Static analysis of unbounded structures in object-oriented programs

Grabe, I.

Appendix A

Proofs

A.1 Wellformedness of generated sequences

In this section we present an extensive proof of Theorem 3.4.2.

Proof: We prove this by induction on the length of the derivation. We treat the following cases:

Let $I_c = \emptyset$ and $t_c \in L$ in the following derivation

$$(I, L) : G \cup \{S^t\} \Rightarrow t_r^m \ (I \cup \{t_c\}, L) : G \cup \{S^t\} \Rightarrow^* t_r^m W$$

(where m is a synchronised method of c). By the induction hypothesis we have that

- W is synchronised and
- $I \cup \{t_c\} \cup \mathrm{Lock}(W) = L$.

It follows that $t_r^m W$ is synchronised and that $I \cup \mathrm{Lock}(t_r^m W) = I \cup \{t_c\} \cup \mathrm{Lock}(W) = L$.

Let $I_c = L_c = \emptyset$ in the following derivation

$$(I, L) : G \cup \{B^t\} \Rightarrow t_r^m \ (I \cup \{t_c\}, L \cup \{t_c\}) : G \cup \{r^t\} \Rightarrow^* t_r^m W$$

(where m is a synchronised method of c). By the induction hypothesis we have that

- W is synchronised and
- $I \cup \{t_c\} \cup \mathrm{Lock}(W) = L \cup \{t_c\}$.

It follows that $t_r^m W$ is synchronised and that $I \cup \text{Lock}(t_r^m W) = I \cup \text{Lock}(W) = L$ (note that $I_c = L_c = \emptyset$ and in $t_r^m W$ all calls by t have a matching return, so $\text{Lock}(t_r^m W) = \text{Lock}(W)$).

Next we treat the case

$$(I, L) : G \cup \{r^t\} \Rightarrow (I, L) : G \cup \{B^t\}\ t_r \Rightarrow^* W t_r$$

By the induction hypothesis we have that

- W is synchronised and
- $I \cup \text{Lock}(W) = L$.

So it suffices to observe that, since there exist no pending calls of t in W, we have $\text{Lock}(W t_r) = \text{Lock}(W)$.

As a final case we treat the derivation:

$$(I, L) : G_1 \circ G_2 \Rightarrow (I, L') : G_1\ (L', L) : G_2 \Rightarrow^* W$$

This derivation is decomposed to $(I, L') : G_1 \Rightarrow^* W_1$ and $(L', L) : G_2 \Rightarrow^* W_2$, where $W = W_1 W_2$. By the induction hypothesis we have that

- W_1 and W_2 are synchronised and
- $I \cup \text{Lock}(W_1) = L'$ and $L' \cup \text{Lock}(W_2) = L$.

We first argue that $W = W_1 W_2$ is synchronised. Let t_r^m be a synchronised call in W, with m a synchronised method of c. If t_r^m appears in W_1 then there exists no preceding pending call to a synchronised method of c by another thread because W_1 is synchronised. On the other hand, if t_r^m appears in W_2 then there exists no preceding pending call (to a synchronised method of c) by another thread *in* W_2 because W_2 is synchronised. There also does not exist such a call by a thread t' different from t in W_1 because $I \cup \text{Lock}(W_1) = L'$ implies $t'_c \in L'$, which in turn rules out the call t_r^m because W_2 is synchronised.

Furthermore, if $t_c \in I$ then there exists no call in W to a synchronised method of c by another thread because both W_1 and W_2 are synchronised.

Finally, $I \cup \text{Lock}(W) = I \cup \text{Lock}(W_1) \cup \text{Lock}(W_2) = L' \cup \text{Lock}(W_2) = L$. Note that indeed $\text{Lock}(W) = \text{Lock}(W_1) \cup \text{Lock}(W_2)$ because if W_2 contains a matching return t_r for a pending call t_r^m in W_1 then $r^t \in G_2$. But this is ruled out by $(I, L) : G_1 \circ G_2 \Rightarrow (I, L') : G_1\ (L', L) : G_2$ because r^t cannot be generated by a split. □

A.2 Existence of derivation

In this section we present an extensive proof of Lemma 3.5.1.

Proof: We prove the lemma by induction on the length of the word W.

Base Case $W = \epsilon$ is straightforward by application of rule $(I, I) : G ::= \epsilon$

Induction Step Let $W = w_1, \ldots, w_n, w_{n+1}$ be the well-formed synchronised sequence. Since w_{n+1} can be either a call or a return there are two cases to deal with. According to the induction hypothesis there exists a well-formed sequence $w'_1, \ldots, w'_n$ such that $G_0 \Rightarrow^* w'_1, \ldots, w'_n$, and $w'_1, \ldots, w'_n \approx w_1, \ldots, w_n$.

