Research Statement

Sudipta Kundu

Programming errors or bugs are omnipresent in critical areas of Computer Science ranging from device drivers to real-time systems to hardware designs, and have lead to economic loss as well as loss of human lives. The Pentium FDIV bug, ARIANE failure, or Therac-25 accident are among several of the horror stories caused by software or hardware bugs. My research focuses on developing techniques that improves the reliability of programs written in various levels of design abstraction for both hardware and software.

Purely from a cost point of view, ensuring reliability is increasingly challenging in the domain of hardware designs, due to the growing size, heterogeneity, and cost of redesign of System on Chip (SOC). Furthermore, the reliability issues in turn makes the design process from initial specification to chip fabrication increasingly complex. However, the growing complexity provides incentive for designers to shift toward using high-level languages such as C, SystemC, and SystemVerilog to do system-level design. While a major goal of these high-level languages is to enable verification at a higher level of abstraction, allowing early exploration of system-level designs, the focus so far has been on traditional testing techniques such as random testing and scenario-based testing.

Recently, approaches inspired by *formal methods* have emerged as an alternative way to ensure the correctness of high-level designs, overcoming several of the limitations found in traditional testing techniques. Moreover, advances in the area of SAT solvers, automated theorem provers, and model checking techniques have allowed researchers to formally verify many properties of real systems. For example, modern approaches can frequently check for the absence of deadlocks and assertion violations in large designs, thereby increasing the overall reliability.

My dissertation focuses on *high-level verification of system designs.* In this work, I envision a design methodology that relies upon advances in synthesis techniques as well as on incremental refinement of design process. These refinements can be done manually or through elaboration tools. My work addresses verification of specific properties in high-level languages as well as checking that the refined implementations are equivalent to their high-level specifications. In this direction, I have worked on various techniques that improve the current state of the art. The novelty of each of these techniques is that they use a combination of formal techniques to do *scalable* verification of system designs completely *automatically.*

More interestingly, the approaches in my thesis show that practical and useful tools can be built by intelligently combining various simple fundamental techniques (like divide and conquer, concurrent program analysis, compositional design, relational approach, theorem proving, model checking, static analysis and dynamic analysis) from different areas of Computer Science. Furthermore, these approaches are not limited to only high-level verification, and can be adapted to a wide variety of areas, such as program analysis, software engineering, compiler design, design automation, and security.

High-Level Verification of System Designs

My work falls into two categories: (a) methods for verifying properties of high-level designs and (b) methods for verifying that the translation from high-level design to low-level Register Transfer Language (RTL) preserves semantics. Taken together, these two parts guarantee that properties verified in the high-level design are preserved through the translation to low-level RTL. By performing verification on the high-level design, where verification is easier to perform, and then checking that all refinement steps are correct, we expand hardware development methodology to provide strong and expressive guarantees that are difficult to achieve by directly analyzing the low-level RTL code.

Property Verification of High-Level Designs. Starting with a high-level design, we use *model checking* techniques to verify that the design satisfies a given property such as absence of deadlock or assertion violation. Model checking in its pure form suffers from the well-known *state explosion* problem. To cope with the state explosion, some systems give up completeness of the search and focus on the bug finding capabilities of model checking. This line of thought lead to execution-based model checking approach, which for a given test input and depth, systematically explores all possible behaviors of the design (due to asynchronous concurrency). The most striking benefit of execution-based model checking approach is that it can analyze feature-rich programming languages like C++, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this approach is the idea of *stateless* search, meaning it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. To address this, one can use dynamic *partial-order-reduction* (POR) techniques to avoid generation of two paths that have the same effect on the design's behavior. Intuitively, POR techniques exploit the independence between parallel threads to search a reduced set of paths and still remain provably sufficient for detecting deadlocks and assertion violations.

We implemented `Satya` [6], a novel *query-based* model checking framework that combines static and dynamic POR techniques along with high-level semantics of SystemC to intelligently explore all possible behaviors of a SystemC design. We reduce the runtime overhead by computing the dependency information statically and using it during runtime, without significant loss of precision. In our experiments `Satya` was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), which may not have been found by simulation.

