Science of Computer Programming 100 (2015) 28–60

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Shared contract-obedient channels

Étienne Lozes [a,*], Jules Villard [b]

[a] Universität Kassel, Germany
[b] University College London, UK

ARTICLE INFO

Article history:
Received 15 January 2013
Received in revised form 24 May 2014
Accepted 22 September 2014
Available online 2 October 2014

Keywords:
Program verification
Message passing
Linear channels
Separation logic

ABSTRACT

Recent advances in the formal verification of message-passing programs are based on proving that programs correctly implement a given protocol. Many existing verification techniques for message-passing programs assume that at most one thread may attempt to send or receive on a channel endpoint at any given point in time, and expressly forbid endpoint sharing. Approaches that do allow such sharing often do not prove that channels obey their protocols. In this paper, we identify two principles that can guarantee obedience to a communication protocol even in the presence of endpoint sharing. Firstly, threads may concurrently use an endpoint in any way that does not advance the state of the protocol. Secondly, threads may compete for receiving on an endpoint provided that the successful reception of the message grants them ownership of that endpoint retrospectively. We develop a program logic based on separation logic that unifies these principles and allows fine-grained reasoning about endpoint-sharing programs. We demonstrate its applicability on a number of examples. The program logic is shown sound against an operational semantics of programs, and proved programs are guaranteed to follow the given protocols and to be free of data races, memory leaks, and communication errors.

0. Introduction

Message-passing idioms appear everywhere in today's software: from the Message Passing Interface (MPI) used in high-performance computing, to the inter-process communication layer in Android apps, and to Web services. As for other forms of concurrency, naively checking the correctness of a message-passing system is severely impaired by the combinatorial explosion of the number of possible interactions between the components of the system. One way to tackle this issue is to develop formal verification techniques for message-passing programs, such that reasoning about a system is tantamount to reasoning about each component in isolation. A promising avenue in this respect is to separate the study of programs from the study of the protocols they are meant to implement, i.e. prove that a program correctly implements a protocol on the one hand, and reason about that protocol independently of its implementation on the other hand. In this context, protocols act both as specifications of what a program is allowed to do and as descriptions of the actions that programs must expect from the environment. Two main approaches coexist for describing such protocols: session types on the one hand [30], used to police interactions in programs expressed either in the π-calculus [20] or in a message-passing variant of Java [21], and *channel contracts* on the other hand [7], used for instance to describe the protocols in the Sing♯ programming language [13] developed for the Singularity operating system [22]. High-level protocol descriptions such as these allow the program verification effort to be split between checking properties at the level of the protocol itself on the one hand, and checking obedience of each thread of the program to its part in the protocol on the other hand. If all threads play their parts

* Corresponding author.

http://dx.doi.org/10.1016/j.scico.2014.09.008
0167-6423/

according to the protocol, then the program as a whole inherits the good properties of that protocol. This idea underpins many analyses of message-passing programs; it has been made more explicit in recent works both on channel contracts [32] and on session types [11].

Most of the existing verification techniques for message-passing programs assume that channel endpoints are used in a linear fashion: no two threads may ever try to send or receive simultaneously on the same channel endpoint. This allows the checking of the conformance to a given protocol to be local to each thread or process. Without this restriction, different threads sharing the same endpoint might have discordant views about at which point in the protocol that endpoint is, which greatly hinders the proof that threads indeed abide by that protocol. However, imposing that endpoints are used linearly reduces the scope of these techniques, as many useful paradigms require some form of endpoint sharing between processes. Moreover, this restriction enforces some form of determinism on programs, which excludes the encoding of standard synchronisation primitives such as locks and semaphores [14].

This work presents a program logic for message-passing programs that may share channel endpoints, while giving meaningful protocols to their interactions. The program logic achieves two goals: on the one hand, checking that the exchange of messages on the channels used by programs obeys protocols defined by *channel contracts*, a particular representation of protocols as communicating finite state machines; on the other hand, ensuring the absence of data races and resource leakage. The program logic allows two forms of sharing on channel endpoints.

The first form of sharing allows threads to use endpoints concurrently as long as they do not advance the protocol state. This ensures that a consistent view of the contract state is maintained amongst all sharers of a given endpoint. This form of sharing is useful when shared endpoints exchange only one kind of messages, and unidirectionally, as long as the sharing subsides. This is the case for instance when one or several producers send the same kind of message repeatedly over a channel, to be received by one or several consumers.

The second form of sharing allows several threads to compete for reception of a message on a shared endpoint. Exclusive access to this very endpoint is granted to the winner, who can then use it to realise the rest of the communication. Meanwhile, the other threads are kept waiting for the initial message until the protocol comes back to the initial state and the ownership of the endpoint is released by the winning thread. This form of sharing is typical of two situations occurring in existing message-passing applications. First, it can occur when several worker threads or processes listen for the same kind of event, each individual event being picked up by one worker only. This is the case for instance for *implicit intents* in the Android framework [1], where components can register as able to provide certain services, later to be called by applications in need of these services. For instance, web browsers register as being able to process intents of the “browsable” category. Clicking on a web link in an email emits an implicit intent of that category, which can be picked up by the web browser. In the case of Android, such sharing results in simple protocols: either a single intent is sent as in the example above, or the intent awaits a response. We show that, in fact, any protocol can be realised on the shared endpoint once it has been acquired in such a way. Surprisingly, sharing in this way does not contradict linear channel usage: each endpoint is effectively used by at most one thread at a time even though several threads compete for the initial message. More generally, this work shows how linearity can be relaxed to get both expressive protocols and sharing. In doing so, we open the way for bringing more forms of sharing to new message-passing run-time libraries based on formal methods in general, such as session types or Sing♯, which are currently more strict in their enforcing of linearity.

This work builds on a previous approach based on a marriage of separation logic and channel contracts [33], which forbade any active sharing of endpoints. As in previous work [26], we consider a simple imperative language that features primitives for dynamically creating and destroying bi-directional, asynchronous channels, each made of two endpoints, and for sending and receiving messages on individual endpoints. Each message is composed of a user-defined tag (which can be used to describe the kind of payload supposed to be transferred, as in Sing♯ or MPI) and zero or more values. Crucially, values may be references to other endpoints, and thus sending a message may create sharing of resources and foster concurrency errors. In the absence of endpoint sharing, previous work was able to prove obedience to channel contracts and absence of data races, and from that to deduce the absence of message reception errors and of orphan messages. A key ingredient of the program logic, which we have retained, is to logically reflect the transfer of a message that is attached to a resource by the transfer of the ownership of that resource (the message’s *footprint* in memory) to the recipient of the message. Contrarily to similar transfer disciplines such as session types, footprints need not be syntactically determined by the value sent: sending the address of an endpoint over a channel does not equate sending the ownership of that endpoint. Rather, any footprint may be attached to the message. This makes our approach more powerful in terms of which programs can be proved correct, as it allows more complex ownership transfer disciplines.

The contributions of the present paper are as follows:

- We identify two patterns for sharing endpoints while retaining the ability to describe their interactions via meaningful protocols.
- We introduce a new program logic able to prove programs that abide by these patterns. The presentation of the program logic unifies both sharing paradigms.
- We justify the soundness of our program logic by proving it sound with respect to an operational semantics. Our soundness proof is able to express the fact that proved programs obey their contracts and are free of communication errors, by linking the semantics of programs to that of the channel contracts they implement.