

Research Statement

Tuba Yavuz-Kahveci

My current research focuses on automated verification of concurrent software systems. I am particularly interested in symbolic model checking technique which has emerged as a promising approach for systematic exploration of large state spaces. I have designed and implemented several tools to make symbolic model checking a viable technique for the verification of concurrent software systems.

Composite Symbolic Library: Symbolic model checking owes its success to the existence of compact symbolic representations for encoding the state space of a system. I have implemented the Composite Symbolic Library to combine different, type-specific symbolic representations. A symbolic model checker that uses this library can map each type in the specification to the most appropriate symbolic representation. Since an object-oriented approach is followed in the design of the Composite Symbolic Library, it is easily extendible with new symbolic representations. It can also be used as an experimental framework for comparing the performances of alternative symbolic representations. I have designed and implemented several heuristics to optimize the performance of the Composite Symbolic Library.

Action Language Verifier: Our infinite-state symbolic model checker is built on the Composite Symbolic Library. It verifies the temporal properties, e.g., safety and liveness properties, of a given system. Since model checking of infinite-state systems is undecidable in general, it uses several heuristics to ensure the termination of the analysis. One consequence is that the analysis becomes approximate. Although the approximate analysis may sometimes be inconclusive, in some cases, it is possible to verify (or falsify) a larger set of properties by increasing the precision of the approximate analysis. Action Language Verifier has the counter-example generation capability for the properties that are falsified. This feature aids the user in debugging the specification.

Action Language Compiler: Action Language is the input specification language of the Action Language Verifier. It is a modular language for specifying the behavior of concurrent systems. It supports both true parallelism and interleaving semantics of concurrency. The communication between the concurrent parts of the specification is achieved via shared variables. Currently, boolean, enumerated, and unbounded integer types are supported. I have implemented a compiler for translating specifications written in Action Language to a transition system encoded using the composite symbolic representation.

Concurrency Control Components: I have proposed a technique for automated verification of concurrent software systems. This technique decouples the concurrency control component from the rest of the application. The model of the concurrency control component is specified in Action Language, verified with Action Language Verifier, and an efficient implementation in Java is automatically generated. I have successfully applied the technique to an airport ground traffic control simulation software.

Shape Analysis: I have proposed a technique for verifying the invariant properties of concurrent singly linked lists. The novelty of the technique lies in its ability to verify the properties which relate the shape of the data structure to the integer variables that appear in the data structure specification. I am currently extending the technique to handle a wider range of data structures, e.g., data structures with multiple fields.

Future Research Directions

My future work will explore some open problems that are critical for the success of ensuring software reliability. The ones that I am specifically interested in are:

Symbolic representations: Programming languages employ various data and control abstractions. Applicability of infinite-state symbolic model checking to software systems depends on the existence of symbolic representations that correspond to such abstractions. I would like to extend the Composite Symbolic Library with new symbolic representations for the abstractions such as arrays, strings, and unbounded channels.

Powerful Abstractions: Software may involve unbounded state spaces due to dynamic memory allocation, dynamic process creation, and so on. We need powerful abstraction techniques, which will be precise enough to reason about the system while making the analysis feasible. There are two main approaches for employment of abstraction techniques in this regard. One approach reduces the state space of a system to a finite abstraction and performs the analysis on this reduced state space. Another approach performs the analysis on the concrete/original state space, however computes an approximation to the result of the analysis. While the superiority of one approach over the other is controversial, I would like to explore ways of automatically choosing the right approach for a given system by analyzing the property to be verified.

Compositional Reasoning: Reasoning about correctness of large software systems require employing compositional reasoning techniques which infer the correctness of the whole system from the correctness of its components. One such technique is assume-guarantee style of reasoning for concurrent systems, which involves significant human guidance for finding the environment assumptions/component guarantees. I think we need techniques that will automatically infer environment assumptions by analyzing the behavior of individual components and the guarantees expected from them. Recently, there have been similar efforts at NASA Ames ASE Group for automatically generating environment assumptions for invariant properties. I am interested to extend this approach to more general properties including liveness properties.

Specification languages: Currently, there are two main approaches for model checking software. The first one is to analyze a system on the source-code level. This requires employing various abstraction techniques in order to make the analysis feasible. The second one is to design an abstract model of the software system and specify it using the input specification language of a model checker. The problem with the former is that it is a kind of reverse engineering and the verification is employed in the very late stages of the software development process. The problem with the latter is that one would like to be able to automatically generate implementation of a verified model. However, the fact that specification languages of model checkers are very low level compared to modern object oriented languages complicates the issue. I would like to explore ways for reducing the gap between the specification languages and the modern programming languages.

Embedded Systems: Embedded systems are widely used in safety-critical applications in the various sectors of the industry such as avionics, automotive, energy, and medicine. Embedded systems are mostly concurrent, subject to complex interactions with the environment, and required to operate in real-time. These characteristics of embedded systems make them more susceptible to design and implementation errors. I would like to apply automated verification techniques to these systems to increase their reliability.

Multi-agent Systems: Agents have emerged as a promising paradigm for applications that can significantly benefit from intelligent/autonomous entities such as deep space explorers that suffer from latencies in the communication between the robots and the ground control. Due to their unconventional features such as learning capability, autonomy, and complex interactions with the environment, we need new languages, formalisms, and techniques in order to effectively design and analyze multi-agent systems. I would like to adapt automated verification techniques to this active research area.