Multilevel Design Understanding: From Specification to Logic Invited Paper

Sandip Ray
NXP Semiconductors
Austin, TX 78735
USA
sandip.ray@nxp.com

Ian G. Harris
University of California Irvine
Donald Bren Hall, 3088
Irvine, CA, 92697
harris@ics.uci.edu

Goerschwin Fey
German Aerospace Center
University of Bremen
28359 Bremen
Germany
fey@informatik.uni-bremen.de

Mathias Soeken
EPFL
1015 Lausanne
Switzerland
mathias.soeken@epfl.ch

ABSTRACT

We present an outline of the field of Multilevel Design Understanding by first defining and motivating the related problems, and then describing the key issues which must be addressed in future research.

CCS Concepts

•Hardware → Electronic design automation; Hardware description languages and compilation;

Keywords

Design understanding, electronic design automation

1. INTRODUCTION

The design process is essentially a creative process which is reliant on the ability of designers to balance the interactions between a complex set of constraints to arrive at successful solutions. In order for designers to manage this task, they must collectively have a complete understanding of the behavior of the system, the mapping between behavior and structure, and the impact of each design feature on constraints such as power, performance, cost, and security. Design tasks require reasoning across multiple levels of abstraction in order to determine the impact of high-level design decisions, or to trace a design characteristic back to the feature which caused it. In a real design, cross-abstraction reasoning is difficult because the relationships between the different abstractions of a design are not captured. Designer time is expended discovering these cross-abstraction relationships in order to perform design, verification, and maintenance tasks. This paper summarizes a special session covering the state-of-the-art in Design Understanding, research in approaches to provide designers with the design information needed in a concise and straightforward way. The volume of design information is enormous, so a significant part of the problem is determining what subset of information is relevant to the designer to assist with the particular design problem currently at hand.

ICCAD '16 Austin, Texas USA

DOI: http://dx.doi.org/10.1145/2966986.2980093

2. THE COMPLEXITY OF SPECIFICATION

The life of a computing system arguably begins with architects developing architectural models, and tuning various design parameters for target performance and power consumption. The result of this process is the definition of the high-level architectural specification of the system, which defines the functional requirements for the system design. This specification is used as a guide for most downstream activities, *e.g.*, decompose system functionality, implement various design blocks, perform code reviews, design pre-silicon and post-silicon validation test-benches, define assertions, etc. Clearly, the specification documents ought to provide an obvious, authoritative source of design understanding that is already available as part of any industrial design flow.

Unfortunately, architectural specifications in practice do not live up to the promise of an "authoritative source" for design requirements. One reason is that they are rarely developed with the rigor, discipline, and formality warranted for something used so extensively throughout the system life cycle. System functionality is typically described informally with prose mixed with charts, diagrams, and tables. Furthermore, these descriptions are spread across a large number of different documents, each document several hundreds of pages long and covering different aspects of the design (*e.g.*, functionality, power, security, communication, etc.). Unsurprisingly, these documents contain inconsisten-

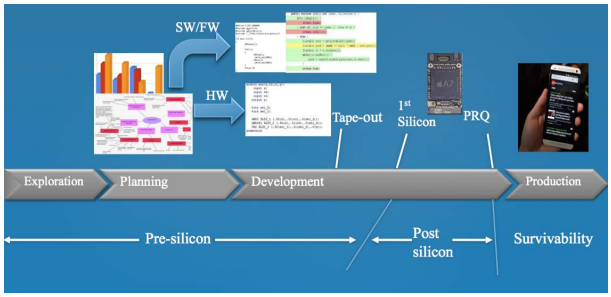

Figure 1: The key elements of a computing system life-cycle. Tape-out refers to the time when the design is mature enough to get to the first fabrication. PRQ or "Product Release Qualification" refers to the decision to initiate mass production of the product.

cies, ambiguities, even errors.

The lack of a cohesive, clear, unambiguous specification has been generally known as the Achilles heel of validation in particular; it is common to hear researchers in various forums on verification and validation to lament on the state of specification definition and single it out as a key factor contributing to validation complexity. Such accusations are not without substance: ambiguous, erroneous, or inconsistent specifications not only prolong validation, but in cases can lead validation planning to wrong directions or miss critical coverage requirements. Identification of such problems late in the implementation or validation phases can lead to late design churns, complex patches, point fixes, de-featuring, and in some cases cancellation of the product.

