WCS: a Witness and Counterexample Server

Alessandro Fantechi[1], Stefania Gnesi[2],
Robert Meolic[3], and Gianluca Trentanni[2]

[1] Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze, Firenze, Italy
fantechi@dsi.unifi.it

[2] ISTI-CNR, Pisa, Italy
gnesi@isti.cnr.it, gianluca.trentanni@isti.cnr.it

[3] Faculty of Electrical Engineering and Computer Science
University of Maribor, Maribor, Slovenia
meolic@uni-mb.si

Abstract. Witness and Counterexample Server (WCS) is planned to be a suite of tools designed to asses the ability of model checking techniques to increase effectiveness of hardware/software testing activities. It is focused on witnesses and counterexamples produced by model checkers. Currently, a BDD-based model checker with the ability of generating linear witnesses and counterexamples, and witness and counterexample automata is available. Witness and counterexample automata are finite automata recognizing a set of all finite linear witnesses and counterexamples for a given formula over a given model, respectively. WCS is organized as a web service, in order to provide public access to the functionality of verification tools. It was developed exploiting Zope, an open source web and application server that allows dynamic server pages generation and interaction with the server file system through the highly compatible Python platform.

1 Introduction

Witnesses that show why a formula is satisfied and (more often) counterexamples that show why it is not satisfied over a model have been used as useful diagnostic information since the first applications of model checking technology. They are usually returned by model checkers in the form of a computation path. However, only for certain kinds of formulae a computation path is able to explain completely the reason of satisfaction or missed satisfaction. Only recently, a greater interest was raised on the study of the relations between the formulae and their counterexamples, on one side looking for richer forms (e.g. tree-like counterexamples [2], proof-like counterexamples [7] for CTL and [**?**] ...), on the other side establishing the subsets of the logics whose formulae guarantee linear computation paths as counterexamples which completely explain the failure (e.g. ACTL det [10] and LIN [1]).

Our work in this field has been motivated by another trend that has consolidated in the recent years, that is the usage of counterexamples as an help to

generate test cases. When testing or simulation does not reach an adequate level of coverage (defined by some code coverage metrics, such as statement coverage and branch coverage) new test cases have to be defined, but the process of manually producing test cases for "corner-case" scenarios is time consuming and error prone. Model checking and counterexamples can help: if we have a model of the system, and we model-check on it a formula expressing "there is no uncovered point", a counterexample returns a computation path with enough information to generate the proper test case. In the adoption of this principle with a "conquer and divide" approach in order to attack the typical state space explosion problem, a more effective option is to have the model checker generating not a single counterexample, but all the counterexamples for the given formula. We refer to [5] for more details.

In [8], we identified a fragment of action computation tree logic (ACTL) [9] which is suitable for application in the field of test case generation. Moreover, we proposed a suitable class of witnesses and counterexamples V such that for the given ACTL formula and LTS there exists a suitable witness (counterexample) in V, denoted as V-witness (V-counterexample), which explain all reasons of validity (failure) of given ACTL formula over given LTS explainable by finite linear witnesses (counterexamples), is as small as possible, and is computable by an effective algorithm. The set of all V-witnesses (V-counterexamples) forms a regular language and therefore it can be represented as an automaton, which we call a witness automaton (counterexample automaton).

We created a recursive algorithm that, given a LTS and a formula from the fragment of ACTL given, generates the witness or counterexample automaton WCA for the given formula over the given LTS. If the formula holds in the initial state of LTS, the generated WCA is a witness automaton, otherwise, it is a counterexample automaton. The algorithm for witness and counterexample automata generation basically works by following the given LTS and using unfolding when necessary, with an unfolding depth of at most the length of the formula. Therefore, the complexity is not higher than the size of the LTS (states and transitions) times the length of the formula. This is exactly the same complexity of an explicit model checking algorithm which has to be employed to compute the labeling of the LTS. We have implemented the algorithm as an extension of a BDD-based ACTL model checker and made a stand-alone on-line demo application.

