Design Understanding: From Logic to Specification*

Goerschwin Fey[†] Tara Ghasempouri[‡] Swen Jacobs[§] Gianluca Martino[†] Jaan Raik[‡] Heinz Riener[¶]
[†]*Hamburg University of Technology* Hamburg, Germany
[‡]*Tallinn University of Technology* Tallinn, Estonia
[§]*CISPA and Saarland University* Saarbrücken, Germany
[¶]*École Polytechnique Fédérale de Lausanne* Lausanne, Switzerland

***Abstract*—We present an outline of the field of Design Understanding and summarize state-of-the-art research in deriving human-understandable knowledge in form of logic properties from an unknown design.**

***Index Terms*—Design understanding, temporal logics, specification, verification, synthesis, assertions, properties**

I. Introduction

Design understanding is the process of reconstructing human-understandable knowledge from an unknown design. This problem arises frequently in practice, e.g., when a part of a design fails or behaves unexpectedly and needs to be debugged, but the initial designer of this part left the design team. In such a situation, the documentation of the design, which is still under development, is often not available, still incomplete, or deviates from the actually implemented behavior. Automated design understanding tools can then be used to assist a human in gaining an understanding of the design and implementing the required changes. Overall, automated design understanding has a great potential to improve fault localization, reduce time-to-market, and improve product quality.

However, design understanding is also a fuzzy process and until today no commonly accepted formalism, notation, and methodology for extracting design knowledge from an unknown implementation exist. We argue that *properties* written in a formal language—the standard formalism in formal verification—are not only useful to describe the input-output semantics of a system, but also capable of describing the interior behavior of a design in a human-understandable manner. Particular temporal logics, that describe relationships between signals of a circuit implementation over time, are attractive. We envision that a design understanding tool derives knowledge in form of temporal-logic formulæ from an unknown design. Such a tool can be useful in various situations:

1) *Reverse engineering*: When the intent of a design is not known or its Register-Transfer Level (RTL) specification got lost over the years, making sense of a gate level description (particularly after heavy optimization) is often impossible for humans. Automated design understanding tools have the potential to isolate a few core properties that give an idea of the overall usage of the design.
2) *Debugging*: Finding and fixing bugs in a design is often a complicated and time-consuming task. Automatically generated properties can assist in spotting faults more quickly, e.g., when the reported properties slightly deviate from expectations.
3) *Verification*: For certain designs, equivalence checking requires considerable effort. In these cases, checking simple automatically generated properties may still be possible due to property-specific abstraction mechanisms. The property-checks can then be used as a lightweight approach to increase the confidence in the correctness of the design or to pinpoint functional differences.

This paper summarizes a special session covering state-of-the-art research in Design Understanding with a focus on deriving human-understandable, logic properties from an unknown implementation in a concise and straight-forward way.

II. Syntax-Guided Property Enumeration

Automated assertion generation approaches [3], [17] for RTL derive properties of a design from simulation data. A formal verification engine guarantees that the assertions hold on the design. The derived assertions consequently depend on the quality and quantity of the simulation data used. Moreover, the generated assertions often capture cycle-accurate signal manipulations implemented in the RTL code, which tend to be long and not easy to understand for humans.

We propose syntax-guided property enumeration [13], a technique that derives temporal-logic formulæ of bounded length directly from a design. The idea is inspired by the recent success of syntax-guided synthesis (SyGuS) [1]. The SyGuS problem is, given a specification Γ in form of a logic formula with an uninterpreted symbol F and a context-free grammar G, to find a syntactic expression $\varphi \in G$ such that $\Gamma[F/\varphi]$ holds for all possible assignments to the free variables in Γ when F is replaced by φ.

Syntax-guided property enumeration [13] reformulates the SyGuS problem in the context of temporal logics and model checking. For a given hardware design S, a list $\varphi_1, \varphi_2, \ldots, \varphi_n$ of temporal-logic formulæ is enumerated by unwinding a grammar G. Each formula is model-checked on the design. Satisfied formulæ are kept and reported to a user, failing formulæ are used to prune the search space. A termination criterion, e.g., in form of a time limit or an upper bound on the maximum length of a formula, is required to guarantee termination. The

*This research was supported by H2020-ERC-2014-ADG 669354 CyberCare and by the German Research Foundation (DFG) under the project ASDPS (JA 2357/2-1).

