A High-Level Language for Active Objects with Future-Free Support of Futures *

Toktam Ramezanifarkhani[1], Farzane Karami[1], and Olaf Owe[1]

Department of Informatics, University of Oslo, Norway

Keywords: Active Objects; Asynchronous Methods; Distributed Systems; Futures;

Introduction

The Actor model [1] has been adopted by a number of languages as a natural way of describing distributed systems. The advantages are that it offers high-level system description and that the operational semantics may be defined in a modular manner. The Actor model is based on concurrent units communicating by means of message passing. A criticism of message passing has been that its one-way communication paradigm may lead to complex programming when there are dependencies among the incoming messages.

The Actor-Based Concurrent Language (ABCL) is a family of programming languages based on the Actor model [3]. It makes use of *futures* [2] in order to make the communication more efficient and convenient. A future is a read-only placeholder for a result that is desirable to share by several actors. Future identities can be passed around as first class objects. This model is suitable for modeling of service-oriented systems, and gives rise to efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling lock operations. The notion of *promises* gives even more flexibility than futures by allowing the programmer to talk about the result of call even before the call has been made.

One may combine the Actor model and object-orientation using the paradigm of concurrent, active objects, and using methods rather than messages as the basic communication mechanism [7]. This opens up for two-way communication. This is for instance done by the Creol language [5] using so-called *call labels* to talk about calls, implementing method calls and replies by asynchronous method passing. Creol introduced *cooperative scheduling*, allowing mechanisms for suspension and process control. A process may suspend while waiting for a condition or a return value. For instance `await f?` makes a process suspend until the reply associated with label `f` appears, resulting in passive waiting. One may also make use of the future mechanism to generalize this setting so that several objects may share the same method result, given as a future. For instance the ABS language [6] is based on the Creol concurrency model, allowing the call labels of Creol to be first class, thereby supporting futures.

In this setting the two-way communication mechanism is replaced by a more complex pattern, namely that a method call generates a future object where the result value can be read by a number of objects, as long as they know the future identifier. Thus for a simple two-way call, the caller will need to ask or wait for the future. This means that each call has a future identity, and that the programmer needs to keep track of which future corresponds to which call. Our experience is that futures are only needed once in a while, and that basic two-way communication suffices in most cases. Thus the flexibility of futures (and promises) comes at a cost. Moreover, implementation-wise, garbage collection of futures is non-trivial, and static analysis of various aspects, such as deadlock, in presence of futures is more difficult. With futures, even normal calls are more complex due to the overhead of the future mechanism.

*Work supported by the *IoTSec* and *DiversIoT* projects (Norw. Research Council) and SCOTT (EU, JU).

In this paper we consider the setting of active objects and compare a future-less programming paradigm to the programming paradigm of future-based interaction. For the future-less programming paradigm we choose a core language derived from Creol, but without call labels nor futures. Comparison of paradigms can be done with respect to several dimensions and criteria. We will use the fairly obvious criteria given by *expressiveness*, *efficiency*, *syntactic complexity*, and *semantic complexity*. Other criteria could also be relevant, such as information security aspects and tool friendliness.

Future mechanisms

Languages may have explicit or implicit support of futures [4, 2]. Implicit futures support the "wait by need" principle. However, when considering cooperative scheduling it is essential that the suspension points are explicit, and we therefore focus on explicit support of futures in the comparison below. Languages based on explicit futures have (a subset of) the following mechanisms (providing ABS style syntax):

- creation of a future (f:=o!m(e))
- first class future operations (assignment, parameter passing)
- polling a future, i.e., using an if-statement to check if a future is resolved (if f? then .. else ..)
- waiting for a future while blocking, i.e., active waiting (x:= get f)
- waiting for a future while suspending, i.e., passive waiting (await f?)

Here f is a future variable, m a method, o an object, e a list of actual parameters, and x a program variable. A non-blocking version of get, can be done by await f?; x:= get f, and is abbreviated await x:= get f. In general, polling may lead to complicated branching structures, and is often avoided in languages with support of explicit futures.

