ODEN: Assertion mining for behavioral descriptions

Alessandro Danese, University of Verona, Italy, alessandro.danese@univr.it
Tara Ghasempouri, University of Verona, Italy, tara.ghasempouri@univr.it
Graziano Pravadelli, EDALab s.r.l. & University of Verona, Italy, graziano.pravadelli@univr.it

Introduction

Specification mining (Fig. 1) is an automatic approach for extracting assertions from the implementation of the system under verification (SUV). Its primary goal is to improve the verification and documentation process by making available a matching between a manual definition of the expected functionality and a formalization of the actual implemented functionality. In order to automatically extract assertions, some approaches perform a static analysis of the SUV source code. These solutions, despite of their effectiveness, suffer of scalability problems. To overcome this drawback, dynamic approaches have been also proposed that extract assertions by relying only on the observation of SUV’s execution traces. This guarantees a better scalability, even if only “likely true assertions” can be extracted. For this reason a qualification phase is generally implemented in order to discard irrelevant and spurious assertions.

In this context, ODEN is a tool for dynamically extracting likely true assertions by combining static and dynamic techniques [1],[2]. ODEN works with both hardware design and software applications. The tools analyses the execution traces of the system under verification and it generates assertions in the form of temporal relationships between arithmetic/logic expressions over the variables of the SUV. With respect to existing tools, ODEN works on a wider range of abstraction levels (e.g., gate-level, RTL, TLM, SW level, …) and it considers a wider set of temporal patterns to more precisely characterize the behaviours of the SUV.

ODEN overview

The execution flow of ODEN consists of four main steps (Fig. 2):

1. Extraction of cone of influence: Initially, execution traces (and when available also the SUV source code) are analysed in order to understand what primary inputs (PIs) can really affect the value of each primary output (PO). As a result of this first analysis, PIs and POs of the SUV are partitioned in cones of influence, which are separately analysed afterwards.

2. Proposition mining. Given a cone of influence, the execution traces are examined in order to detect the set of relevant arithmetic/logic expressions that are exposed at different time intervals over SUV variables. These expressions are then combined to create logic propositions that represent candidate antecedents and consequents for the target assertions. A proposition trace, per each execution trace, is then created, which specifies whether extracted propositions hold or not at every instant of the execution trace.

3. Assertion mining. Propositions extracted at step 2 are then combined according to a set of temporal patterns. Given a candidate proposition $p_a$ that acts as antecedent and a set of candidate propositions $P_c=(p_1^c, p_2^c, \ldots, p_n^c)$ which act as consequents, the following temporal patterns are currently considered in order to extract the final set of candidate assertions:

   a. $\text{always } (p_a \rightarrow \text{next } p_1^c)$;
   b. $\text{always } (p_a \rightarrow \text{next } [N] p_1^c)$;
   c. $\text{always } (p_a \rightarrow \text{next } (p_e, \text{ or } p_2^c \ldots, \text{ or } p_n^c))$;
   d. $\text{always } (p_a \rightarrow \text{next } [N] (p_e, \text{ or } p_2^c \ldots, \text{ or } p_n^c))$;
   e. $\text{always } (p_a \rightarrow p_b \text{ until } p_e^c)$;
   f. $\text{always } (p_a \rightarrow p_b \text{ until } (p_e^c, \text{ or } p_2^c \ldots, \text{ or } p_n^c))$;
   g. $\text{always } (p_a \rightarrow \text{next } (p_e^c \text{ before } p_b))$.

   Further temporal patterns can be added to extend the tool capability.

   Every candidate assertion is synthesized into an automaton and checked in every proposition trace. If the automaton does not reach the error state, the candidate assertion is collected as a likely true assertion.

4. Stressing phase. Collected assertions are finally converted into checkers and connected to the SUV. Testbenches are applied to simulate the SUV and the checkers searching for counterexamples. When a checker fails, the corresponding candidate assertion is discarded. Only assertions that survive to this stressing phase are definitely collected and provided as output of the tool.

References
