Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

English summary

Software plays a crucial role in our interaction with the real world and is also embedded within some of the most critical systems. Ensuring that software is free of bugs and works as intended presents a significant challenge in software development. Object-oriented programming is a programming paradigm widely used in the development of many software systems. Applying formal specifications to verify the correctness of object-oriented programs can be very beneficial, as even a minor error within widely used programs can lead to significant issues, such as system outages and failures. This thesis demonstrates the use of formal methods for systematically verifying state-of-art, real object-oriented programs.

In Chapter 3, we focus on the formal verification of object-oriented classes. We discuss the specification and verification of a corrected version of the linked list implementation from the Java Collection framework, which originally contained an overflow bug. Our formal specification aimed at two goals: to establish the absence of the overflow bug, and to capture the essential behavior of the methods with respect to the structural properties of the linked list. We successfully demonstrated, using the KeY theorem prover, that the fixed version of the core methods of the linked list implementation in Java is formally correct.

The work on verifying linked list successfully verified most methods but excluded some method implementations that contain an interface type as a parameter. Interfaces abstract away from state and other internal implementation details, facilitating modular program development. However, tool-supported programming logics and specification languages are predominantly state-based, which as such cannot be directly used for interfaces. In Chapter 4, we introduce a novel specification method using histories, recording method calls and returns, on an interface. The abstractions over histories, called attributes, are used to describe all possible behaviors of objects regardless of its implementation. Interface specifications can then be written in the state-based specification language JML by referring to histories and its attributes to describe the intended behavior of implementations.

To demonstrate the feasibility of the history-based reasoning approach, we have specified part of the core methods of Java's Collection interfaces in Chapter 5, using the *executable* history-based approach (the EHB approach). This approach uses an encoding of histories as Java objects on the heap. That encoding, however, made use of pure methods in its specification. While the methodology works in principle, in practice, for advanced use, the pure methods were a source of large overhead and complexity in the proof effort.

To enhance the history-based approach, Chapter 6 discusses integrating abstract data types (ADTs) in the KeY theorem prover by a new approach to model data types using Isabelle/HOL as an interactive back-end, and represent Isabelle theorems as user-defined taclets in KeY. We model histories as elements of an ADT, separate from the sorts used by Java in the EHB approach (Chapter 5). Histories then can not be touched by Java programs under verification themselves. We refer to this as the *logical* history-based approach (the LHB approach). We showed how ADTs externally defined in Isabelle/HOL can be used in JML specifications and KeY proofs for Java programs. As a more advanced case study, we provide a specification of the addAll method and verify the correctness properties of its clients. Furthermore, we reasoned about advanced, realistic use cases involving multiple instances of the same interface.

Chapter 7 focuses on the use of hierarchy in object-oriented programs. Hierarchy naturally follows from behavioral subtyping, in which the subtypes must not only match the method signatures defined by their supertypes but also adhere to the intended behaviors. We develop a general history-based refinement theory that used to verify the subtype relation. We associate each interface and class with a history that represents the sequence of method calls performed on the object since its creation. The subtype relation is described in terms of projection relation, which means the relationship between subtypes and supertypes hinges on the projection between histories, with each type having its own history. Through the use of a running example, we demonstrate the practical applicability of our approach.

Programming to interfaces is one of the core principles in object-oriented programming and is central to most of the standard libraries, which provide a hierarchy of interfaces and classes that represent object containers. We proposed techniques that are capable of specifying and verifying class, interface, and hierarchy structure. Taking the Java collection framework as a case study, we show the usefulness of our techniques. Thus, this thesis provides important novel contributions, insights, and findings for the research community and future applications in the field of software verification.