A high-level, future-less language for active objects

We build on the Creol model for active objects, but avoid call labels (and futures). Object interaction is done by so-called asynchronous method calls, implemented by asynchronous message passing. This means that communication is two-way, passing actual parameters from the caller to the callee object when a method is called, and passing method return values from the callee to the caller when the method execution terminates. We include the Creol primitives for process control and conditional suspension, using the syntax await condition, where condition is a Boolean condition. The syntax for method calls is as follows:

- x:=o.m(e)[s] for a blocking call where s is done while waiting for the future to be resolved, and if needed, active waiting happens after s (as in f:=o!m(e); s; x:= get f, using Creol)
- await x:=o.m(e)[s] for a non-blocking call, where the suspension point is after s (as in f:=o!m(e); s; await x:= get f, using Creol/ABS)
- o!m(e), for calls where no return value is needed.

Here [s] may be empty as in x:=o.m(e)/await x:=o.m(e), or may include additional calls as in for instance await x:=o1.m1(e1)[<calculate e2>; await y:=o2.m2(e2)[s]], where the suspension point is after s, passively waiting for *both* calls to complete. In this manner, programs with nested call-get structures can be expressed without futures.

For the comparison we note that the future mechanism involves non-trivial garbage collection. Even if a future is short-lived, it may be complex to detect when it is no longer needed.

Comparison

By defining "future" classes supporting the future primitives above, as illustrated below, we show that our high-level core language is expressive enough to define futures, by means objects of (one of the) future classes. This means that efficient two-way interaction is directly supported, without garbage collection and future objects, while futures can be obtained, when needed, by using future objects. In the former case, efficiency is better than in an implementation using futures, in the second case it is similar (modulo optimizations). For programs with a majority of two-way interaction, efficiency is improved by our paradigm. We also note that programming with two-way interaction is conceptually simpler, since the declaration and usage of future variables are avoided. This is also beneficial for static analysis, since in static analysis of future retrieval (get) one typically needs to associate a call statement with each get statement. This can in general be difficult, and it is less modular when these associations cross class boundaries. Program reasoning is also more complex in the presence of first class futures [8].

Our language is able to encode futures in a straight forward manner. For instance the ABS code f:=o!m(e) is imitated by f:= new Fut_m(o,e) in our language, where class Fut_m is a predefined class, outlined below with initial code, a local method start , and exported methods:

```
class Fut_m(o,par) {
  Bool res:= false; // is the future resolved?
  T value; // the value of the future when resolved
 {start()} // initial code
  Void start(){await value:=o.m(par); res:=true} // see comment below
  Bool resolved(){return res} // polling
  Bool await_resolved(){await res; return true} // waiting until resolved
  T get(){await res; return value} // waiting for the resolved value
}
```

In start we use await when polling is allowed, then the future object will be able to perform incoming call requests, and for instance return the appropriate result of polling requests. (The class parameters should here have the types given by the method m.)

A more detailed comparison will be made in the full paper.

References

[1] C. Hewitt, P. Bishop, R. Steiger: A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI. 1973.

[2] H. Baker, C. Hewitt: The Incremental Garbage Collection of Processes. Proc. Symposium on Artificial Intelligence Programming Languages, ACM Sigplan Notices 12, 8. pp. 55-59. 1977.

[3] ABCL: An Object-Oriented Concurrent System. A. Yonezawa ed, MIT Press 1990.

[4] R. H. Halstead: MultiLisp: A Language for Concurrent Symbolic Computation. TOPLAS, 1985.

[5] E. B. Johnsen, O. Owe: An Asynchronous Communication Model for Distributed Concurrent Objects, Journal of Software and Systems Modeling 6(1): 39-58, Springer 2007.

[6] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen: ABS: A Core Language for Abstract Behavioral Specification. Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, LNCS vol. 6957, pp. 142-164. 2010.

[7] F. de Boer et al: A Survey of Active Object Languages. ACM Computing Surveys 50(5):1-39, 2017.

[8] C. C. Din, O. Owe: Compositional reasoning about active objects with shared futures. Formal Aspects of Computing, vol. 27, Issue 3, pp 551-572. May 2015.