Another approach for model checking is to use symbolic algorithms that manipulate sets of states instead of individual states. These algorithms avoid ever building the graph for the system; instead, they represent the graph implicitly using a formula in propositional logic. *Bounded Model Checking* (BMC) is one such algorithm that unrolls the control flow graph (loop) for a fixed number of steps (say k) and checks whether a property violation can occur in k or fewer steps. This typically involves encoding the bounded model as an instance of Satisfiability (SAT) problem. This problem is then solved using a SAT or SMT (Satisfiability Modulo Theory) solver. A key challenge for BMC is to generate efficient verification conditions that can be easily solved using the appropriate solver.

We developed a new symbolic method [2], which combines POR with an asynchronous modeling approach that generate verification conditions directly without an explicit scheduler. We introduce the notion of *Mutually Atomic Transactions* (MAT): two transactions are mutually atomic when there exists exactly one conflicting shared-access pair between them. Previous approaches add interleaving constraints between all pairwise global accesses, thereby allowing redundant interleavings. We reduce the verification conditions by allowing pairwise interleaving constraints *only* between MATs. Our experimental results show that our approach improves the current state of the art both in performance and in size of the verification condition [2].

Verifying the synthesis from high-level design to low-level RTL. Once the important properties of the high-level components have been verified, the translation from the high-level design to low-level RTL still needs to be proven correct, thereby also guaranteeing that the important properties of the components are preserved. High-Level Synthesis (HLS) is the process that transforms a high-level design, usually expressed in languages like C, C++, or Java, to a low-level RTL design. One approach to prove that the translation from high-level design to low-level RTL is correct is to show – for each translation that the HLS tool performs – the output program produced by the tool has the same behavior as the original program. This technique is called *Translation validation.* Although this approach does not guarantee that the HLS tool is bug free, it does guarantee that any errors in translation will be caught when the tool runs, preventing such errors from propagating any further down the hardware fabrication process.

We developed a translation validation algorithm [1, 7, 5] that uses a *bisimulation relation* approach to automatically prove the equivalence between two concurrent systems. We implemented our algorithm in a system called `ARCCoS` and used it to validate the synthesis process of `Spark`, a parallelizing HLS framework. `ARCCoS` validates all the phases (except for parsing, binding and code generation) of `Spark` against the initial behavioral description. Furthermore, our experiments showed that with only a fraction of the development cost of `Spark`, our algorithm can validate the translations performed by `Spark`, and it even uncovered two previously unknown bugs that eluded testing and long-term use.

Another approach to guarantee the translation from high-level design to low-level RTL is correct, is by proving the HLS tool itself correct. Unlike translation validation, this approach proves the correctness of an HLS tool *once and for all*, before it is ever run. Because some of the most error prone parts of an HLS tool are its optimizations, we developed a technique that proves the correctness of optimizations using *Parametrized Equivalence Checking* (`PEC`) [4]. Furthermore, our approach is not limited to only HLS tools; it can be used for any domain that transforms an input program using semantics-preserving optimizations, such as optimizers, compilers, and assemblers.

The `PEC` technique is a generalization of translation validation that proves the equivalence of *parameterized programs*. A parameterized program is a partially specified program that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not modify certain variables. To highlight the power of `PEC`, we designed a language for implementing complex optimizations using many-to-many rewrite rules, and used this language to implement a variety of optimizations including software pipelining, loop unrolling, loop unswitching, loop interchange, and loop fusion. Using our `PEC` implementation, we were able to automatically verify that all the optimizations we implemented in our language preserve program behavior.

Moving Forward

In the future almost every aspect of our day-to-day activities will involve devices that contains interacting hardware and software components as well as interfaces to the physical world. As these components get more and more complex, their design will become increasingly difficult. For example, a moderately complex cyber-physical system may in the end require generating software code in C and Java, network code in Ruby, hardware code in VHDL, glue code in Python, and a web interface in JavaScript. My broader research goal is to build next-generation design tools (like synthesis, analysis, and compilation tools) for such complex systems. Within this domain, in the near term I would like to broadly work in the following areas.