2 Witnesses and counterexamples

Given a model $\mathcal{M}$ and a formula φ such that $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \varphi$), a witness (counterexample) is a structure $\mathcal{R}$, in relation with $\mathcal{M}$, that completely shows one of the possible reasons why $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \varphi$). The type of the relation between $\mathcal{R}$ and $\mathcal{M}$ determines the nature of the witnesses and counterexamples. Linear witnesses and counterexamples are finite or infinite computation paths over $\mathcal{M}$. For a particular subsets of the logics they completely explain one of the reasons of validity or failure of a given formula over a given model [1]. In

general, they contains enough information for applications such as debugging of hardware and software design [4, 6].

There have been recently some attempts of introducing richer forms of witnesses and counterexamples, which are defined as non-linear structures related to the original model $\mathcal{M}$. It has been recognized that not all richer forms of witnesses and counterexamples fits all applications. Therefore a notion of a *viable* class of witnesses and counterexamples for a particular application has been introduced [2, 8]. A class of witnesses and counterexamples $\mathcal{V}$ is viable for a particular application if it meets the criteria of:

- **Completeness**. All reasons of validity (failure) of a given formula over a given model, which are important for the particular application, can be explained by a single witness (counterexample) in $\mathcal{V}$.
- **Intelligibility**. Witnesses (counterexamples) in $\mathcal{V}$ are specific enough to suit the particular application (e.g. simple enough to be analysed by engineers).
- **Effectiveness**. There exist effective algorithms for generating and manipulating witnesses (counterexamples) in $\mathcal{V}$.

Examples of non-linear structures introduced to support various applications are tree-like counterexamples [2],.., [3], proof-like counterexamples [7], witness and counterexample automata [8]. The latter are defined as:
A witness (counterexample) automaton for an LTS $\mathcal{M}$ and a formula φ is an automaton which recognizes the language of all $\mathcal{V}$-witnesses ($\mathcal{V}$-counterexamples) of φ over $\mathcal{M}$.

3 Witness and Counterexample Server Architecture

Witness and Counterexample Server (WCS)has been built to support the generation of Witness/Counterexample Automata and is planned to be an online suite of tools. Currently, a BDD-based model checker with the ability of generating linear witnesses and counterexamples, and witness and counterexample automata is available. It is called *WCA generator for ACTL* and basically it

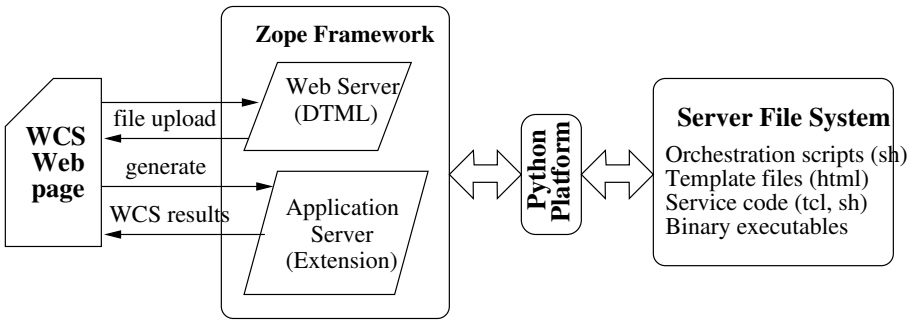

Fig. 1. Witness and Counterexample Server Architecture

takes a CCS specifications and a formula of an action-based logic (ACTL) and it creates a witness or a counterexample automaton. It can also explain the obtained results. Witness and counterexample automata can be created only for the subset of ACTL which guarantees finite linear witnesses and counterexamples defined in [8]. In Fig. 1 the WCS architecture is shown.

