Developing Verifiable Concurrent Software

Tevfik Bultan
Department of Computer Science, University of California
Santa Barbara, CA 93106, USA
bultan@cs.ucsb.edu

1 Challenges in Dependable Concurrent Programming

Concurrent programming is an error prone task. When writing a concurrent program, a programmer has to keep track of not only the possible values the variables can take, but also the states of the threads. Automated verification techniques that rely on state exploration, such as model checking, face a similar difficulty while analyzing a concurrent program. The number of possible states increase exponentially with the number of concurrent threads, significantly increasing the cost of state space exploration. Concurrent programs are challenging both for programmers and automated verification tools for the same reason: The interactions among the threads are hard to keep track of. One can speculate that a concurrent programming approach that will enable efficient automated verification will also likely make concurrent programming easier and visa versa, i.e., improving verifiability and programmability are closely related goals.

Automated verification tools for concurrent systems employ various state space reduction techniques in order to achieve scalable verification. Two core techniques that are repeatedly used for reducing the complexity of the verification task are modularity and abstraction. Modularity and abstraction are also the two core principles in software development. Programming languages provide constructs that help programmers in reasoning about programs modularly and in obtaining mental abstractions of programs. For example, procedures enable modular reasoning. A programmer can focus on the implementation of a procedure without worrying about the implementation of the caller as long as the contract between the caller and the callee is specified. Scoping rules enable programmers to mentally abstract away the states of variables that are outside the scope of the program segment they are working on. Unfortunately, introducing concurrency to a program breaks down the modularity and abstraction supported by most programming constructs.

I believe that the key step in achieving dependable concurrent programming will be finding software development techniques that support modularity and abstraction in concurrent programs. Moreover, in order to achieve dependability, such techniques should support and integrate the modularity and abstraction at the software design level with the automated verification techniques that depend on these principles.

2 Modularity and Abstraction via Behavioral Interfaces

Modularization requires specification of module interfaces. In order to achieve modularity in verification, the interfaces have to provide just the right amount of information. If the interfaces provide too much information, then they are not helpful in achieving modularity in verification. On the other hand, if they provide too little information, then they are not helpful for verifying interesting properties. Interface of a module should provide the necessary information about how to interact with that module without giving all the details of its internal structure. The key to this problem is finding appropriate interface specification mechanisms that will allow effective specification of interfaces.

Current programming languages do not provide adequate mechanisms for representing module interfaces because they provide too little information. Think of an object class in an object oriented language. The interface of an object class consists of names and types of its fields, and names, return and argument types of its methods. Such interface specifications do not contain sufficient information for most verification tasks. For example, such an interface does not contain any information about the order the methods of the class should be called. In order to achieve modular verification, module interfaces need to be richer than the ones provided by the existing programming languages.

One possible approach is to use state machines for specification of behavioral interfaces. Interface state machines can be used to specify the order of method calls or any other information that is necessary to interact with a module. Hence, the behavioral interface of a module can also serve as its abstraction during the verification of other modules. In a way, behavioral interfaces provide a way to encapsulate behavior of a module by providing an abstraction of its environment.

As an example, consider the verification of a concurrent bounded buffer implementation. Access to the buffer operations can be protected with user defined synchronization operations. For example, one requirement could be that the synchronization method *read-enter* should be called before the *read* method for the buffer is called. Such constraints can be specified using an interface machine which defines the required ordering for the method calls. For complex interfaces one could use an extended state machine model and provide information about some of the input and output parameters in a module's interface. Other possible directions are to use hierarchical state machines or grammars for interface specification.

Behavioral interfaces enable assume-guarantee style verification that separate behavior and interface verification steps. Interfaces can be used to isolate the behavior of interest by separating it from its environment. For example, the behavior of interest could be the behavior of an object shared among multiple threads. A behavioral interface for the object class can be used to isolate the object behavior by decoupling it from its environment. Similarly, interfaces of different modules can be used to isolate the interaction among multiple modules from the module implementations.

In order to achieve modularity, during behavior verification one can assume that there are no interface violations. Based on this assumption, interfaces can be used as environment models. Environment generation is a crucial problem in software model checking. If software developers are required to write interfaces during the software development process, these interfaces can then be used as a model of the environment during behavior verification. Using such interfaces we can encapsulate the behavior in question and perform the verification on this encapsulated behavior separately. Note that, interfaces should represent all the constraints about the environment that are relevant to the behavior of interest, i.e., the interfaces should provide all the information about the environment that is necessary to verify the behavior. This is analogous to requiring programmers to declare types to enable type checking.

During behavior verification one could use domain specific verification techniques. Recall the concurrent buffer example above and assume that this buffer is implemented as a linked list. During behavior verification we can assume that the threads that access to this buffer obey its interface and we can verify the correctness of the linked list implementation without worrying about the interface violations. For the verification of the linked list we can use specialized verification techniques such as shape analysis. Since, interfaces allow isolation of the behavior, application of domain specific verification techniques (which may not be applicable or scalable in general) becomes feasible. These domain specific verification techniques may enable verification of stronger and more complex properties than that can be achieved by more generic techniques.

During interface verification we need to verify that there are no interface violations. If interfaces of different modules are specified uniformly, for example using finite state machines, this step can be handled using a uniform verification technique. Note that, this verification technique should be capable of handling different types of modules. Hence, during interface verification, it is more suitable to use the generic verification techniques developed for the purpose of applying model checking directly to existing programming languages.

During interface verification, the interfaces can be used to abstract the behavior that is verified during behavior verification. Recall the concurrent buffer example. Interface verification step for this example requires that each thread which has access to the concurrent buffer has to be checked for interface violations. During the verification of a thread, the behavior of the concurrent buffer can be abstracted by replacing the concurrent buffer by its interface machine. Note that, here, we are assuming that the concurrent buffer itself does not make calls to its methods directly or indirectly. If that is not the case, the behavior of the concurrent buffer and any other part of the code which is not relevant to the interface violations can be abstracted away using static analysis techniques such as slicing, since we are only interested in interface violations.

3 Conclusion

Writing dependable concurrent programs is a challenging task. It is likely that any software development technique that improves programmability of concurrent systems will also improve their verifiability and visa versa. Current concurrent programming techniques do not provide adequate support for modularity and abstraction. I argue that behavioral interfaces can be a useful tool in achieving modularity and abstraction in concurrent program design and verification.