Given the critical effect of specification on system design on the one hand, and the sordid state of practice in specification definition on the other, a variety of formalisms have been proposed in recent years to provide ways of standardizing specifications. These range from formalisms based on temporal logic [1], a plethora of charts, diagrams, and message flows [11, 5, 28], as well as formats for specific design collaterals such as control register and interfaces [2]. Nevertheless, there has been no consensus in industry today on unified adoption of any formalism.

Why is it that we cannot impose a rigor and formalism into a topic so important as design specification, while knowing that informal, ambiguous treatments lead to serious consequences for the business? The short answer is that most of the formalisms are not compatible with the complex development flows of modern computing systems. In order to derive an effective approach to specification, it is therefore imperative to understand closely how the development occurs, and how the specifications are used at different points of this flow. To that end, this section gives a brief introduction to an industrial system development activity, focusing in particular on the specification aspect.

Fig. 1 gives a high-level overview of the system design activities along the life-cycle of a computing system development. A high-level specification document, also referred to as "HAS" (High-level Architectural Specification) is developed around the middle of the planning phase and is used as an input for architectural exploration, and is generally one of the first system-level design documents generated.

Ideally, the HAS provides a high-level specification of system functionality. So why is this insufficient? The trouble is that the HAS is developed at a point where very little of the architectural features has been concretized. Those features get defined through architectural explorations of the system, using different architectural models derived from the HAS. The architectural parameters include critical features such as cache sizes, pipeline depths, bus bandwidths, etc., and are embodied in a new set of documents called MAS (Micro-Architectural Specifications). Significantly, the exploration may result in refinement of the high-level requirements themselves, *e.g.*, exploration of different power-management capabilities may result in the conclusion that a certain power profile as specified by the HAS is unattainable under the other constraints for the product; the architectural requirements at that point must be modified to account for this discovery, and the consequent architectural exploration must account for the result. Overall, architectural exploration is a highly iterative process resulting in continuous refinement of the HAS; clearly, this already breaks the view of the HAS as created originally as the authoritative source of design requirement. One important factor to keep in mind is concurrency of design development. Implementation of key components typically start concurrently with architectural exploration, and are thus based on the original HAS. If architectural exploration results in a HAS change affecting a design component under active implementation, the refinement needs to be propagated directly to the implementation teams (and the HAS at this point may be obsolete).

For modern systems such as phones or tablets, several other factors add to the complexity induced by the HAS and MAS inter-dependencies described above. Most of these systems use the System-on-Chip (SoC) design methodology. The idea is to develop a system quickly by integrating predesigned hardware (or software) blocks, often referred to as an "Intellectual Property" of IP. Each IP, of course, comes with an architectural specification (which, by the above description, may not fully correspond to the implementation). The HAS for the SoC then defines the aggregation and composition of these IPs, and the MAS defines the system-level architectural parameters. Note that by this process, we already have a large number of specification documents in play, (*e.g.*, one per IP, the system-level HAS, and system-level MAS), each of which is concurrently developed with inconsistencies galore.

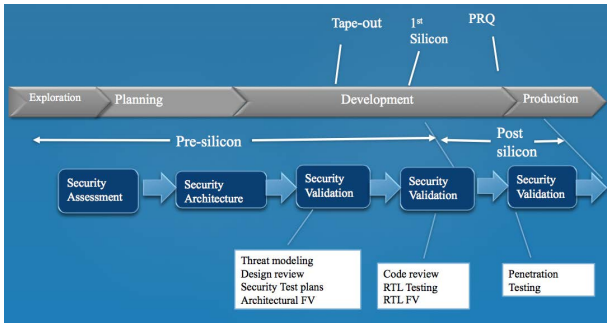

Figure 2: System Life Cycle from Security View.