978-1-5386-4756-1/18/$31.00 ©2018 IEEE

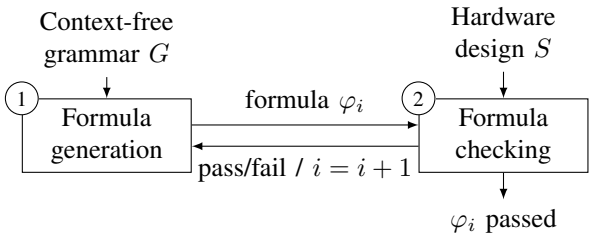

Fig. 1. High-level overview of syntax-guided property enumeration

$$
\begin{aligned}
S &::= \mathsf{AG}(F) \\
V &::= \mathit{signal} \\
F &::= \mathsf{false} \mid V \mid (\neg F) \mid (F \wedge F) \mid \\
&\quad\ \mathsf{AX}(F) \mid \mathsf{AF}(F) \mid \mathsf{AG}(F) \mid (F)\,\mathsf{EU}(F)
\end{aligned}
$$

Fig. 2. CTL grammar for invariant generation with start symbol S

overall formula generation process is visualized in Fig. 1 and consists of two steps: 1) in the formula generation step, a new temporal-logic formula φ_i is generated from the grammar. 2) In the formula checking step, the formula φ_i is model-checked on S. The verdict of model checking is reported to the user and provided to the formula generation step to enable learning from previous results.

We have implemented a proof-of-concept of syntax-guided property enumeration in C++ using the enumeration library behemoth[1] and the model checker IImc [9]. Our implementation takes as input a gate level circuit design in the AIGER format, a grammar that describes a fragment of *Computation-Tree Logic* (CTL), a bound on the maximal number of logic operators in a formula, and a time limit in seconds. The implementation enumerates all formulæ by length and thus guarantees that shortest and best understandable properties are generated—assuming that shorter formulæ are easier understood by humans.

In an experiment, we have generated assertions (or invariants) using our implementation with the gammar in Fig. 2, a limitation to at most 4 logic operators per formula, and an overall time limit of 15 minutes for generating invariants. As signals, we consider all primary outputs and latch outputs of a design. For evaluation, we use six benchmarks provided with the model checker IImc. All experiments have been conducted on a Linux workstation with an Intel Core i7-7820HQ, which supports up to 8 parallel threads, and 16 GB RAM.

Fig. 3 shows the number of generated invariants and the number of invariants successfully verified by the model checker.

In a second experiment, we have demonstrated that syntax-guided property enumeration, due to the independence of the individual model checking problems, qualifies for multi-threaded implementation. As shown in Fig. 4, it is possible to exploit the parallelism capabilities of modern processors to speed-up invariant generation.

III. Assertion Rewriting to Improve Readability

Assertions describe the behavior of a system. They can be used to verify the consistency between design intent, the actual implementation, and the specification [6]. This work

[1]behemoth, `https://github.com/hriener/behemoth`

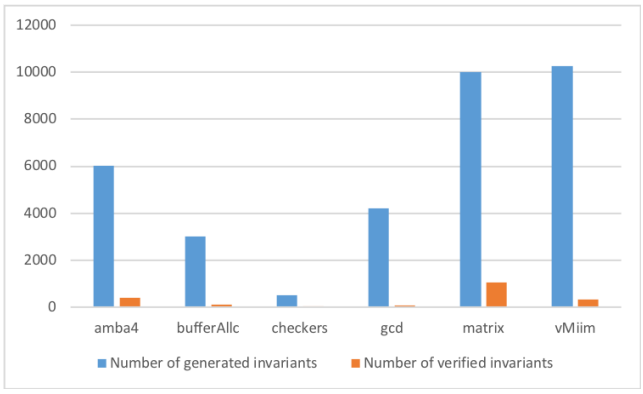

Fig. 3. Single-threaded invariant generation (time limit: 15 minutes)

