Research Statement
Towards declarative programming for software systems

Haoxian Chen
University of Pennsylvania

The wide adoption of software systems in production and business has transformed the landscape of software development. Software programs are having higher stakes, and therefore require more rigorous guarantee on their correctness and performance. Traditional software development paradigm where the programs are implemented first, and then tested or verified afterwards, is hard to catch up with the increasing program complexity, scale, and the level of correctness assurance.

I believe a verification-centric approach will play a more important role in future software development, where developers start with the correctness specification, and then incrementally implement the software that satisfy the specification. My research aims to realize this vision by designing declarative and executable specification languages, with support for automatic code generation and verification. For application domains where a large amount of legacy programs exist and need to be refactored into new languages, I also develop program synthesis techniques to improve the refactoring efficiency.

Research Overview

Having the right level of abstraction is essential in software verification. Approaches that directly verify the software implementation, e.g., the source code, byte-code, etc., are often limited in scalability, and thus have to make different trade-offs between soundness and efficiency. On the other hand, approaches based on abstract formal models of the programs can reason about high-level properties more efficiently, but may be more difficult to adapt in practice due to the steep learning curve of formal specification languages, and the extra effort to specify an abstract yet accurate model.

My research aims to improve the verification process by designing high-level and declarative domain-specific languages, such that formal verification can be performed efficiently, and the implementation can be generated automatically from the high-level languages. In particular, domain-specific languages should be able to naturally capture the application logic, so that the specification can be concise, and declarative, where the programmers should only focus on the application logic, not the implementation, so that program implementation and optimization can be automated.

I also work on synthesizing high-level specifications from legacy programs. Formal specification languages, despite their promising benefits, haven’t gained mainstream adoption in practice, partly due to their steep learning curve. Today, it remains challenging for regular programmers to write a formal specification of their software systems. It is even harder to make sure that the written specification accurately capture every aspect of the underlying implementation. To address this challenge, I design tools that automatically generates declarative specifications from input-output examples, or execution traces of the legacy programs, thus lowering the bar for adopting formal specification in practice.

Declarative domain-specific languages

Declarative smart contracts. Smart contracts are programs that automatically enforce transactions on the blockchains. They have gained in popularity in recent years, managing digital assets that worth
billions of US dollars. Due to their high stakes, it is important to ensure that these smart contracts are implemented correctly. Today, verifying smart contracts remains a challenging task, and existing work mostly focus on low-level and generic properties like reentrancy attacks (the sending function being called multiple times in one transaction), or integer overflow. For contract-level properties that should hold across an infinite sequence of transactions, e.g., account balances always add up to the total supply of tokens, they typically have to trade soundness for efficiency, for example, verify within a bounded number of transactions.

To address this challenge, we propose DeCon [6], a declarative domain-specific language for smart contracts. DeCon serves as a high-level specification language for efficient analysis and verification, and enables automatic code generation. The design of DeCon is based on the observation that smart contracts are similar to databases, and that smart contract operations can be naturally mapped to relational constraints. Evaluation shows that representative smart contracts can be expressed in DeCon, and the implementation generated by DeCon only has moderate overhead over the reference implementation, with median at 14%.

Safety verification of declarative smart contracts. DCV [4] is a safety verification tool for smart contracts written in DeCon. It is sound and automatic, requiring no extra user annotations other than the property specification. It improves upon existing smart contract verification tools by exploiting the high-level abstraction of the DeCon language.

To verify a DeCon smart contract, it first extracts a high-level model from the DeCon contract, in the form of a state transition system. Next, it proves the safety properties using mathematical induction. One of the main challenges in mathematical induction is to infer an inductive invariant. In the context of DCV, a smart contract is interpreted as a reactive system that takes an infinite sequence of transactions as input. In order to prove that such a system satisfies a target property, a verifier needs to infer an invariant such that it holds inductively across all possible transactions, and it can imply the target property. Automatically inferring such invariants have been a long standing challenges in formal verification, and still remains an active research problem. Many safety verification tools have to rely on the users to annotate such inductive invariants.

DCV can automatically infer such invariants. The insight of DCV is that, in the context of DeCon smart contract verification, inductive invariants can be composed from logical predicates in the DeCon programs, following simple templates. Since DeCon is declarative, meaning that only the transaction logic is specified but not the implementation, the number of extracted predicates are relatively small, making the inference problem tractable. Evaluation shows that DCV outperforms state-of-the-art tools in terms of verification efficiency and coverage on the benchmark verification tasks, demonstrating the benefits of high-level and declarative domain-specific languages.

Program synthesis

Example-based specification synthesis. NetSpec [5, 7] is a program synthesis tool that generates declarative specification of network protocols from input-output examples. It formulates the program synthesis task as an optimization problem, where the search space is the declarative specification program space, and the optimization objective is a metric that measures how close a candidate program’s output resembles the reference output examples. A solution is a program that generates exactly the reference output examples. It uses a novel stochastic search algorithm to efficiently uncover the solution.