Unfortunately, the above is only the tip of the iceberg. In addition to the above, in a modern system design, there are high-level flows defining requirements, architecture, and implementation of security, power management, validation,

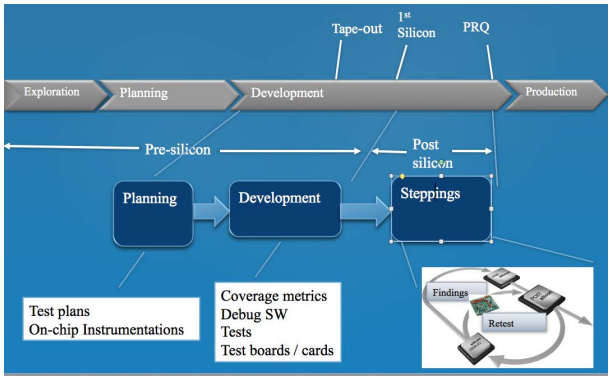

Figure 3: System Life-cycle from Post-silicon Validation View.

physical layout, etc. Each of these flows involve activities at various stages of system development. To illustrate the point, Fig. 2 shows the system life-cycle view focusing on activities related to security, and Fig. 3 focuses on the activities related to post-silicon validation. Each of these are critical activities, performed concurrently with functionality along the system design life-cycle, but by different teams each with a very different view of the life-cycle. Perhaps more pertinently for the topic of this section, each activity generates and consumes a large number of different documents, including access control and protection requirements, debug requirements, on-chip instrumentation architecture, etc. Each such document, in its turn, refers to the architectural definitions from different IP documents. To make matters worse, there are numerous trade-offs between the different flows that need to be addressed, *e.g.*, security and post-silicon debug involve significant conflicts and competitions [23].

Given the above, it should be clear that specification definition is a complex enterprise. To our knowledge there is no formalism or infrastructure that comprehends the conflicting needs from design specification by different architecture, design, and validation activities, and their subtle dependencies. On the other hand, this clearly is a very important area of research: as we move in an era where we are envisioning computing system applications of the scale of smart cities, highways, and multiplexes, it is imperative to be able to comprehend the design requirements at least to the point where we can say whether we have built what we intended to build; without such accountability the entire computing infrastructure can fall like a house of cards with disastrous consequences. The onus is thus on researchers in design understanding to develop an infrastructure for creating, maintaining, and comprehending specifications.

3. DESIGN AUTOMATION FROM NATURAL LANGUAGE SPECIFICATIONS

The task of interpreting natural language specifications has traditionally been exclusively manual because, generally speaking, only humans with expert design knowledge have the ability to properly interpret specification documents. However, a number of researchers in electronic design automation have attempted to automate the design and validation tasks using information contained in natural language documents. Design automation from natural language has the potential to provide many benefits including reduced time-to-market, reduced number of design errors, and early identification of incomplete and inconsistent specifications.

Attempts to perform synthesis from natural language can be found as early as the late 1980s as part of the first research in high-level synthesis. Researchers have generated partial designs from natural language specifications [9] by identifying a set of concepts expressed, together with a textual pattern for each concept. The approach taken in [4] defines a grammar to parse natural language expressions, and generates VHDL snippets. More recently, researchers have improved on the sophistication of the analysis by relying on the semi-formal structure of test scenarios described by acceptance tests [27]. A UML class diagram is generated based on the entities referred to in the scenario, and a UML sequence diagram is generated from the sequence of operations described. Researchers have performed text mining of hardware specifications to create a domain ontology describing relationships between components in the design [25]. Several previous research efforts have attempted to generate properties and assertions using different formal languages including ACTL[7], SystemVerilog [22, 10], and OCL [14]. However, each of these approaches imposes limitations on the generation process in order to make the problem tractable. Several techniques rely heavily on manual interaction to convert the original natural language into a form which is easier to process [22, 14, 7]. Some techniques only process a tightly constrained English subset [7].

3.1 Key Problems

Automation from natural language is a many-to-many mapping problem between the space of natural language descriptions and the space of formal behavioral descriptions. When performing any such mapping, two essential problems must be addressed.

- **Ambiguity:** A single natural language specification may map to multiple formal behaviors.
- **Linguistic Variation:** Many different natural language specifications may map to the same formal behavior.

Ambiguity is clearly bad for the design process and must be eliminated, or at least managed. Linguistic variation is positive from the perspective of the user because it allows him/her greater flexibility of expression. However linguistic variation increases the complexity of the synthesis process because more alternatives must be handled by the tool.