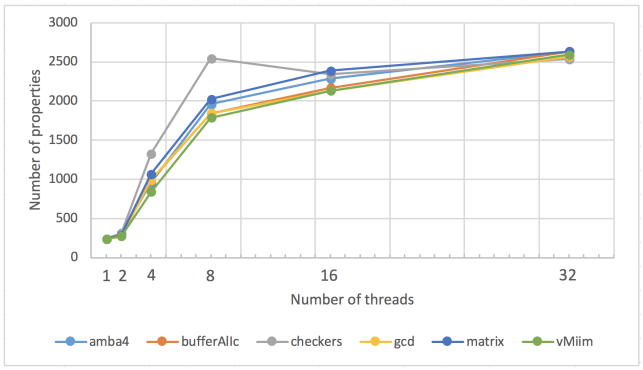

Fig. 4. Multi-threaded invariant generation (time limit: 90s)

proposes a methodology to estimate the quality of an assertion by analyzing its propositions and to rewrite the assertion to improve readability for humans. The quality estimation is based on a metric called Q evaluated with respect to a given test set of simulation traces. The metric Q is a linear combination of three metrics adapted from data mining: 1) The *support* refers to the number of occurrences of an assertion during simulation. 2) The *Correlation Coefficient* (CC) [16] refers to the dependency between antecedent and consequent of an assertion. 3) The *strength* [7] represents an assertion with a low occurrence but highly correlated to the other assertions which may cover corner cases.

The proposed methodology utilizes the three metrics to determine redundant and vague propositions and then generates a set of new assertions, which is more readable and shorter without loss of accuracy with respect to the original assertion when code coverage and fault coverage analyses are taken into account. Previous works have developed quality estimation techniques for assertions based on data-mining metrics; however, none of them have broken assertions down into their propositions to exercise them. For instance, in [10], an approach has been proposed that estimates the quality based only on the number of propositions. In [8], assertion quality is estimated based on their frequencies and correlation during simulation. The work does not consider assertions with a low number of frequencies, which may cover corner cases of a design.

Fig. 5 shows the flow of the proposed methodology. In Step 1, assertion A is broken down into its propositions. Consider the

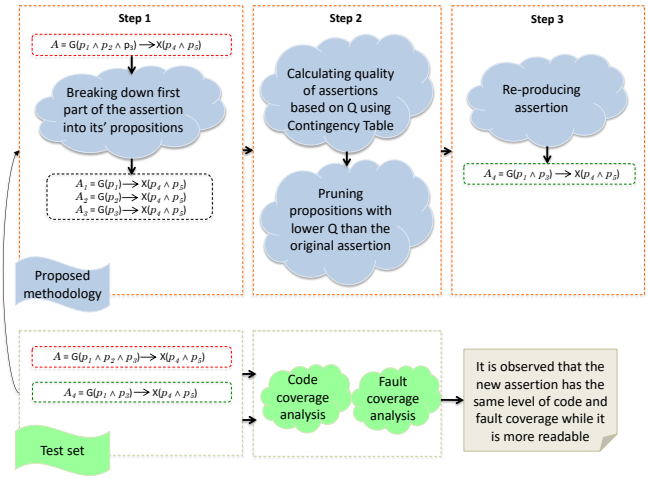

Fig. 5. Overview of the proposed methodology

assertion $A = \mathsf{G}(p_1 \wedge p_2 \wedge p_3) \rightarrow \mathsf{X}(p_4 \wedge p_5)$ in *Linear Temporal Logic* (LTL), which states it always happens that p_4 and p_5 are satisfied one simulation instant later than p_1, p_2 and p_3 become true. The propositions $p_1, p_2, \ldots, p_5$ are composed of invariants over the signals $v_1, v_2, \ldots$, such as $p_1 : (v_1 = \mathsf{true}) \wedge (v_2 > v_3)$, $p_2 : v_4 = \mathsf{false}$, $p_3 : (v_4 = \mathsf{false}) \vee (v_1 = \mathsf{true})$, etc. The assertion A is first broken down into three assertions A_1, A_2, and A_3 as shown in Fig. 5. In the next step, Step 2, the quality of assertion A and the three new assertions A_1, A_2, and A_3 are calculated with the help of the *Contingency Table* (CT). The CT represents the relation between antecedent φ and consequent ψ for each assertion A for form $A = (\varphi \rightarrow \psi)$ and counts the occurrences of φ and ψ.

