High-Level Information Interface

Deliverable Report: SRC task 1875.001 - Jan 31, 2011

Task Title: Exploiting Synergy of Synthesis and Verification
Task Leaders: Robert K. Brayton and Alan Mishchenko
     Univ. of California/Berkeley

Introduction

This report summarizes progress made in the last period on the use of high-level information to enhance synthesis and verification.

We investigated ways to automatically generate and prove word-level invariants inside our existing bit-level solver, i.e. invariants over bit-vectors in the design-under-test that cannot be expressed as a conjunction of bit-level invariants, such as bit-equalities, bit implications, or local cuts. These have been shown useful in practice, but also there is a class of common and easy-to-generate invariants that are only recognizable and expressible at the word-level, such as bounds on the ranges of a bit-vector (treated as an integer). For example, consider a bit-vector that serves as a pointer in a queue whose size is not a power of two. Thus, not all numbers in the range are allowed, but it is common to have counterexamples when doing induction (not the base case) that has an illegal value of the queue pointer. This class of traces would be eliminated by an invariant that bounds the queue pointer. Such invariants are often easily proved, and can dramatically facilitate the ability to prove other properties.

Other possible types of invariants are comparisons and more general relations between bit-vectors. Our goal is to find good subsets of possible invariants that are easy to generate and useful in practice. Information about bit-vectors can be extracted in various ways. One method is to use slicing information (discussed in a previous report), by processing the RTL without explicit slicing, or even by extracting vectors from flop names.

A stated in this research is to discover and utilize problem-specific inductive invariants to push the boundaries of existing bit-wise advanced model-checking methods such as those in ABC [1]. This may prove crucial in successfully solving difficult problem instances, enabling formal methods in domains where these could not be applied previously.

Bringing in High-Level Information

As part of our exploration of word-level techniques for synthesis and verification, we are integrating an RTL analyzer with ABC to parse a Verilog description so that a word-level view of the design can be generated. Our routine will read in a Verilog implementation of an RTL and generate information about the hierarchy of the design and about bit-sampling and segmentation of its word-level variables (popularly known as slicing - see [[2], [4]] for relevant definitions). These word segments will be treated as potential points of abstraction in subsequent synthesis and verification efforts.

Since our development is based on a commercial Verilog analyzer, we expect to be able read almost any industrial design, parse its hierarchy and make use of this
information. In addition, we are trying to leverage this information for word-level invariant analysis and synthesis.

For synthesis, wide data-paths can become an ‘unnecessary’ overhead. Both synthesis and equivalence checking routine can benefit if we can abstract away wide data-paths. However, intricate interactions between control logic and data-path prevent abstracting away data-paths completely. Most of the time, due to lack of knowledge at the RTL level, we don’t know how much potential for abstraction is available in an RTL.

Using our slicing routine, we propose to observe this interaction between control and data and leverage the scope of abstraction, if any, in synthesis. We are currently developing an RTL slicing algorithm, as proposed in [4]. Our implementation is based on Verific, a Verilog analyzer with a wide user-base in industry [5]. As a first step, we are closely following the assumptions on RTL structure from [4] and we will gradually refine our implementation to handle most of the design primitives seen in industrial designs.

In the next section, we include more details about our developments in this area.

**Interfacing Verific with ABC (Veri-ABC)**

A Verific [5] front-end was integrated with ABC [1] for formal verification. Verific was chosen because of its commercial success and proven track record as a comprehensive and robust RTL front-end tool. Figure 1 shows the normal compilation flow of the Verific front-end in processing RTL designs (in Verilog, VHDL or SystemVerilog languages) into net-list representations. Verific does general-purpose RTL parsing and elaboration, which then are used typically by EDA companies for various specializations in synthesis, simulation and verification tool flows. Our goal is to utilize Verific for synthesis and for property and equivalence checking verification and to use the various synthesis and verification engines of ABC as back-end tools.

Veri-ABC was created, which produces a semantically consistent BLIF or AIG model from Verific net-list structures. This interfaces with ABC providing support for assertions, environments, resets, etc and as well as user-friendly feedback to aid in RTL debugging.

Verific provides an external C/C++ API to access its databases so they can be processed for different purposes. Veri-ABC uses these APIs to access the net-list internal structures, conduct further analysis and produce an AIG or BLIF model for formal verification and synthesis with ABC.
Integration Architecture

Figure 1 shows the compilation flow. After the Verific front-end compilation, a set of databases is generated containing net-list, hierarchy, and book-keeping information.

In our flow, RTL designs, target assertions specified in SVA/PSL assertion languages and environment setup information are compiled through the Verific front-end. Next, Veri-ABC, illustrated in Figure 2, interprets the generated net-list along with other data to produce a formal model in the AIGER format, from which ABC will invoke its engines to attempt to prove or disprove the target properties or to synthesize a design.

This flow expands the ABC capability to handle RTL designs directly, going beyond the restriction of working on existing BLIF or AIGER formats. Plus, it opens up the opportunity to extract higher level information from the RTL design itself to help ABC’s formal verification engines.
To support a practical formal verification or synthesis flow of digital designs specified in industry standard RTL languages, the following were minimal requirements at a higher level for Veri-ABC:

- Design style modeling: this includes clock modeling, latch modeling, reset identification, design constructs modeling, etc. The goal is to model RTL constructs into a finite-state-machine model such that it is semantically consistent with the RTL specification.
- Design initialization through reset state computation
- Design abstraction through cut-points and assumption specifications.
- Book-keeping in general to support back-annotation to the original RTL design for easy debugging, e.g. generate error-traces in VCD waveform format for easier understanding.
- Assertion compilation of standard assertion languages.

**Implemented Commands**

To carry out the above methodology, we implemented the following TCL commands for the afore-mentioned steps respectively, such that an AIGER model is produced.

1. `hier read < blif file>`
   This command reads in the generated hierarchical design from the Verific front-end and conducts design style, clock, latch, and design construct modeling. The hierarchical net-list is flattened into a single net-list internally for later transformation into the AIGER format.

2. `hier_reset`
   This specifies the reset information such that reset state can be computed.
3. **hier_cutpoint and hier_assume**
   These two commands are used to specify cut-points in the design and assumptions for environment or support of an assume-guarantee framework.

4. **hier_write_aig**
   Final AIGER model is written into a file.
   The above allows flexible configuration of the verification problem for specification of clocks, resets, environment constraints and advanced an assume-guarantee verification framework. All configurations are finally transformed into an AIGER model which is language and tool independent.

**Summary of Progress**

Our implementation was successfully run through a FIFO design and an industrial DDR2 controller from the SUN OpenSparc processor design, validating the tool flow and infrastructure. The following observations were made:

- The Verific compiler front-end for RTL designs is robust, configurable and easy to use. It produces and keeps enough higher level information in the databases for later exploration such as at the word-level or procedures that require high-level design information.

- The architecture implemented to interface Verific with ABC is flexible such that changes made to ABC and Verific are minimal. This allows for a clean interface of functionality between Verific, ABC, and Veri-ABC.

- More complex design constructs need to be supported and tested in Veri-ABC to support additional design styles. It is important to have enough testing to ensure correct semantic transformation.

- To complete the verification flow, ABC needs to be enhanced to generate VCD files to aid debugging.

**References**