Ambiguity in natural language has been defined as the existence of multiple, alternative linguistic structures for a single document. In practice, an ambiguous document results in the existence of multiple, non-equivalent computational models. *Structural ambiguity* describes the condition when there are multiple valid parses for a given sentence. The following sentence shows an example of structural ambiguity, "The transmitter initializes the receiver in active mode". In this sentence, the prepositional phrase "in active mode" might refer to either the transmitter of the receiver. The two different assumptions would result in two different valid parses of the sentence, and different resulting designs. *Functional ambiguity* describes the condition where the behavior is not fully specified, so the computational model is incomplete. The following sentence is an example of functional ambiguity, "When an error is detected, the machine must

enter a valid state". The particular valid state which must be entered is not stated so many valid computational models can satisfy this constraint. Functional ambiguity may be either *intentional* or *unintentional*. Intentional functional ambiguity, sometimes referred to as *vagueness*, is inserted by the specification author in order to give more freedom to the designers when details are not essential. In the example above, the valid state entered by the machine may be a detail which is not essential to the function of the design.

Linguistic variation describes the aspect of language which enables a single concept to be expressed in multiple ways. For a design automation tool to be useful, it should accept a wide enough range of linguistic variation to allow reasonable freedom to the designer. Linguistic variation can be *morphological* resulting from the existence of synonyms in the language. For example, the verbs "set" and "assign" are often used interchangeably in design documents. This type of variation can be modeled in a straightforward way using a thesaurus to identify words with the same meaning. Each word associated with the same meaning can be considered equivalently and result in identical design/verification artifacts. Linguistic variation can also be *syntactic* where sentences with different structures have the same meaning. An example in the hardware domain would be the sentences, "Assign X to one" which is written in the active voice, and "Signal X is asserted" which is written in the passive voice.

4. AUTOMATIC FEATURE LOCALIZATION IN HARDWARE DESIGNS

To fulfill tight time-to-market constraints, more and more blocks from previous designs are reused or third party IP blocks are licensed. However, such blocks are often only poorly documented making adjustments to the blocks a difficult task. Moreover, design teams consist of many members resulting in frequent exchange of persons that need then to be trained. We consider techniques for automatic feature localization in hardware designs. These approaches support a developer in understanding a design by localizing parts of the code that implement a certain feature of interest.

4.1 Problem Description

Feature localization is understood as follows. A feature is some functionality of a circuit design that is triggered by some input stimuli and can be observed as an output response. This view on features explicitly does not account for non-functional aspects such as throughput or power-consumption.

The main goal is to understand the functionality of a given design. Thus, the problem is to identify the parts of the source code implementing a feature.

The assumption is that the designer is able to provide some (or better multiple) use cases to trigger the execution of a feature. For each use case, the feature(s) performed are known, yielding a mapping between use cases and features. The better the designer describes the parts of the use case needed to trigger the execution, the better will the localization be. Similarly, if the designer is able to specify where the execution of the functionality can be observed, the better the localization will be. This information is treated as additional input to improve the resolution of the algorithms.

4.2 Solutions

```
92|wire          invalid_0;
93|
94|wire add_enable_0 = (fpu_op_reg == 3'b000) & !(opa_reg[63] ^ opb_reg[63]);
95|wire add_enable_1 = (fpu_op_reg == 3'b001) & (opa_reg[63] ^ opb_reg[63]);
96|reg           add_enable;
97|wire sub_enable_0 = (fpu_op_reg == 3'b000) & (opa_reg[63] ^ opb_reg[63]);
98|wire sub_enable_1 = (fpu_op_reg == 3'b001) & !(opa_reg[63] ^ opb_reg[63]);
99|reg           sub_enable;
100|reg          mul_enable;
```

Figure 4: Highlighting Code

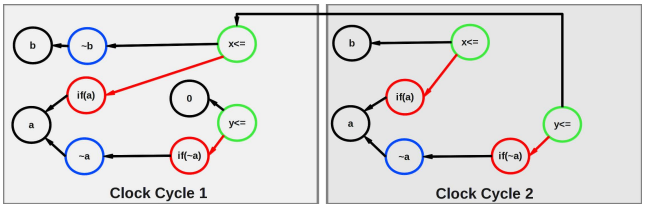

Figure 5: Dynamic Dependency Graph (DDG)

For software feature localization has been proposed early on already [6] using concept analysis to relate multiple features with source code. The underlying analysis when executing the source code has been performed by gathering coverage information for statement coverage.