NetSpec is also robust to example quality. Given a set of input-output examples, it identifies ambiguities by running differential testing among the generated programs that satisfy these examples. If ambiguity is detected (differentiating input is found), it asks the user to annotate the expected output for this differentiating input instance, and then run the synthesis and disambiguation process again. This process iterates until there is no ambiguity can be found.
Evaluation shows that NetSpec outperforms state-of-the-art synthesis tools in terms of expressiveness (number of supported benchmark specifications) and synthesis efficiency. It is robust to incomplete examples, being able to synthesize the desired specifications even when input-output examples are incomplete. It can also synthesize specification from large scale execution traces containing up to 4,500 communication messages.

Other Research

AutoML for networking. AutoML systems take data as input, perform model selection and hyperparameter tuning, and return a trained model as output. However, the output of AutoML systems is highly dependent on the data quality: they yield sub-optimal models when given incomplete data. Therefore, we incorporate a feedback mechanism into existing AutoML systems, and customize it for networking tasks (selecting the best congestion control protocol under different network environments). Using the networking context, it returns interpretable feedback on how to improve data quality [1].

Cache replacement policies. Cache hit rate is essential for system performance. However, popular key-value store services still use LRU (least recently used) as the cache replacement policy, whose performance suffers from pathological access patterns. We propose a novel cache replacement policy that can adopt itself to the access pattern for key-value store systems, and implement a prototype that outperforms state-of-the-art policies in cache hit rate and system throughput [2].

In addition, I worked on network measurements [3], where we propose an end-host based measurement system with a centralized query interface. I have also worked on network switch hardware architecture for stateful software-defined network (SDN) applications [8].

Future Directions

The main theme of my research is to design declarative and executable domain-specific languages to automate the verification and implementation process in software system development. This approach requires deep understanding of the target application domain, and the ability to innovate existing program verification and optimization techniques to solve new problems, as demonstrated in my research. It will benefit large and complex systems where verification is limited by the scalability of traditional approaches. In the future, I will continue to develop this approach and expand it to other application domains.

Program optimization based on declarative specification. Optimizing program performance requires a lot of expertise and effort. It typically takes a lot of profiling to locate the performance bottlenecks, and careful engineering to mitigate these issues. And it is often ad hoc and repetitive: when a system migrates to different platforms, the original assumptions about the underlying systems may not hold and the optimization process has to start over. High-level declarative specification can help automate this optimization process. As we have seen in the success of database query optimization, where the automatically generated execution plan can match or even outperform manually optimized programs, I believe the same principle should apply to many different domains. In the near future, I plan to explore how to take advantage of the declarative specifications to automatically generate efficient implementation. Such optimization includes determining the right data structure and algorithm, making caching decisions, and estimating workload patterns, etc.

Smart contract verification beyond safety. As discussed in my prior research, smart contracts are high stake programs that require rigorous correctness guarantee. However, prior work is limited to safety verification. Liveness properties, on the other hand, are also essential to smart contracts. For example, many token contracts requires liquidity. In other words, owners can always withdraw or transfer their fund. Prior work have proved liveness properties in both interactive and automatic fashions. But those
approaches are limited in scalability. Declarative specification of the smart contracts, as demonstrated in prior research on safety verification, can improve verification efficiency by utilizing the higher-level abstractions of smart contracts. It is an exciting direction to further explore its benefits in liveness verification and many other verification tasks.

It is also interesting to prove the end-to-end soundness of these model-based verification approaches. In particular, the translation correctness, that is, the consistency between declarative specification and the Solidity or EVM implementation, plays an important role in proving end-to-end correctness. In addition, proving the compiler is free from known Solidity vulnerabilities, e.g., reentrancy attacks, unhandled exceptions, etc., is another interesting future direction.

Program synthesis. In the longer term, I am interested in making programming more accessible to everyday users via more intuitive programming interfaces, e.g., input-output examples, natural languages, diagrams, etc. Many domain-specific languages with a variety of compelling features have been proposed, but the effort to learn the new languages and refactor existing code base prevents their adoption in practice. Program synthesis can make this adoption process smoother via intuitive interfaces and higher degree of automation. The alternative programming interfaces of the program synthesis tools can also serve as an intuitive documentation of the program intent, making it easier to maintain and update programs. The core techniques of program synthesis, including inductive and deductive synthesis, has been extensively studied over the past couple of years, and have seen great advancement in scalability. It is the right time to explore adopting and improving these techniques in a more practical setting. As demonstrated in my prior research on example-based program synthesis, there are still many challenges remaining, e.g., how to extend the expressiveness of the target language while keeping the synthesis problem tractable. I will continue to innovate program synthesis techniques to improve programming efficiency for different application domains.

Concluding Thoughts

Declarative specification and its synthesis span all aspects of software systems, all the way from the front-end applications to the back-end systems and networks. I have worked on designing new specification languages with its compiler and verification tools, and program synthesis techniques that automatically generate specifications from input-output examples. As new applications emerge and the underlying system technologies evolve, many new challenges and opportunities arise. For example, how should the specification language capture the new application logic in a concise and accurate way? Given the specification, how to automatically generate efficient implementations on new system architectures? I look forward to collaborating with colleagues from all areas to tackle these new challenges.

References


