A FORMAL LANGUAGE AND ANALYSIS TOOL FOR BLACK BOX SPECIFICATIONS

Gabriel J. Ferrer
Department of Mathematics and Computer Science
Hendrix College
Conway, AR 72032
(501) 450-3879
ferrer@hendrix.edu

ABSTRACT

The black box specification, developed by Harlan Mills, addresses the problem of software errors that result from failing to properly specify a response for an input scenario. Each black box models how an artifact responds to a particular input from its environment. This response depends on both the current input and the entire history of interactions it has had with the environment.

We have observed that students find the black box concept useful and comprehensible, but tedious and error-prone as well. In order to enhance the utility and accessibility of this technique, we have developed a formal specification language and analysis tool for black box specifications. The analysis tool verifies whether a black box is a well-formed specification. To this end, it ensures that a response is specified for every possible combination of inputs from the environment, that every condition is logically disjoint with every other condition in the specification, and that every condition in the specification matches at least one potential input scenario. We have evaluated the utility and performance of the tool with two different groups of undergraduate students.

INTRODUCTION

Many errors in software can be attributed to the failure to properly specify a response for an input scenario. A black box specification [1][3][4] models how a software artifact interacts with its environment. A black box does not represent any internal state of the artifact; it only represents externally observable interactions with its environment. This model can assist a software engineer in ensuring that a response for every input scenario has been specified [3].

We have previously introduced undergraduate students to black box specifications based on the presentation by Prowell et al [3]. While students informed us that the process was useful, they also complained about the tedium involved in constructing and verifying the specifications. To address this problem, we have developed a formal language and modeling tool. The models built in this language can be algorithmically analyzed by the tool to ensure that the black box is a well-formed specification. The analysis tool ensures that a response is specified for every possible combination of inputs from the environment, that every condition is logically disjoint with every other condition, and that every condition matches at least one potential input scenario.

The analysis tool is also able to generate an animation of the specified software artifact. This animation is a GUI with a button for each stimulus. It shows both the current response and the entire list of responses issued so far. This allows both the specifier and domain experts to validate the black box against the requirements.

We begin by presenting our representation of black box specifications. Next, we describe the verification algorithms employed by the analysis tool. We then discuss the results from our evaluation of the tool with two different groups of undergraduate students. We first analyze the usability and utility of the tool, followed by an analysis of efficiency concerns. We then discuss related work.

BLACK BOX SPECIFICATION

We call each input from the environment a *stimulus* and each output a *response*. A *history* is a finite-length sequence of alternating stimuli and responses. A well-formed black box specification is a function from the domain of the set of history-stimulus pairs to the range of the set of responses. As a function, it will deterministically respond in the same way for each history-stimulus pair.

We represent a black box by a table with three columns: the History Set, the Current Stimulus, and the Response. Each stimulus and response is a discrete symbol. A history set is a (potentially infinite) set of histories. History sets are formally specified using history patterns. Each history pattern maps a history to a true/false value, indicating whether or not that history is a member of the set that the pattern specifies.

The formal language for history patterns is constructed as follows. The most primitive language element is the name of a stimulus. This matches any history with exactly one stimulus-response pair that includes the named stimulus. A pair can also be specified in terms of a named response; in this case, the syntax requires the keyword "response" prior to the named response. The keyword "stim" matches any single stimulus-response pair regardless of names. The keyword "any" matches any history, even if it is of zero length. The keyword "none" matches zero-length histories only.

These language elements are composed into more general patterns as follows. The ":" operator concatenates any two patterns. The concatenated pattern will match any history that can be divided so that its first part matches the pattern before the ":" while the second part matches the pattern after the ":". "Count" returns the number of disjoint sub-histories of a history that match its argument. It matches sub-histories greedily; hence, it prefers shorter sub-histories to longer ones in ambiguous situations. The values generated by count can be compared with each other using the standard arithmetic comparison operators ($<$, $>$, $\geq$, $\leq$, $=$). They can also be incorporated into linear combinations with addition and subtraction. The construct "includes <arg>" is equivalent to "count <arg> $\geq$ 1". Since determining membership in a history set is a boolean operation, the standard boolean operators "and", "or", and "not" are available.

In order to bridge the gap between the pattern language and an application domain, macro patterns can be defined. Each macro is a binding of a name to a pattern. Whenever the name of the macro is used in an expression, the pattern bound to it is used to determine whether a particular history is a match.

We will illustrate these language elements using the specification of a simple text editor as an example. The black box specification is given in Table 1. The text editor has three operations: appending a character onto the end of the buffer, removing a character from the end of the buffer, and saving the buffer. To describe these operations we use three stimuli: Char, Back, and Save.