Automatic Reliable Design Tools. A major issue with most design tools is that these tools are hard and inconvenient to use, and even harder to modify or extend. Thus, each time a new methodology is proposed a new tool has to be written, often from scratch. Furthermore, their reliability has always been a concern for both the tool developer and user. Unfortunately, building reliable design tools is difficult, error-prone, and requires significant manual effort. Indeed, it takes a long time to develop a mature design tool that is stable enough for broad adoption (often up to a decade), which in turn hinders the development of new designs and increases their time-to-market. Building upon my experience on implementation and verification of various design tools such as HLS and compilers, I would like to develop a modular and reusable framework that can be used to quickly prototype new ideas in a reliable manner and can also be extended easily by users.

Automatic Concurrent Program Analysis Tools. Another interesting challenge for design tools is to exploit and also help the designs to exploit the massive amount of concurrency offered by the hardware. One of the major problems in this area stems from the presence of multiple threads of control, which can lead to subtle and often unanticipated interactions between components. For instance, issues such as interference, race conditions, deadlock, and livelock are particularly important in this domain. Furthermore, the fact that many concurrent systems, such as hardware designs, operating systems and databases, are reactive, adds to the complexities in this area. Although, many commercial concurrent program analysis tools have become available, their adoption is in the early stages and the tools are often limited in the quality of the results and the kinds of correctness guarantees they can provide. Based on my experience with various reduction techniques like partial-order reduction, symmetry reduction and transactions, I would like to build an automatic and scalable analysis framework that can accurately account for the various complexities of concurrency. Here again my main focus will be to develop an extensible and reusable framework, which will allow user-defined properties and domain-specific reductions to be easily incorporated.

My research will allow us to build reliable and extensible frameworks that analyze and design complex hardware-software systems, thereby bridging the gap between software engineering and design automation. The challenges are very exciting, and I feel pursuing a career in this area will fulfill my ambition.

References

[1] **Sudipta Kundu** and Sorin Lerner. Translation Validation in High-Level Synthesis. *In Submission*, 2009.

[2] Malay Ganai and **Sudipta Kundu**. Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In ***SPIN '09**: Proceedings of the 16th International SPIN Workshop on Model Checking of Software*, 2009. *To Appear.*

[3] **Sudipta Kundu**, Sorin Lerner, and Rajesh Gupta. High-Level Verification. ***IPSJ Transactions** on System LSI Design Methodology*, 2009. (Invited Paper). *To Appear.*

[4] **Sudipta Kundu**, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct using Parameterized Program Equivalence. In ***PLDI '09**: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009. *To Appear.*

[5] **Sudipta Kundu**, Sorin Lerner, and Rajesh Gupta. Validating High-Level Synthesis. In ***CAV '08**: Proceedings of the 20th international conference on Computer Aided Verification*, pages 459–472, Princeton, NJ, USA, 2008. Springer.

[6] **Sudipta Kundu**, Malay Ganai, and Rajesh Gupta. Partial Order Reduction for Scalable Testing of SystemC TLM Designs. In ***DAC '08**: Proceedings of the 45th annual conference on Design Automation*, pages 936–941, New York, NY, USA, 2008. ACM.

[7] **Sudipta Kundu**, Sorin Lerner, and Rajesh Gupta. Automated Refinement Checking of Concurrent Systems. In ***ICCAD '07**: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 318–325, Piscataway, NJ, USA, 2007. IEEE Press.

[8] Gurashis Singh Brar, **Sudipta Kundu**, Pratik Worah, Susmit Biswas, Arijit Mukherjee, and Anupam Basu. OaSis: An Application Specific Operating System for an Embedded Environment. In ***VLSI Design '04**: 17th International Conference on VLSI Design*, pages 776–779, Mumbai, India, 2004. IEEE Press.

[9] Frederic Doucet, **Sudipta Kundu**, Ingolf H. Krüger, R.K. Shyamasundar, and Rajesh Gupta. Compositional Design Methodology for Scalable Verification of System Design. *In Preparation*, 2009.