Looking at (1), f_{11} represents the number of times where φ and ψ are true during simulation. f_{10} represents the number of times where φ is true but ψ is false. f_{01} is the dual of f_{10}, i.e., it is the number of times where φ is false and ψ is true during simulation. f_{00} is the number of times an assertion is not true through the simulation. f_{1X} is the sum of f_{11} and f_{10}, f_{0X} is the sum of f_{01} and f_{00} and so on. For more detail in calculating metric Q based on the CT please refer to [8] and [7]. The formula of Q is equal to:

$$Q(A) = \frac{f_{11}}{f_{XX}} + \frac{f_{11}f_{XX} - f_{1X}f_{X1}}{\sqrt{f_{1X}f_{0X}f_{X1}f_{X0}}} + \sqrt{\frac{f_{11}^2}{|f_{X1} - f_{X0}| \cdot |f_{1X} - f_{0X}|}} \tag{1}$$

At this step the quality of all the assertions are estimated. The assertions with the lower degree of quality than the original one are discarded. In Step 3 the antecedents of all the remained assertions are recomposed together to reproduce a more readable and shorter assertion with respect to the original one. We supported the effectiveness of the proposed methodology with code coverage and fault coverage analysis.

We broke 6 assertions from an LBDR design (one components of a router) down into 28 new assertions: 5 of these new assertions had a lower degree of Q with respect to the original assertion and were pruned. The final assertion set was reproduced and exercised with code and fault analysis. 331 faults were injected to the design. The original assertions and the final set detected the same number of faults i.e., 301, while 100% of the code were covered.

Experimental results show that the proposed methodology allows us to prune vague and useless propositions in assertions without loss of accuracy when fault and code coverage analyses are taken into account.

IV. Understanding Formal Specifications

Using formal languages to specify properties that a system should or does satisfy has undeniable benefits: a fixed, formal semantics of the statements, as well as (the possibility of) formal proofs that the statement indeed holds. However, when formal languages are used to describe the *intended* behavior of a design, say in formal verification or synthesis, it is often difficult to write specifications that accurately reflect the intent of the designer. We argue that this is because of a gap between the formal semantics and the perceived meaning by a human designer, which must therefore also be taken into account in design understanding.

For example, consider the linear temporal logic specification

$$(\mathsf{G}\,\mathsf{F}\,r \wedge \mathsf{G}\,(r \rightarrow \mathsf{X}\,(\neg r\,\mathsf{W}\,g))) \rightarrow \mathsf{G}\,(r \rightarrow \mathsf{X}\,g)\,,$$

where r stands for a request that is issued by the environment, and g for a grant action triggered by the system under design. Intuitively, the formula states that the environment should never completely stop sending requests, and that after every request it waits until it was granted before sending a new request. Under this assumption, the system should grant every request one step after it appeared. Now, which systems do satisfy this specification? On the one hand, every system that grants a request one step after it appeared. But on the other hand, this specification is also satisfied by a system that *never grants a request*, because this forces the environment to stop sending requests, which means that the first part of the assumption will be violated.

While an expert in temporal logics may detect such pitfalls, one must take into account that this is a very simple example. Actual specifications of a system, in particular if they are automatically generated by a process for design understanding, may be much more complex.

In the following we consider some common pitfalls in writing specifications for verification and synthesis, try to predict what this might mean for design understanding, and offer a few directions that may be worth investigating in the future.

The example above highlights a problem that is very common in formal synthesis (and to some extent also verification): virtually no system runs on its own, but instead we have to consider its behavior in communication with other systems, or as one of many components of the same system. To obtain scalable formal methods for analysis and design, we need to be able to decompose systems into their components, which in turn requires that the specification of any component also contains the necessary *assumptions* on other components.

However, writing good formal assumptions is highly non-trivial. Two reasons make it particularly challenging:

1) By the standard interpretation of temporal logics like LTL, assumptions are interpreted in a *worst-case* manner. That is, we can only assume exactly what is specified, and in all other cases we have to expect utmost hostility from the other components. On the one hand, this is the only sound way to interpret assumptions. On the other hand, such an interpretation is overly pessimistic if we consider a system of components that are really intended to co-operate. Further, the worst-case interpretation requires specifications to be overly specific, which makes it harder for the other components of the system to satisfy them, and harder for a human to understand and write correctly.
2) Again by the standard interpretation of LTL, a specification of the form $\varphi \rightarrow \psi$ is satisfied if the assumption φ does not hold. As we have seen above, such a specification can of course be satisfied by fulfilling the guarantee, but in some cases the system can also force the environment to violate the assumptions in order to satisfy the specification.