This simple approach was extended to hardware where the application to *Register Transfer Level* description [17] as well as descriptions on *Electronic System Level* [20] is equally possible. As an advantage of using standard coverage metrics any simulation tool capable of gathering the relevant information can be applied to perform feature localization. However, concurrency that is inherent in hardware and many structural dependencies cannot easily be analyzed using these metrics. A first improvement over the previous approaches is to replace the earlier discrete grouping of code pieces into related and unrelated parts per feature by a continuous approach [17] proposed for debugging before [12]. This allows to reflect the likelihood of a piece of source code to be related to a feature by a given color, e.g., green denotes the code is related to the feature under consideration, red denotes the code is not related to the feature, in between the color continuously changes from green to red. Additionally, the hue expresses confidence, i.e., if a large amount of data supports the conclusion bright, colors are used; given only a few use cases as support, the code is shown in dark colors. Figure 4 illustrates this for a piece of source code.

By using mutation testing [24], the degree of automation is improved [16]. Mutation testing in this case decides whether a certain change in the design affects the feature under consideration. If this is the case, the modified code pieces are assumed to be related to the feature. Since mutation testing requires a sufficient amount of use cases, the generation of use cases and their annotation with related features has been automated. Still the strength of the analysis using mutation testing is limited as only coarse relations between changes and their effects are reflected without any insight into the design.

The use of *Dynamic Dependency Graphs* (DDGs) provides a much more powerful tool that still relies on dynamic analysis without the need for using static approaches [18]. A DDG can be created during simulation of a use case, yielding the relations between all code pieces executed as exemplary il-

lustrated in Figure 4.2. By using the additional information about relevant output and relevant pieces of input stimuli, the DDG can be reduced to pieces relevant for a feature. Using this information as coverage metric and relating it to source code allows for a much more precise analysis compared to the previous techniques [19]. In a case study, this approach was even more precise than the available documentation.

5. REVERSE ENGINEERING

The problem of reverse engineering (RE) is, given a low level netlist of gates, find a word-level netlist description which has the same behavior. It can be considered as generalized technology mapping problem. It can also be considered as a technique similar to feature localization but on a lower level of abstraction, i.e., gate level instead of register transfer level. An instance would be to start with an AIG (and-inverter graph, the subject graph) and a list of components (the library) which contains word-level component such as adders, multipliers, and shifters, of various bitwidths, and to find instances of these components in the subject graph. One can assume that the library is complete, so that every node in the subject graph can be covered. Thus the library might include also the gate types which make up the subject graph. In the classical technology mapping problem, there are two classes of methods employed, structural matching and Boolean matching. Boolean matching is the more powerful because it suffers less from structural bias. In recent years, Boolean matching has become the dominant choice because very efficient techniques have been developed for this. For example, cut-based mapping, NPN classification, pre-computation, table look-up, choice nodes, etc. are used in an advanced Boolean matching based method (see e.g., [21]). Also, typically there are many objective functions to be optimized; area, delay, power, CNF clause count, and wire count. Mapping is usually based on combinations of these. Iteration of mapping with changing optimizing criteria is also used. In RE there is a single objective: to find as many of the high level components in the subject graph as possible. RE is of interest for a number of reasons:

1. RTL may not be available; the design may be heritage and there was no initial RTL, it may be an AIG is being passed between various tools with no accompanying RTL, it may have come from having bit-blasted an RTL and synthesized at the bit level,
2. Creating a word-level description of a design to enable verification or synthesis.
3. Analyzing a competitor's chip
4. Detecting Trojan hardware

In general, RE is very difficult, but there are circumstances where the problem is made simpler. According to [15] the two main steps to solve a reverse engineering problem are: (1) block identification and (2) matching blocks against components in a library. The first step is considered the more complex part of reverse engineering and no satisfactory solutions have been found so far. This implies that we are faced with a variety of possibilities in which blocks can occur and in how they are related to the component. Hence, the second step can be regarded as generalized equivalence checking.