WCS was developed exploiting Zope [11], an open source web and application server that allows dynamic server pages generation and interaction with the server file system through the highly compatible Python platform. Starting from the WCS start page, the user can upload files browsing the local file system or by means of a *cut-and-paste* from plain text files. The ability to browse among local files is usually delegated to common web clients like Internet Explorer or Mozilla; the ability to retrieve the file from the client file system, and visualize it, is realized taking advantage of simple DTML code calling the Zope built-in *read function*. WCS exploits simple javascript and further DTML code to open the output window and to control that any request is definitely sent in the correct form. In Fig. 2 the WCS Web page is shown.

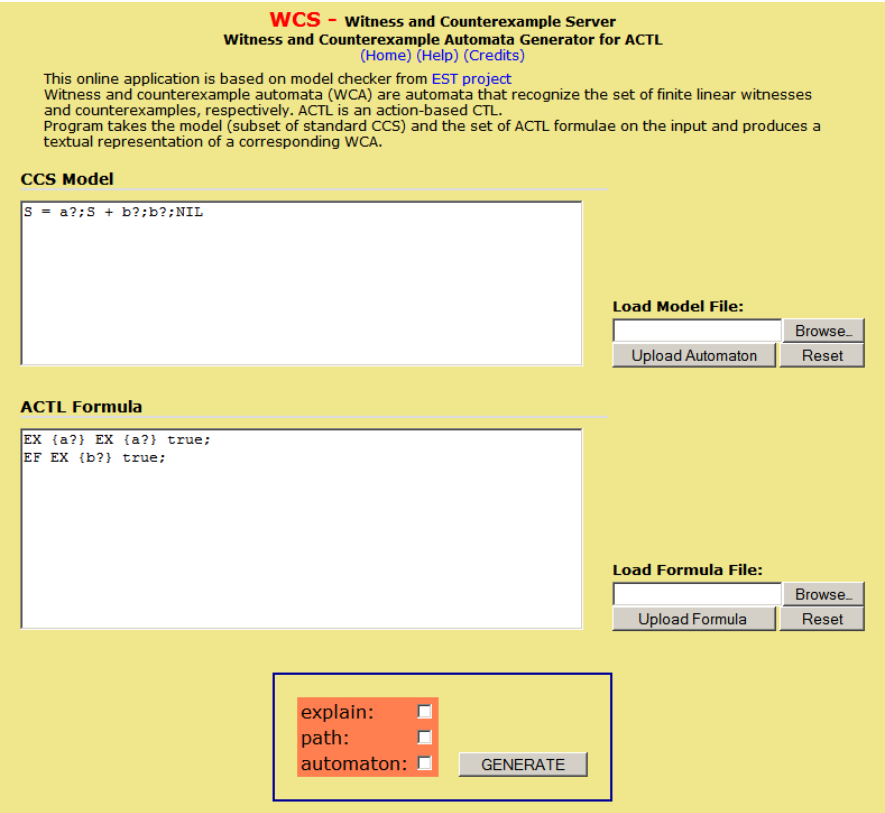

Fig. 2. Witness and Counterexample Server Splashscreen

Generate button allows the user submit the request to the server. The content of text areas are bundled via DTML with the chosen parameters and submitted to a Python script registered as an *external method*. The external methods are made of unrestricted code that output on the Zope framework with the server system allowing system calls and controlled execution of programs. Actually the WCS is running on a Unix Sun Solaris machine and its core consists of statically linked binary executables. Through the Python platform, the request data is input by a main orchestration CSH script that builds the temporary execution environment, wraps the binary executable, runs it passing the data and parameters as inputs, and picks up its response after execution. This orchestration script controls the right execution of any single step of the process and generates a displayable execution log, rising some exception if needed. Other essentials operations this script performs are the parsing of the output to avoid its misunderstood rendering by the web browser and the building The final web page is realized gluing static HTML web page templates with the output itself, and is sent backward through the standard output to the Python platform that passes it up to the Zope framework.

