Futures, Histories and Smart Contracts

Elahe Fazeldehkordi and Olaf Owe

Department of Informatics, University of Oslo, Norway
{elahefa,olaf}@ifi.uio.no

Abstract

In this work we reconsider the concept of shared futures as used in the setting of asynchronous method-oriented communication. The future mechanism offers some advantages such as decoupling of method invocation from retrieval of the result, and sharing of the result. However, it also has some drawbacks, including the need of garbage collection and poor privacy protection. We here suggest a more general and more expressive notion than futures, but without the two mentioned drawbacks. We use the term *history objects* for this new concept since it encapsulates past communications in addition to the information normally found in futures. We argue that the suggested new concept has several advantages, including support of functionalities found in smart contracts.

Keywords: Futures; Transactions; Asynchronous Communication; Smart Contracts.

Introduction

We reconsider the future concept, which has become popular in the setting of concurrent objects (or agents) communicating asynchronously. This setting is adopted in the active object paradigm, supported by several languages [2]. Remote method calls are handled by message passing and the result of a method invocation is placed in a future object, at which time the future is said to resolved. The caller generates a reference to the future object and this reference may be passed to other objects. Any object with a reference to the future object may ask for the value, typically via a *get* statement, which will block when the future is not yet resolved (some languages allow polling to check if a future is resolved). The main advantages of the future mechanism are improved flexibility compared to the traditional blocking remote call mechanism, delaying or avoiding the blocking, and providing safe sharing of results since a future is a write-once, multiple-read data store. Without polling the future mechanism is race-free, since a get operation waits while the future value is not there. Polling, would make an object sensitive to the speed of object in the environment (something which is often acceptable).

However, it is not trivial to detect when a future can be discarded; and as many futures may be generated, garbage collection is in general needed. In the active object paradigm, this is a clear disadvantage since the active objects themselves have a long life time. If local data inside objects is defined by data types, using a functional programming language to express and manipulate values of the data type (such as in the Creol and ABS languages), there is no need for general garbage collection of these values, assuming storage for values of the data types can be retrieved efficiently. Another disadvantage of the future mechanism is that the future value is unprotected, and an object getting the value may not know where the future came from and what it represents. In particular, privacy aspects are unknown and the information can easily be misused [4].

In this work we propose a new language construct to reduce these drawbacks. We propose a “container box” for recording all calls and futures related to the interactions involving a given object. This “container” will then hold all future values generated by the given object in the same “box”, including present and past communications. For this reason we will call it *history object.* For simplicity we consider one history object for one active object, but one could associate several history objects with one object, reflecting the interactions according to different interfaces of the object. The aim of this abstract is to sketch and motivate this new language primitive.

History objects

Notation: We consider a syntax similar to that of the Creol/ABS language family [3]. We let v denote a variable, x a formal parameter (assumed to be read-only), o an object variable (an object reference), f a future variable (a reference to a future object), e an expression (assumed to be pure), and m a method. We use capitalized words for types and interfaces, while variable and method names start with a lower-case character. We use [...] for lists and < ... > for tuples. The statement syntax v: + e is permitted when v is a list to express that an element e is appended to the tail of list v. This statement adheres to the *write-once* discipline since it cannot change previously written elements.

As is the case in the traditional future concept, we consider three kinds of method calls:

- A *blocking call* has the syntax $v := o.m(\overline{e})$, where o is the callee, m the called method, $\overline{e}$ the (input) parameters. The method result will be assigned to v.
- A *simple asynchronous call* has the syntax $o!\,m(\overline{e})$. The caller is not blocked and the result value is not communicated to the caller.
- An *asynchronous call* has the syntax $f := o!\,m(\overline{e})$ where f is a future variable declared with type $Fut[T]$ where T is the return type of m. The variable f is assigned a new call identity (a reference to the future object) uniquely identifying the call. This identity may be communicated to other objects. The caller is not blocked. In order to obtain the value returned from the call, the caller object (or another object that knows the future identity) may perform $v :=$ **get** f where v is a program variable of type T. This statement will block if the future is not resolved, and otherwise the result value is copied into v.

Consider now our proposed setting (a bit simplified) where we associate a history object, $history(o)$, to each object o called with an asynchronous call. We allow the same call operations as above. But there are two major differences: In our proposed solution, the same history object contains all future values generated by a given callee object, including those of the past as well as future ones. This history object is therefore long-lived and need not be garbage-collected. However, as the storage need is growing dynamically, the history objects could be placed in a cloud. The history object could be split over several objects if needed. (In that case for each function definition in the top level history object, the function value over the empty history must correspond to the function value over the final history in the underlying object.)

