHAVE platform has complete control over the kind of high-level events that arise during an application's execution and the information about a program's run-time state that is associated with each event. Developers are therefore able to express documentation in terms of events at the conceptual level instead of the implementation level.

The BEHAVE platform strives to reconcile the human-readability and lightweight machine-verifiability desirable properties of behavioral documentation approaches by introducing conceptual events from the application's domain into the documentation. The resulting documentation is lifted from the concrete implementation level to a higher conceptual level. The introduction of high-level events ensures, in combination with an expressive specification language, that the resulting models are descriptive, concise and hence ideally suited to knowledge transfer. The introduction of high-level events in execution traces moreover allows for the goal-oriented verification of larger applications as the resulting traces comprise, in general, fewer events than those consisting of low-level run-time events.

In this paper, we have evaluated the feasibility of our approach in an experiment on the Pico interpreter by translating the behavioral documentation found in its source code comments into the machine-verifiable format understood by our platform. We are currently preparing experiments to evaluate the human-readability aspect of the resulting documentation empirically in a very specific knowledge transfer setting: the transfer of knowledge between a knowledgeable teacher and a student new to an algorithm or datastructure.

References

[1] B. Adams and T. Tourwé. Aspect-Orientation in C: Express Yourself. In *Proc. of the AOSD Worksh. on Software-engineering Properties of Languages for Aspect Technologies (SPLAT05)*, 2005.

[2] 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.

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

[4] C. Brzoska. Temporal Logic Programming with Metric and Past Operators. In *Proc. of the Worksh. on Executable Modal and Temporal Logics*, pages 21–39, 1995.

[5] E. M. Clarke, 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 (TOPLAS86)*, 8(2):244–263, 1986.

[6] P. Cousot and R. Cousot. On Abstraction in Software Verification. In *Proc. of the 14th Intl. Conf. on Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 37–56, 2002.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE99)*, pages 411–420, 1999.

[8] M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. *ACM Trans. Softw. Eng. Methodol.*, 1994.

[9] P. Flach. *Simply Logical*. John Wiley, 1994.

[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] M. A. Orgun and W. Ma. An Overview of Temporal and Modal Logic Programming. In *Proc. of the 1st Intl. Conf. on Temporal Logic (ICTL94)*, pages 445–479, 1994.

[12] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Proc. of the 5th Intl. Worksh. on Run-time Verification (RV05)*, 2005.

[13] 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.

[14] H. Yin and J. M. Bieman. Improving Software Testability with Assertion Insertion. In *Proc. of the Int. Testing Conference*, 1994.