First we consider the case that w_{n+1} is a method call t^m_r. We prefix the derivation $G_0 \Rightarrow^* w'_1, \ldots, w'_n$ by a composition step: $G_0 \Rightarrow G_0\ G_0 \Rightarrow^* w'_1, \ldots, w'_n G_0$, and apply the rules $G_0 ::= t^m_r G_0$ and $G_0 ::= \epsilon$ to obtain $G_0 \Rightarrow^* w'_1, \ldots, w'_n, w_{n+1}$.

The proof of the equivalence $w'_1, \ldots, w'_n, w_{n+1} \approx w_1, \ldots, w_n, w_{n+1}$ is straightforward. Appending the same call to both sequences $w'_1, \ldots, w'_n$ and $w_1, \ldots, w_n$ preserves the equality of projection and the equality of the lock sets.

Next we consider the case that w_{n+1} is a return t_r. Let $w'_i = t^m_r$ be the matching call in $w'_1, \ldots, w'_n$. For this call we can decompose the derivation into $G_0 \Rightarrow^* \ldots G \cup \{S^t\} \ldots \Rightarrow \ldots t^m_r\ G \cup \{S^t\} \ldots \Rightarrow^* w'_1, \ldots, w'_n$. In this derivation we replace the step $\ldots G \cup \{S^t\} \ldots \Rightarrow \ldots t^m_r G \cup \{S^t\} \ldots$ by the following sequence of steps

$$\ldots G \cup \{S^t\} \ldots \Rightarrow \ldots G \cup \{B^t\} \ldots \Rightarrow \ldots t^m_r\ G \cup \{r^t\} \ldots \Rightarrow \ldots t^m_r\ G \cup \{B^t\}\ t_r \ldots$$

Note that in the derivation $\ldots t^m_r\ G \cup \{S^t\} \ldots \Rightarrow^* w'_1, \ldots, w'_n$ the non-terminal S^t can be replaced by B^t since after the call t^m_r the thread t only generates a balanced sequence of calls and returns. Therefore we obtain a derivation

$$G_0 \Rightarrow^* w'_1, \ldots, w'_i, w'_{i+1}, \ldots, w'_k, t_r, w'_{k+1}, \ldots, w'_n$$

where $G \cup \{B^t\} \Rightarrow^* w'_{i+1} \ldots w'_k$. Due to the nested nature of the method calls (and the grammar rules) t_r is added to $w'_1, \ldots, w'_n$ in such a way that it appears at the end of the projection of $w'_1, \ldots, w'_i, w'_{i+1}, \ldots, w'_k, t_r, w'_{k+1}, \ldots, w'_n$ to t like it does for the projection of $w'_1, \ldots, w'_n, t_r$ to t preserving the equality of the projections. Since the same return is added to both sequences the equality of the lock sets is preserved. This establishes $w'_1, \ldots, w'_i, w'_{i+1}, \ldots, w'_k, t_r, w'_{k+1}, \ldots, w'_n \approx w'_1, \ldots, w'_n, t_r$. □

A.3 Soundness of lock handling

In this section we present an extensive proof of Lemma Lemma 3.5.2.

Proof: Instead of proving the lemma directly we prove a more general statement: If $G_0 \Rightarrow^* W$ with W synchronised with respect to I, then $(I, I \cup \text{Lock}(W)) : G_0 \Rightarrow^* W$.

We prove the lemma by induction on the length of the derivation $G_0 \Rightarrow^* W$.

Base Case $G ::= \epsilon$ is straightforward by application of rule $(I, I) : G ::= \epsilon$.

Induction Step We first treat the following cases of synchronised method calls and returns:

Pending Synchronised Call Let m be a synchronised method in class c and $G \cup \{S^t\} \Rightarrow t_r^m \; G \cup \{S^t\} \Rightarrow^* t_r^m W$ with $t_r^m W$ is synchronised with respect to I. Due to t_r^m being a pending call we derive that W is synchronised with respect to $I \cup \{t_c\}$. By the induction hypothesis we get $(I \cup \{t_c\}, I \cup \{t_c\} \cup \text{Lock}(W)) : G \cup \{S^t\} \Rightarrow^* W$. By application of rule $(I, I \cup \text{Lock}(W)) : G \cup \{S^t\} ::= (I, I \cup \text{Lock}(W)) : G \cup \{S^t\}$ in case $t_c \in I_c$, resp. application of rule $(I, I \cup \text{Lock}(W)) : G \cup \{S^t\} ::= (I \cup \{t_c\}, I \cup \text{Lock}(W)) : G \cup \{S^t\}$ with $t_c \in \text{Lock}(W)$ otherwise, we get a derivation $(I, I \cup \text{Lock}(W)) : G \cup \{S^t\} \Rightarrow^* t_r^m W$.