Secondly, we treat the history objects as normal objects, which means that one may communicate with the history objects when desired, using simple or blocking calls. This is following the spirit of [5]. Moreover, the behavior of the history objects is given by interface declarations, including a *get* method corresponding to reading a future value. A predefined class implementation can be given, and by means of inheritance, this class declaration may be extended and modified in the same way as other classes. In particular one may add functionality by implementing new interfaces and methods, and one may add protection mechanisms in redefined *get* methods to implement security and privacy restrictions. For instance by requiring that the caller satisfies some conditions. In the case of a two-party contract, we can require that only the two parties may see the history (through *get* calls). This means that calls to the history objects may distinguish their behavior depending on the caller (using the implicit parameter caller and its interface).

The predefined history object contains a (private) transaction list

```
List[Transaction] trans = empty
            // restricted by incremental-write/multiple-read access
```

and a predefined (hidden) *put* method to be used only by the underlying runtime system for recording each new message to or from the object by appending it to the transaction list.

```
Void put(Transaction t) {trans :+ t} //appending t to trans
```

We allow read access to the transaction list. The transaction corresponding to a call $f := o!\,m(\overline{e})$ is a tuple of form

$$< fid, caller, method, par, result >$$

where fid is a future identity (the value assigned to f), *caller* is the caller identity, *method* is the method name (m), *par* is the input values (the values of $\overline{e}$), and *result* is the result of the call, possibly *error*. This transaction is generated when the call has completed normally or abnormally (i.e., resulted in an error). The statement **return** e in the body of a method $m(\overline{T\ x})$ is executed at runtime by doing $history(this).put(< callid, caller, "m", [\overline{x}], e >)$ where this refers to the current object, $\overline{x}$ are the formal parameters, and where *callid* and *caller* are implicit parameters in our setting giving the future identity of the call and the caller object, respectively. (The *put* statements do not appear in the program code.) Abnormal termination results in a *put* call with result *error*. Similarly, a history object includes a public *get* method T **get**$(Fut[T]\ f)$ for each method result of type T. A get statement is therefore possible in our setting as a blocking call to *get* on the appropriate history object.

We observe that the transaction history of an active object as given by its history object, is sufficient to define the state of the object at the end of a method execution. The state of an active object at its last method completion can be reconstructed from the transaction history, and also the pre-state of method execution resulting in error. (Cooperative scheduling would require more events to be recorded in the *trans* variables.) We also observe that the history objects define the transaction history in a faithful way due to the language restriction on the *trans* variables by means of incremental-write by the underlying runtime system and is immutable for others. This restriction is also imposed on subclasses of the history classes. This means that one may rely on the transaction histories in a way similar to smart contracts. This is checked statically. If the runtime system also offers protection of unauthorized write access to these variables by means of a trusted execution environment, one does not need block-chain technology to guarantee incremental-write/multiple-read access. If not, one may use an underlying block-chain technology at runtime to obtain full trust. An active object with an associated history object may be used to accommodate the functionality of a smart contract, and with immutability of the transaction history guaranteed at the software level. We support the following aspects of smart contracts:

- *Trust* is provided by the reliable independent history object.
- *Reliability* due to the write-once (by the underlying operating system) and multiple-read access by users.
- A way to *formally specify and verify contract requirements* as well as invariants using the communication history and defining functions over the history.
- *Roll-back* possibility in case a transaction cannot complete.

Further details and examples will be given in an extended version of this paper, where we will also consider an example of a smart contract for auctions [1].

References

[1] W. Ahrendt, G. J. Pace, and G. Schneider. Smart contracts: A killer application for deductive source code verification. In P. Müller and I. Schaefer, editors, *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of His 60th Birthday*, pages 1–18. Springer, 2018.

[2] F. de Boer and et al. A survey of active object languages. *ACM Comput. Surv.*, 50(5):1–39, 2017.

[3] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig. F. S. de Boer, and M. Bonsangue, editors, *Revised Papers from 9th Int. Symp. on Formal Methods for Components and Objects, FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[4] F. Karami, O. Owe, and T. Ramezanifarkhani. An evaluation of interaction paradigms for active objects. *J. Log. Algebr. Methods Program.*, 103:154–183, 2019.

[5] I. Sergey and A. Hobor. A concurrent perspective on smart contracts. arXiv preprint 1702.05511, 2017.