Anytime a character is typed, it is appended to the end of the buffer. Hence, the stimulus "Char" can be specified using a single row. Backspacing removes a character only if at least one character is present. If the buffer is empty, backspacing is impossible. Hence, we use two rows to specify the "Back" stimulus. Characters are present only if the number of character additions exceeds the number of successful deletions. Hence, we define the macro "canBackspace" to represent "count Char > count response Del". Since the stimulus "Back" does not always represent a successful deletion, the history pattern matches the response "Del" in order to ensure that actual deletions are being counted.

We specify that the text editor will respond to a request to save the buffer only if the buffer has changed since the last save, which again requires two rows. We will first define the idea of a "change". We will use that definition to specify "the buffer changed since the last save". As a change involves either the addition or successful deletion of a character, we define the macro "change" to represent "Char or response Del". A properly saved buffer cannot have any changes subsequent to being saved. Hence, we define the macro "isSaved" as "any:response Saved:(not includes change)". We use this macro in both rows to indicate that a save should succeed if and only if the buffer has not been saved.

Table 1: Black Box for a Simple Text Editor

History Set	**Stimulus**	**Response**
Any	Char	Add
canBackspace	Back	Del
not canBackspace	Back	None
not isSaved	Save	Saved
isSaved	Save	None

ANALYSIS TOOL

The purpose of the analysis tool is to automatically prove whether all possible history-stimulus pairs have specified responses, whether the black box corresponds to a function, and whether each row in the function has a non-empty domain. We call these properties *completeness*, *row disjunction*, and *well-defined rows*, respectively. We employ two different search strategies to check these properties:

• History generation: Use best-first search to generate a history that is a member of the specified history set. The algorithm will produce a history if it can, but might not halt if it fails. Each syntax element in the language makes a heuristic estimate of the number of additional stimulus-response pairs that need to be appended to a given history to enable it to be a member of the set that the syntax element specifies. Non-terminal language elements combine the estimates of their child nodes.

• Satisfiability checking: The boolean satisfiability of the history pattern is checked. Every element of the expression other than "and", "or", and "not" is considered to be a propositional variable. If the expression is unsatisfiable, the set is proven empty. If it is satisfiable, nothing is concluded, as relationships between distinct propositional variables have not been modeled.

These search strategies complement each other. For sets in which no history exists, a proof of unsatisfiability can quickly halt a history generation that might otherwise fail to terminate. For sets in which a history exists, the history generator can provide a constructive existence proof. In order to check whether the rows are well-defined, the satisfiability checker determines, for each row, whether its history set is satisfiable. If it is, the history generator tries to generate a history for that history set.

Completeness is checked by synthesizing a boolean expression to represent the set of all histories not covered by a history set for a given current stimulus S. Each history set for S is negated and then wrapped into a conjunction. If this expression is unsatisfiable, then completeness is proven for S. If a concrete history is generated for the synthesized expression, it serves as a counterexample to the completeness property. Completeness is proven or disproven for the black box as a whole by checking completeness for every stimulus listed as a current stimulus. One counterexample is presented to the user for each stimulus with an incomplete history set specification.

Row disjunction is checked in a similar way. For each pair of rows for a given current stimulus S, the search algorithms attempt to prove that the intersection of their history sets is the empty set. If the proof fails, a counterexample history for each conjoined pair is presented to the user.

The property of well-defined rows is checked by generating a history for each history set in the History Set column of the black box table. If a history can be generated for the set in a given row, that row is reported as well-defined. If the set can be proven unsatisfiable, that row is reported as not well-defined.

For both the GUI animation and for processing during verification, the analysis tool must perform row matching. When a black box is presented with a history and a current stimulus, a string-matching algorithm determines the matched row and current response. The algorithm employs top-down dynamic programming to avoid redundant recomputations of subsequences when using the ":" operator.

EVALUATION

We examined several black box specifications created by undergraduate students in upper-division software engineering courses at two different institutions. Group 1 was a Software Engineering course for juniors and seniors taught by the author. Group 2 was a survey course in Formal Methods at a different institution, containing juniors, seniors, and graduate students. Group 1 spent two weeks of class time learning to create black boxes, submitting solutions at the end of this period. Later, they implemented the specified applications. Group 2 heard one lecture on the topic and turned in black boxes after one week. They did not implement the specified artifacts, although the author did implement the application specified by his own solution to the exercise. The Group 1 students used the tool to build specifications for a spreadsheet-like tool ("ListMaker"), a family-tree program, and a video game. The Group 2 students specified a simple web browser. Table 2 contains statistics about these projects, including lines-of-code for the finished applications. For the web browser, "lines of code" refers to the author's implementation.

Except for two instances from Group 2, the student-produced black boxes met the requirements well. Except for one of the three boxes from Group 1 and two of the seven boxes from Group 2, all boxes passed all of the verification algorithms. Three students created boxes that met the requirements but had incompletely specified stimuli. In two cases, the student authors were confused about how to handle unreachable histories. To address this, following the suggestion of a third student we plan to introduce a special

"Impossible" response to model this type of situation. In Group 2, some black boxes failed to meet the requirements resulted because of failing to use the "response" keyword. (This was due to confusing instructions from the author.) Five of the students suggested that something like "response" be added to the language, confirming its utility.