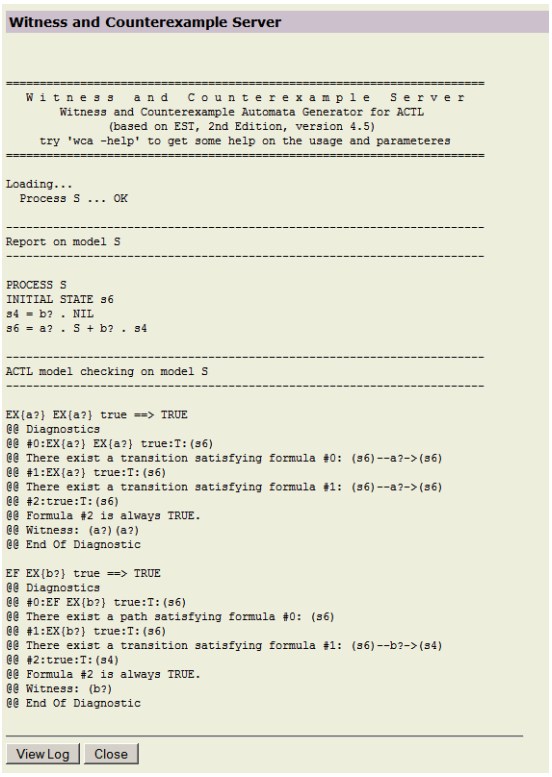

Fig. 3. A Witness and Counterexample Server execution result

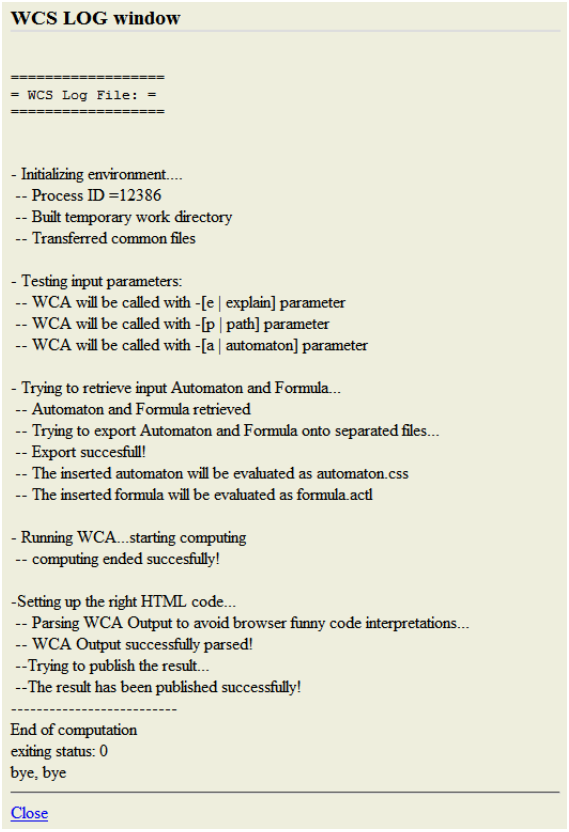

Fig. 4. Witness and Counterexample Server Log

As an implementation choice, the WCS results are shown in a separate window. The output (Fig. 3) is rendered as plain text and the execution log is available too (Fig. 4).

In Fig. 5, two results from WCA generator for ACTL are shown. In general, the size of obtained automata is the size of the model × the length of the formula.