Assume-guarantee reasoning [11], [14] and vacuity-detection [12] provide frameworks to handle these problems in verification. In synthesis, Bloem et al. [2] have considered these problems, among others, when handling assumptions in synthesis. When using automated methods for automatically *generating a specification* for a given system, we may well run into the dual problem: while the specification states exactly what the system does, the human designer may interpret it in a wrong way because of somewhat counter-intuitive corner-cases in the semantics of temporal logics.

For the problems above in particular, a possible solution would be to highlight such potential corner-cases, or otherwise increase the awareness of their existence. In general however, while we cannot solve the problem completely, we advocate to minimize it by guiding the automatic generation of formal properties towards statements that are less likely to be misinterpreted.

One obvious way to make formal statements understandable is to keep them short or simple, informally spoken. Formally, this may mean different things. Let us again look at the dual problem of formal synthesis. There, one tries to obtain systems that are human-understandable in a number of different ways:

- by starting from predefined sketches [15],
- by supplying a predefined grammar [1],
- by bounding their size [5], or
- by bounding more complex metrics of the implementation, such as the number of cycles in the state machine [4].

All of these give rise to similar approaches that could be taken when generating a specification from a given system. These approaches are:

- using predefined templates of properties, for each of which the intuitive meaning is clear or can be supplied to the designer,
- using a fixed grammar that is limited in a way as to make the resulting formulas clear and easy to understand,
- bounding the size of formulas with respect to standard metrics such as length, number of different subformulas, nesting depth of temporal operators, etc., or
- bounding the formulas under more complex metrics that correlate with human understanding of the property. This may include a small representation by an accepting automaton, the decomposition of the overall property into several (largely) independent and easy to understand properties, or, vice versa, the generalization of several related properties into one property that subsumes them.

The proposed property mining approaches described in the previous sections already show the benefits of some of these considerations.

V. Conclusion

In this paper, we have provided an overview of state-of-the-art research in the field of Design Understanding focusing on techniques to derive readable properties in temporal logic from an unknown hardware design.

References

[1] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. IOS Press, 2015.

[2] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer. How to handle assumptions in synthesis. In *SYNT*, volume 157 of *EPTCS*, pages 34–50, 2014.

[3] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen. Inferno: Streamlining verification with inferred semantics. *TCAD*, 28(5):728–741, 2009.

[4] B. Finkbeiner and F. Klein. Bounded cycle synthesis. In *CAV*, volume 9779 of *LNCS*, pages 118–135, 2016.

[5] B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[6] H. Foster, A. Krolnik, and D. Lacey. *Assertion-based design (2. ed.)*. Kluwer, 2004.

[7] T. Ghasempouri, S. Payandeh Azad, B. Niazmand, and J. Raik. An automatic approach to evaluate assertions' quality based on data-mining metrics. In *ITC-Asia*, 2018.

[8] T. Ghasempouri and G. Pravadelli. On the estimation of assertion interestingness. In *VLSI-SoC*, pages 325–330, 2015.

[9] Z. Hassan, A. R. Bradley, and F. Somenzi. Incremental, inductive CTL model checking. In *CAV*, volume 7358 of *LNCS*, pages 532–547, 2012.

[10] S. Hertz, D. Sheridan, and S. Vasudevan. Mining hardware assertions with guidance from static analysis. *TCAD*, 32(6):952–965, 2013.

[11] C. B. Jones. Software development based on formal methods. In *WSFA*, pages 153–172, 1986.

[12] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.

[13] G. Martino, H. Riener, and G. Fey. Coverage-guided CTL property enumeration for understanding models of reactive systems. In *IWLS*, 2018.

[14] A. Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[15] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, volume 5904 of *LNCS*, pages 4–13, 2009.

[16] P.-N. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *KDD*, pages 32–41, 2002.

[17] S. Vasudevan, D. Sheridan, S. J. Patel, D. Tcheng, W. Tuohy, and D. R. Johnson. GoldMine: Automatic assertion generation using data mining and static analysis. In *DATE*, pages 626–629, 2010.