Table 2: Project Statistics

Program	Rows	Stimuli	Responses	Nodes	Lines of Code
Genealogy	39	23	26	174	905
Game	35	26	33	172	935
List-Maker	22	13	18	141	1210
Browser	12	7	10	216	1322

Many students spoke well of the interactive animation. One student stated: ``The test prototype GUI functioned in a way that would be very similar to a web browser, which made testing very intuitive." Several others mentioned that the animation was invaluable in finding where the specification disagreed with the requirements.

Regarding performance, best-first search required very few node expansions, as seen in Table 2. The history patterns were small, so satisfiability checking was fast. To analyze best-first search in more depth, we determined which imprecisions are present in the heuristics relative to the inputs. The "and" keyword creates imprecision by minimizing the estimate of its arguments. Observing student work, we found that "and" is used as a filter; it was always the case that one argument was essentially negative and had a precise distance of zero to achieve. Estimates for "response" use a complex algorithm that is imprecise in combination with ":". In practice, however, we saw combinations such as "any:response Del" in which a zero estimate for "any" avoided trouble. For the web browser, modeling the "Forward" and "Back" buttons required arithmetic that did introduce underestimates. This explains the significantly larger number of nodes expanded for the relatively small black box created.

RELATED WORK

Black box specification was introduced by Mills [1]. In that paper, history sets are described by natural-language boolean conditions. This paper expands upon his work by formalizing history sets in order to facilitate automated verification. Mills incorporated black box specification into the Cleanroom Software Engineering process, which is described in detail in books by Prowell et al [3] and Stavely [4].

Prowell et al [3] use sequence-based specification to verify black box completeness. In sequence-based specification, one enumerates every possible stimulus sequence in order of length, with all combinations and permutations considered systematically. Sequences that are impossible or equivalent to a shorter (canonical) sequence are not extended further. Enumeration stops when no sequences can be extended. The canonical sequences represent a logically complete and disjoint black box. Their proto-seq tool [5] confirms completeness by ensuring that no eligible sequences remain to be extended. The potentially infinite set of stimulus sequences is reduced into manageable subsets by the specification author marking related sequences as equivalent during enumeration. In contrast, our tool requires the author to directly specify the subsets into which the set of all stimulus sequences is to be partitioned.

The system of Heimdahl and Leveson [2] includes tabular specification with support for checking completeness. Heimhahl and Leveson's system specializes in domains for which the software artifact is part of a complex embedded application. This introduces complexities in both the specification language and in flagging false positives that would be distracting in light of our own goals.

Kiniry and Zimmerman [6] describe how they incorporated formal methods into their software engineering courses. They demonstrated that automated analysis tools for formal methods greatly improved undergraduate student response to the topic.

CONCLUSION

Our analysis tool has been shown to be effective for building and analyzing black box specifications for small GUI applications. The formal language presented in this paper is a first step towards developing formal specifications of the boundary between software artifacts and their environments that can be automatically verified and systematically refined into formally annotated executable code. This language has been shown to be quick and easy to learn for upper-division undergraduate students.

Several applications specified with black boxes have been developed to completion; larger applications need to be specified and developed in order to explore scalability in more depth. In general, proving completeness, row-disjunction, and row-definition is intractable. While the specifications studied so far have avoided the cases that provoke poor performance from our algorithm, more specifications need to be studied to determine whether good performance should be the expected result in general.

To build on the positive responses to the animation of the specification, we are currently developing a GUI building tool with integrated black box support. Each GUI component is specified by a black box. The responses from input-oriented components are mapped to the black box for the application. The responses from the application are then mapped to output-oriented GUI components. When completed, this system will enable students to simultaneously prototype and formally specify desktop applications.

Our analysis tool is open-source software and is freely available for downloading from http://boundalyzer.sourceforge.net under the GNU General Public License 3.0.

REFERENCES

[1] H. D. Mills. Stepwise refinement and verification in box-structured systems. IEEE Computer, 21(6):23–36, June 1988.
[2] M. Heimdahl, and N. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. IEEE Trans. on Software Engineering, 22(6):363–377, June 1996.
[3] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. Cleanroom Software Engineering: Technology and Process. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.
[4] A. Stavely. Toward Zero-Defect Programming. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.
[5] Software Quality Research Laboratory at University of Tennessee Knoxville. proto_seq User's Guide, 2005.
[6] J. R. Kiniry and D. M. Zimmerman. Secret Ninja Formal Methods. In Proceedings of the 15th international symposium on Formal Methods, 214-228, Turku, Finland, 2008.