To give an idea of the size of generated automata, we report the data concerning the classical dining philosophers results when looking for the possibility for the first philosopher to eventually eat (formula EF EX eat1 true):
3 philosophers: LTS = (99 states, 240 transitions, WCA = (106 states, 245 transitions, 8 final states)
4 philosophers: LTS = (465 states, 1508 transitions), WCA = (502 states, 1542 transitions, 38 final states)
5 philosophers: LTS = (2163 states, 8770 transitions), WCA = (2338 states, 8941 transitions, 176 final states)

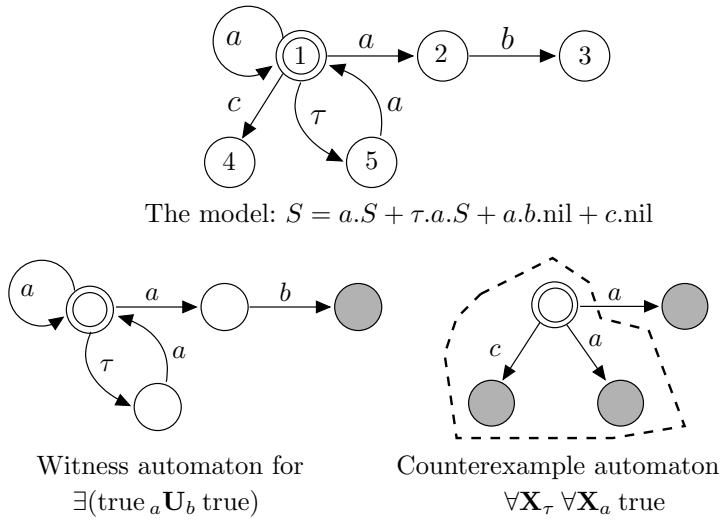

Fig. 5. Two examples of generated automata. The second automaton contains some redundancy which can be eliminated by a simple minimization

As shown in Fig. 5, the resulting WCA is not minimal; a recent addition to WCS is the integration of an existing minimization tool, taking advantage of the web service architecture. Indeed, the adopted web service structure will enable the composition of several tools over the web in order to explore and evaluate the integration of different verification tools.

References

1. F. Buccafurri, T. Eiter, G. Gottlob, N. Leone. On ACTL formulas having linear counterexamples. Journal of computer and syst. sciences, 62(3), 2001, pp. 463–515.
2. E. M. Clarke, S. Jha, Y. Lu, H. Veith. Tree-like Counterexamples in Model Checking. In *17th IEEE Symp. on Logic in Computer Science (LICS)*, 2002, pp. 19–29.
3. R. Cleaveland .... CAV 2002.
4. F. Copty, A. Irron, O. Weissberg, N. Kropp, G. Kamhi. Efficient Debugging in a Formal Verification Environment. In *Conf. On Correct Hardware Design and Verification Methods (CHARME)*, LNCS 2144, 2001, pp. 275–292.
5. A. Fantechi, S. Gnesi, A. Maggiore. Enhancing test coverage by back-tracing model-checker counterexamples. In *Int. Workshop on Test and Analysis of Component Based Syst. (TACOS)*, Electronic Notes in Comp. Sci., 2004.
6. M. Glusman, G. Kamhi, S. Mador-Heim, R. Fraer, M. Vardi. Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. In *Tools and Algorithms for the construction and analysis of syst. (TACAS)*, LNCS 2619, 2003, pp. 176–191.
7. A. Gurfinkel, M. Chechik. Proof-Like Counter-Examples. In *Tools and Algorithms for the construction and analysis of syst. (TACAS)*, LNCS 2619, 2003, pp. 160–175.
8. R. Meolic, A. Fantechi, S. Gnesi. Witness and Counterexample Automata for ACTL. In *24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*. LNCS 3235, 2004, pp. 259–275.

9. R. De Nicola, F.W. Vaandrager. Actions versus State Based Logics for Transition Systems. Proc. Ecole de Printemps on Sem. of Conc., LNCS 469, 1990, pp. 407–419.
10. M. Maidl. The Common Fragment of CTL and LTL. In Proc. 41th Symp. on Foundations of Computer Science (FOCS) , pp. 643-652, 2000.
11. The Zope Book — the official on line Zope manual.
`http://zope.org/Documentation/Books/ZopeBook/2_6Edition/`