Matching Synchronised Call For the next case let m be a synchronised method in class c and $G \cup \{B^t\} \Rightarrow t_r^m \; G \cup \{r^t\} \Rightarrow^* t_r^m W$ with $t_r^m W$ is synchronised with respect to I. Due to $t_r^m W$ being synchronised with respect to I we conclude W is synchronised with respect to $I \cup \{t_c\}$. By the induction hypothesis we get $(I \cup \{t_c\}, I \cup \{t_c\} \cup \text{Lock}(W)) : G \cup \{r^t\} \Rightarrow^* W$. By application of rule $(I, I \cup \text{Lock}(W)) : G \cup \{B^t\} ::= (I, I \cup \text{Lock}(W)) : G \cup \{r^t\}$ in case $t_c \in I_c$, resp. application of rule $(I, I \cup \text{Lock}(W)) : G \cup \{B^t\} ::= (I \cup \{t_c\}, I \cup \{t_c\} \cup \text{Lock}(W)) : G \cup \{r^t\}$ otherwise, we get a derivation $(I, I \cup \text{Lock}(W)) : G \cup \{B^t\} \Rightarrow^* t_r^m W$.

Return of a Synchronised Method For the next case let t_r be a return to a call of a synchronised method in class c and $G \cup \{r^t\} \Rightarrow G \cup \{B^t\} \; t_r \Rightarrow^* W t_r$ with $W t_r$ is synchronised with respect to I. Due to $t_r^m W$ being synchronised with respect to I we conclude W is synchronised with respect to I. By the induction hypothesis we get $(I, I \cup \text{Lock}(W)) : G \cup \{B^t\} \Rightarrow^* W$. Since W is derived from $G \cup \{B^t\}$ it does not contain a pending call of t. It follows that

$\text{Lock}(Wt_r) = \text{Lock}(W)$. We conclude that $(I, I \cup \text{Lock}(Wt_r)) : G \cup \{r^t\} \Rightarrow (I, I \cup \text{Lock}(Wt_r)) : G \cup \{B^t\}\ t_r \Rightarrow^* Wt_r$

We treat composition as the final case

Composition $G_1 \circ G_2 \Rightarrow G_1 G_2 \Rightarrow^* W$ with W is synchronised with respect to I. It follows that $G_i \Rightarrow^* W_i$ $(i = 1, 2)$, with $W = W_1 W_2$, W_1 is synchronised with respect to I and W_2 is synchronised with respect to $I \cup \text{Lock}(W_1)$. By the induction hypothesis we get derivations $(I, I \cup \text{Lock}(W_1)) : G_1 \Rightarrow^* W_1$ and $(I \cup \text{Lock}(W_1), I \cup \text{Lock}(W_1) \cup \text{Lock}(W_2)) : G_2 \Rightarrow^* W_2$ We have that $I \cup \text{Lock}(W_1) \cup \text{Lock}(W_2) = I \cup \text{Lock}(W)$ as argued in the proof of theorem 1. So we conclude that $(I, I \cup \text{Lock}(W)) : G_1 \circ G_2 \Rightarrow^* W$ □

Abstract

Formal methods provide the foundation to reason about systems and their characteristics, e.g. safety or security properties. To be able to reason about systems and to provide valid statements about the system the formal methods must provide means to address key features of the language, e.g. concurrency or object creation. These features introduce complexity to systems and the formal methods to reason about these systems.

Both the development of hardware, e.g. multicore processors or the increase of available memory, and the development of software, e.g. networked and integrated systems and the introduction of high-level programming languages like *Java* and C♯, lead to an increased usage of the aforementioned features. New concepts to address problems like concurrency and delegation, e.g. futures and promises, are developed and promoted. Formal methods need to include these features to be capable of reasoning about state of the art software systems.

In this thesis, we show how to address the complexity introduced by modern software systems, e.g. structural complexity introduced by object creation and behavioural complexity introduced by multi-threading. We present formalisms to check interesting properties, e.g. deadlock freedom or termination, of such systems. Furthermore we show how to extend a calculus to reason about systems based on active objects with futures and promises.

In the first part of the thesis we address complexity introduced by object creation. Object creation is an abstraction from the underlying representation of objects used for example in object-oriented programming languages like *Java* or C♯. 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 a representation of a weakest precondition calculus for abstract object creation in dynamic logic which allows to both specify and verify properties of objects at the abstraction level of the programming language. We generalize this approach to allow for symbolic execution to integrate our approach to the setting of the KeY theorem prover.

In the second part of this thesis we address complexity introduced by multi-threading. Multi-threaded programs show complex behaviour due to the interleaving of activities of the individual processes and the sharing of state among these processes. One example of such complex behaviour is the so-called deadlock. Deadlock describes a situation in which concurrent processes share resources. Though shared among the processes a single access to a resource is exclusive. Depending on the order the processes are interleaved the deadlock may or may not arise. The number of interleavings of the processes is in general not bound which makes analysis hard. We present a formalism to reason about deadlock in multi-threaded systems. The formalism focuses on the control flow of such systems and abstracts from data.

In the third part of this thesis 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 technique by application to an Actor-like subset of the *Creol* language featuring 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.