Static analysis of unbounded structures in object-oriented programs

Grabe, I.

Chapter 1

Introduction

As indicated by the title our interest lies with the analysis of unbounded structures for object-oriented languages. In particular we address two sources of unboundedness that are commonly found in object-oriented languages: object creation, i.e. structural complexity, and multi-threading, i.e. behavioural complexity.

Construction and validation of programs requires a deep understanding of the sources of these complexities and ways to address them in the validation process.

Thesis est omnis divisa in partes tres. All parts address the aforementioned complexities in different ways and settings. Since the setting and approach presented in each section is introduced individually we restrain ourselves here to an overview of the overall thesis to motivate the individual parts.

First we present our approach to address structural complexity by an abstraction of the underlying representation of objects and object creation. In object-oriented programming languages like Java, objects can be dynamically created by the constructor methods provided by their class. Using constructors for object creation is an abstraction from the underlying representation of objects and the implementation of object creation. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language. We give such a representation in terms of a weakest precondition calculus for abstract object creation in dynamic logic, the logic underlying the KeY theorem prover [19]. This representation allows to both specify and verify properties of objects at the abstraction level of the programming language. Based on this weakest precondition calculus we show how to symbolically execute abstract object creation in the KeY

theorem prover.

The work presented in the first part was published as [10].

Second we present a technique to address behavioural complexity introduced by multi-threading. Multi-threaded programs can show complex and unexpected behaviour due to the interleaving of the activities of the individual processes or threads. An example of such an unexpected and undesired behaviour is a system reaching a deadlock, i.e. a situation in which processes are blocked due to mutual waiting. Sources of such mutual waiting can be for example locks either by explicit or implicit lock handling, e.g. monitor concept. Reasoning about locks in an object-oriented language the concept of reentrance comes into picture, i.e. a process that has acquired a lock (to an object) is free to pass the same lock several times (further method invocations on the object locked by the process). The lock can only be freed as soon as the initial method invocation (acquiring the lock) and all consecutive method invocations of the process to methods protected by the lock (e.g. at the object in case of a monitor) have terminated. This complicates the analysis of deadlock behaviour as it introduces the need to look at the call stack of the processes. We present an abtraction of multi-threaded programs (with respect to data and control flow) that allows us to detect deadlocks. Our technique allows for reentrant method calls. To the best of our knowledge this is the first automata based approach tailored to deadlock detection of the abstract control flow of method calls and returns of multithreaded reentrant programs.

The work presented in the second part was published as [43].

Third we extend a calculus to reason about active objects with futures and promises. We present an open semantics for the core of the *Creol* language including first-class futures and promises. A future acts as a proxy for, or reference to, the delayed result of a computation. As the consumer of the result can proceed its own execution until it actually needs the result, futures provide a natural, lightweight, and transparent mechanism to introduce parallelism into a language. A promise is a generalization of a future as it allows for delegation with respect to which process performs the computation. The formalization is given as a typed, imperative object calculus to facilitate the comparison with the multi-threaded concurrency model of object-oriented languages, e.g. *Java*.

We close the third part of this thesis by presenting a technique to detect deadlocks in concurrent systems of active objects. Our technique is based on a translation of the system to analyse into a P/T net and the application of a technique to detect termination in such P/T nets. We illustrate our techique by application to an Actor-like subset of the *Creol* language featur-

ing asynchronous calls using futures as means of communication. The so-called discipline of cooperative multi-tasking within an object as found in *Creol* can lead to deadlock. Our technique can be applied to detect such deadlocks.

The work presented in the third part was published as [4] and [44].