Static Deadlock Detection for Active Objects*

Frank S. de Boer

CWI, Amsterdam, The Netherlands

Immo Grabe

CWI, Amsterdam, The Netherlands
Christian-Albrechts-University, Kiel, Germany

Martin Steffen

University of Oslo, Norway

Abstract

We present a static technique for deadlock detection for active objects. To do so, we introduce a novel kind of automata (visibly multiset automata, VMAs for short) to describe in an approximative manner the behaviour of one active object resp. a deadlock scenario. In this setting deadlock detection is checking the intersection of the VMAs for emptiness. We illustrate our technique in terms of the Creol language.

1 Introduction

Active objects [1] form a well established model for distributed systems. We present a static technique for deadlock detection for active objects. To do so, we introduce a novel kind of automata (*visibly multiset automata*, VMAs for short) to describe in an approximative manner the behaviour of one active object. We also use VMAs to describe possible deadlock situations. Thus, deadlock detection is reduced to language intersection and checking for emptiness. In a broad sense, therefore, the VMAs play a role comparable to (visibly) pushdown automata that have been used successfully to abstractly describe the behavior of procedural or object-oriented languages with multi-threading. The role of the stack, however, is replaced by the multiset data structure, reflecting the different concurrency model.

Our technique is illustrated in terms of the Creol language [3]. Creol is a high-level modeling language based on active objects. The communication model of

* Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services* and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

Creol is based on exchanging messages *asynchronously.* For each asynchronous method invocation, a new thread is created. This is in contrast to object-oriented languages based on multi-threading, such as *Java* or $C^{\#}$, which use synchronous message passing in which the calling thread inside one object blocks and control is transferred to the callee.

The result of an asynchronous method call is returned by a *future.* A future is created upon method invocation and available to the caller. Upon termination of the method call, the result of the computation is stored in the future by the callee. A future is local to the thread which invoked the call and can not be stored in a variable or passed around. Completion of a method call can be tested by testing its future. Requesting a result from a future is blocking, i.e., in case the result has not been computed the thread requesting the result blocks until the result is computed.

In Creol, each object acts as a monitor, i.e., at most one thread can be active within an object. Context switch is limited to so–called processor release points.

The combination of exclusive access to the processor and blocking waiting for the result of an asynchronous method call may result in a deadlock. In the following sections we present a static analysis technique for Creol programs to detect such deadlocks.

2 Deadlock Detection

For deadlock detection we restrict ourselves to Creol programs with a finite number of objects. We do not deal with object creation, i.e., we assume all objects to exist at the beginning of the program execution. Furthermore we restrict Creol to method definitions without *await* statements. An *await* statement denotes a conditional processor release point within a method body. In our setting, methods resemble atomic tasks executed by different sites in a distributed system.

In contrast to our previous work on deadlock detection for Java via context–free–language reachability [4], we use language intersection to detect deadlocks in Creol. Due to the asynchronous nature of Creol, a designated thread is created for each method call. Modeling the active objects, instead of the individual threads, is an elegant way to deal with this kind of thread creation.

We introduce **visibly multiset automata** to model active objects and possible deadlock situations. Such a deadlock situation is characterized by a cyclic chain of objects, executing threads that are waiting for a result of a method call to the next object in the chain.

In case the intersection of the automata is empty the program is deadlock free otherwise each word within the language describes a computation leading to a deadlock.

3 Visibly Multiset Automata

A visibly multiset automaton resembles a visibly pushdown automaton [2] with a multiset as storage instead of a stack. The operations on the multiset are determined by the input letter, i.e., an input letter can trigger either an add action, cf. push, or a remove action, cf. pop, but not both.

Definition 3.1 (Visibly Multiset Automaton) A (nondeterministic) visibly multiset automaton on finite words over an add-remove alphabet $\Sigma = \Sigma_{\mathrm{A}} \uplus \Sigma_{\mathrm{R}} \uplus \Sigma_{\mathrm{Int}}$ is a tuple $M = (Q, Q_I, \Gamma, \emptyset, \delta, Q_F)$, where Q is a finite set of states, $Q_I \subseteq Q$ is the set of initial states, $Q_F \subseteq Q$ is the set of final states , Γ is a finite alphabet that contains a special symbol $\emptyset \in \Gamma$ denoting the empty set, and $\delta \subseteq (Q \times \Sigma_{\mathrm{A}} \times Q \times \Gamma \setminus \{\emptyset\}) \cup (Q \times \Sigma_{\mathrm{R}} \times \Gamma \times Q) \cup (Q \times \Sigma_{\mathrm{Int}} \times Q)$ is a transition function.

To address the asynchronous communication we split each call, resp. return, into two phases. As a first step a token denoting the call, resp. the return, is added to the multiset. Later as a second step the token is removed and the call is "executed", resp. the execution is "finished". The VMAs are synchronized via their multisets.

We represent a VMA as a Multi-Stack Visibly Pushdown Automaton (MVPA) [5] with two stacks. The first stack is used to store the elements of the multiset. The second stack is used to swap stack symbols of the first stack when accessing a symbol within the first stack. With this construction we reduce the decidabilty of intersection and emptiness of VMAs to the decidabilty of intersection and emptiness of MVPAs which were shown in [5].

4 Conclusion

4.1 Results

We have introduced visibly multiset automata as a natural representation for concurrent systems communicating via asynchronous method calls.

We have presented a technique to prove absence of deadlock for asynchronous, active objects. This technique is based on a representation of the static structure of the program and a representation of the possible deadlock situation.

4.2 Future Work

We plan to investigate, object creation, inheritance, and context switches within method bodies, i.e. *await* statements. Implementation in the context of the Creol toolsuite is a goal as soon as more features of the language are covered by our technique.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In L. Babai, editor, *STOC*, pages 202–211. ACM, 2004.

[3] The Creol language. http://heim.ifi.uio.no/creol.

[4] F. S. de Boer and I. Grabe. Finite-State Call-Chain Abstractions for Deadlock Detection in Multithreaded Object-Oriented Languages (extended abstract). In E. B. Johnsen, O. Owe, and G. Schneider, editors, *Proceedings of the 19th Nordic Workshop on Programming Theory (NWPT'07) (Abstracts)*, Oct. 2007. University of Oslo, Dept. of Computer Science, Research Report 366.

[5] S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. *Logic in Computer Science, Symposium on*, 0:161–170, 2007.