Chapter 9
Modular Specification and Verification

Daniel Grahl, Richard Bubel, Wojciech Mostowski, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß

Software systems can grow large and complex, and various programming disciplines have been developed addressing the problem how programmers can cope with such complex systems. We focus in this book on the paradigm of object-orientation which seems to be the widely adopted mainstream approach.

In parallel to these development in software engineering, formal verification needs complementary techniques for dealing with large software systems and increased complexity. Yet, achieving complete functional verification of a complex piece of software still poses a grand challenge to current research ([Leino, 1995, Hoare and Misra, 2005, Leavens et al., 2006a, 2007, Klebanov et al., 2011, Huisman et al., 2015]). In this chapter we will present which support the KeY system offers in this direction and review the research background it is based on. In most subsections we will come back to concepts already presented in earlier chapters, but now with special emphasis on modularization. We will take extra pain to precisely delineate these dependencies.

It is common wisdom that the keys to scale up a technique for large applications are modularization and abstraction. In our case, the deductive verification of object-oriented software, the central pillar for modularization and abstraction is the Design by Contract principle as pioneered by Meyer [1992]. Once the contract for a method has been separately verified we need not at every call to this method inspect its code again but use its contract instead. In Chapter 7 method contracts have already been introduced as a central concept of the behavioral specification language JML (Java Modeling Language). Syntax and semantics of JML method contracts have been thoroughly explained there. Chapter 8 explained how JML method contracts are translated into proof obligations in JavaDL whose validity entails the correctness of the method w.r.t. the contract. In this chapter, we explain what needs to be considered when using a contract instead of the code of a method on the caller side and present logical calculus rules implementing this. A separate subsection is devoted to recursive methods. They are a special case since the contract to be verified is used itself at every recursive call of the method.

Method contracts can only play out their advantages if they do not themselves make use of implementation details. To achieve this it is necessary to have means...
available that abstract away from the code. To this end JML offers *model fields* and *model methods*, syntax elements that only occur in specifications and are not part of the code. These have already been addressed Section 7.7.1. Here we present in great detail the semantics of these concepts at the level of JavaDL and also show and discuss calculus rules.

Object invariants have already been addressed in Section 7.4.1. Here we present technical details, the representation of invariants as implicit model fields and how they are handled in KeY. In modular specification and verification, knowing which memory locations a method does *not* change is almost as important as knowing the effects of it. How to formalize and utilize this information is known as the *frame problem*. There is a long history of verification techniques that deal with the frame problem. Our approach, see [Weiβ, 2011], is inspired by the *dynamic frames* technique from [Kassios, 2011] that aims at providing modular reasoning in the presence of abstractions as they occur in object oriented programs. In Section 7.9 we encountered already the *assignable* clause in JML specifications that provides a set of locations that a method might at most assign to. In Section 8.2 we saw how the JML *assignable* clause is translated into the mod part of a JavaDL loop or method contract. The calculus rules for proving these contracts were already covered in Section 3.7. In this chapter rules will be presented (1) for a more fine grained treatment of anonymization in loop verification and (2) for *using* method contracts.

In a way complementary to the information which locations a method may write to is the information which locations a method may read from. How this information is formulated was already explained (1) on the JML level via *accessible* clauses in Section 7.9, (2) as a JavaDL dependency contract in Definition 8.3 and, (3) as a JavaDL proof obligation in Definition 8.5. In this chapter *accessible* clauses for model methods are introduced. The previous proof obligation for dependency contracts has to be revised to cover this extension.

Although we use Java and JML as technological basis in this chapter, we expect all mentioned concepts to be adaptable to other object-oriented programming languages and their associated specification languages.

**Chapter Overview**

We start off this chapter with introducing the basic concepts of modular specification in Section 9.1. This will explain in general method contracts, behavioral subtyping, and lead up to our formalization of modular code correctness. A running example that will be used throughout this chapter will make its first appearance in Section 9.1.2. The special case of unbounded recursion is discussed in Section 9.1.4.

We present model fields as they appear in standard JML and their role in the KeY system in Section 9.2.1, as well as the more advanced concept of model methods in Section 9.2.2. The *frame problem* is the topic of Section 9.3.

Section 9.4 takes us to the second theme in the title of this chapter—verification. Building on the calculus for JavaDL from Section 3.5, we introduce additional rules for modular reasoning. This includes 1. an improved loop invariant rule, that