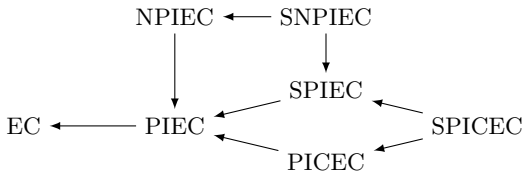

Figure 6: Equivalence checking problems

5.1 Equivalence Checking Problems

We consider different generalizations of equivalence checking to perform block matching of a block represented by a function f to a block represented by a function g. For example, if the number of inputs and outputs of the block equals the number of inputs and outputs of the component and if also the order is known, then matching the component to the block can be done with combinational equivalence checking (EC). EC is the problem of deciding whether two Boolean functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are equal. The problem can likewise be considered for functions that have multiple outputs. It has high significance in formal verification and is successfully applied to prove circuit equivalence in industrial environments (e.g., [13]).

However, the assumption that the order of inputs and outputs is known is very strict and quite unlikely in practical reverse engineering problems as it requires a high precision from the block identification algorithm. If the order is unknown, other variants of EC can be used to perform block matching. Permutation-independent equivalence checking (PIEC) asks whether two functions are equivalent under a specific permutation of primary inputs and primary outputs. Negation-and-permutation-independent equivalence checking (NPIEC) generalizes the problem by additionally allowing negation of primary inputs and primary outputs. Besides in reverse engineering and formal verification, these problems also play a central role in Boolean matching and technology mapping (e.g., [3]).

Further generalizations were recently proposed particularly for the application in reverse engineering. In the permutation-independent conditional equivalence checking (PICEC, [8]) problem, the block f has additional control inputs, which are known in advance. The problem is to find an assignment to the control inputs such that the resulting function is equivalent to g under a permutation of input and output variables. In the subset permutation-independent equivalence checking (SPIEC, [26]) problem, the block f can have more input and output variables than g. The problem is to find injective functions that map input and output variables of g to variables of f such that g is equivalent to the resulting subfunctions.

Fig. 6 summarizes the discussed equivalence checking problems. An arrow from problem A to problem B means that A generalizes B. As an example PIEC generalizes EC, since EC is a specialized version of PIEC in which the input and output permutation is known.

In the context of reverse engineering, algorithms have been presented based on constraint satisfaction programming to solve SPIEC (see, e.g., [26]) or based on Satisfiability Modulo Theories (SMT) (see, e.g., [8]). In the short run good algorithms for block identification are required to tackle reverse engineering in a larger scale. However, solving block matching for a variety of different scenarios and in an ef-

ficient manner is the backbone of reverse engineering flows and therefore also requires special attention.

6. CONCLUSIONS

In this paper we have provided an overview of some of the main problems associated with the field of Multilevel Design Understanding. As growing design complexity continues to be a burden on the efficiency of the design process, we expect to see increasing interest in these research questions.

Acknowledgments.

We wish to thank all our colleagues and collaborators who were involved in carrying the research discussed in the paper. This research is supported partly by the German Academic Exchange Service (DAAD) in the PPP 57134066, by the European Research Council (ERC) in H2020-ERC-2014-ADG 669354 CyberCare, by the European Comission (EC) in H2020-RIA-2015 644905 (IMMORTAL), ...

7. REFERENCES

[1] 1850-2005 - IEEE Standard for Property Specification Language (PSL), 2005.

[2] 1685-2014 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, 2014.

[3] Benini, L., and Micheli, G. D. A survey of boolean matching techniques for library binding. *ACM Trans. Design Autom. Electr. Syst. 2*, 3 (1997), 193–226.

[4] Cyre, W. R., Armstrong, J., Manek-Honcharik, M., and Honcharik, A. J. Generating VHDL models from natural language descriptions. In *Proceedings of the conference on European design automation* (1994), EURO-DAC '94.

[5] Damm, W., and Harel, D. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design 19*, 1 (2001), 45–80.

[6] Eisenbarth, T., Koschke, R., and Simon, D. Locating features in source code. *IEEE Transactions on Software Engineering 29* (2003), 210–224.

[7] Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., and Moreschini, P. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design 4*, 3 (1994).

[8] Gascón, A., Subramanyan, P., Dutertre, B., Tiwari, A., Jovanovic, D., and Malik, S. Template-based circuit understanding. In *Formal Methods in Computer-Aided Design* (2014), pp. 83–90.

[9] Granacki, J. J., and Parker, A. C. PHRAN-SPAN: a natural language interface for system specifications. In *Proceedings of the 24th ACM/IEEE Design Automation Conference* (1987).

[10] Harris, C. B., and Harris, I. G. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2016).

[11] ITU-TS Recommendation Z.120. Message Sequence Chart (MSC) - Annex B: Algebraic Semantics of Message Sequence Charts, ITU-TS, Geneva, 1995.

[12] Jones, J. A., Harrold, M. J., and Stasko, J. T. Visualization for fault localization. In *Proceedings of ICSE Workshop on Software Visualization* (2001), pp. 71–75.

[13] Kaiss, D., Skaba, M., Hanna, Z., and Khasidashvili, Z. Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In *Formal Methods in Computer-Aided Design* (2007), pp. 20–26.

[14] Keszocze, O., Soeken, M., Kuksa, E., and Drechsler, R. Lips: An ide for model driven engineering based on natural language processing. In *Natural Language Analysis in Software Engineering, 1st International Workshop on* (2013).

[15] Li, W., Wasson, Z., and Seshia, S. A. Reverse engineering circuits using behavioral pattern mining. In *Int'l Symp. on Hardware-Oriented Security and Trust* (2012), pp. 83–88.

[16] Malburg, J., Encrenaz-Tiphene, E., and Fey, G. Mutation based feature localization. In *International Workshop on Microprocessor Test and Verification (MTV)* (2014), pp. 49–54.

[17] Malburg, J., Finder, A., and Fey, G. Automated feature localization for hardware designs using coverage metrics. In *Design Automation Conference (DAC)* (2012), pp. 941–946.

[18] Malburg, J., Finder, A., and Fey, G. Tuning dynamic data flow analysis to support design understanding. In *Design, Automation and Test in Europe (DATE)* (2013), pp. 1179–1184.

[19] Malburg, J., Finder, A., and Fey, G. A simulation based approach for automated feature localization. *IEEE Transactions on Computer Aided Design of Circuits and Systems (TCAD) 33*, 12 (2014), 1886–1899.

[20] Michael, M., Grosse, D., and Drechsler, R. Localizing features of ESL models for design understanding. In *Forum on Specification and Design Languages* (2012), pp. 120–125.

[21] Mishchenko, A., Chatterjee, S., and Brayton, R. K. Improvements to technology mapping for lut-based fpgas. *IEEE Trans. on CAD of Integrated Circuits and Systems 26*, 2 (2007), 240–253.

[22] Mueller, W., Bol, A., Krupp, A., and Lundkvist, O. Generation of executable testbenches from natural language requirement specifications for embedded real-time systems. In *Distributed, Parallel and Biologically Inspired Systems*, M. Hinchey, B. Kleinjohann, L. Kleinjohann, P. A. Lindsay, F. J. Rammig, J. Timmis, and M. Wolf, Eds., vol. 329 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2010.

[23] Ray, S., Yang, J., Basak, A., and Bhunia, S. Correctness and Security at Odds: Post-silicon Validation of Modern SoC Designs. In *Design Automation Conference* (2015).

[24] Serrestou, Y., and Robach, V. B. C. Functional verification of rtl designs driven by mutation testing metrics. In *Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)* (Aug 2007), pp. 222–227.

[25] Shankar, A., Singh, B., Wolff, F., and Papachristou, C. Ontology-guided conceptual analysis of design specifications. In *2014 51st Design Automation Conference* (2014).

[26] Soeken, M., Sterin, B., Drechsler, R., and Brayton, R. Simulation graphs for reverse engineering. In *Formal Methods in Computer-Aided Design* (2015).

[27] Soeken, M., Wille, R., and Drechsler, R. Assisted behavior driven development using natural language processing. In *TOOLS (50)* (2012).

[28] Talupur, M., Ray, S., and Erickson, J. Transaction Flows and Executable Models: Formalization and Analysis of Message passing Protocols. In *FMCAD 2015* (2015), pp. 168–175.