Deadlock analysis with behavioral types for actors.

Vincenzo Mastandrea[1,2]

[1] University Nice Sophia Antipolis, CNRS, I3S, UMR 7271, France
[2] INRIA - Sophia Antipolis Méditerranée

1 Introduction

Actors are a powerful computational model for defining distributed and concurrent systems [1,2]. This model has recently gained prominence, largely thanks to the success of the programming languages Erlang [3] and Scala [9]. The actor model relies on a few key principles: (*a*) an actor encapsulates a number of data, by granting access only to the methods inside the actor itself; (*b*) method invocations are *asynchronous*, actors retain a queue for storing the invocations to their methods, which are processed sequentially by executing the corresponding instances of method bodies. The success of this model originates at the same time from its simplicity, from its properties, and from its abstraction level. Indeed, programming a concurrent system as a set of independent entities that only communicate through asynchronous messages eases the reasoning on the system.

1.1 Problem: Actors and synchronizations.

Actors do not explicitly support synchronization: requests between actors are in general remote procedure calls. The only guarantee of asynchronous messages is the causal ordering created by the communication. The retrieval of the result of an asynchronous message is usually simulated by a callback mechanism where the invoker sends its identity and the invoked actor sends a result message to the invoker. However callbacks introduce an inversion of control that makes the reasoning on the program difficult. Henceforth, providing synchronization as first-class linguistic primitive is generally preferable.

Some languages extend the actor model and provide synchronizations by allowing methods to return values. In general, this is realised by using *explicit futures*. A method of an actor returns a special kind of objects called *future*; in turn the type system is extended so that some values are tagged with a *future type*. A special operation on a future allows the programmer to check whether the method has finished and at the same time retrieves the method result. The

V. Biló, A. Caruso (Eds.): ICTCS 2016, Proceedings of the 17th Italian Conference on Theoretical Computer Science, 73100 Lecce, Italy, September 7–9 2016, pp. 257–262 published in CEUR Workshop Proceedins Vol-1720 at `http://ceur-ws.org/Vol-1720`

drawback of this approach is that programmers must be aware of futures and must know how to deal with them.

We study a different extension of the actor model that uses *implicit futures* and a *wait-by-necessity* strategy: the caller synchronizes with a method invocation only when its returned value is *strictly* necessary [4]. This strategy does not require explicit synchronization operators and ad-hoc types: the scheduler stops the flow of execution when a value to be returned by a method is needed for computing an expression. The synchronization becomes data-flow oriented: if some data is accessed and this data is not yet available, the program is automatically blocked. This way, an actor can return a result containing a future without worrying about which actor will be responsible for synchronizing with the result: the synchronization will always occur as late as possible. Replacing a future by its value is no more an operation that has to be explicitly written by the programmer, as it automatically happens at some point of the computation that can be optimized by the designer of the language runtime. We defined a simple actor calculus with wait-by-necessity synchronizations, called `gASP` [6].

While synchronization is useful, if it used improperly it can cause *deadlocks* (deadlocks cannot occur in the basic actor model). Deadlock detection is a sensible issue, in particular because it is hard to verify in languages that admit systems with unbounded (mutual) recursion and dynamic actor creation.

The following example illustrates the expressiveness of (implicit) futures and the difficulties of deadlock analysis:

```
01 Int fact(Int n, Int r){
02     Act x; Int y;
03     if (n == 0) return r;
04     else {  x = new Act(); r = r*n; n = n-1;
05             y = x.fact(n,r); return y;  }}
```

The access to `fact(n,1)` boils down to exactly `n` synchronizations. Indeed, since the value of `y` is never accessed within the method, the future is returned to the caller. When accessing the value of `fact(n,1)` a synchronization is performed on the result of the first nested invocation `fact(n-1,n)` which will need to access the result of the next invocation `fact(n-1,n*n-1)`, and so on. Technically, let the type of an asynchronous invocation be called *future type*. Then the type of `fact(n,r)` is a *recursive future type*. Because of this type, it is not possible to determine at compile time how many explicit synchronizations happen when the value of `fact(x,1)` is needed, with `x` unknown.

1.2 A technique for deadlock analysis.

To address (static-time) deadlock detection of `gASP` programs, we rely on a technique that has been already used for pi-calculus [7] and for a concurrent object-oriented calculus called (`core`) `ABS` [5,8]. Our technique consists of two modules: a **front-end type (inference) system** that automatically extracts abstract behavioral descriptions relevant to deadlock analysis from `gASP` programs, called *behavioral types*, and a **back-end analyzer of types** that computes a model of dependencies between runtime entities using a fixpoint technique.

According to this technique, a synchronization between actors α and α' is modeled by a dependency pair (α, α'), which means that the termination of

a process of α depends on the termination of a process of α'. Programs are denoted by finite models that are sets of relations on names. If a circular dependency $(\alpha_1, \alpha_2) \cdots (\alpha_{n-1}, \alpha_n)(\alpha_n, \alpha_1)$ is found in one of the relations, then the corresponding program may manifest a deadlock.

Synchronization on explicit futures boils down to checking the end of a method execution and retrieving the returned object, the retrieved object can be a future itself. On the contrary, with wait-by-necessity, if a computation requires a not-yet available value then a synchronization occurs, until a proper value is available. Retrieving this value might require to wait for the termination of several methods. Indeed, consider the factorial example, let β be the actor needing the value of `fact(n,1)`. This synchronization requires that β simultaneously synchronizes with all the actors computing the nested factorial invocations, say $\beta_1, \ldots, \beta_{\mathtt{n}-1}$. A translation from `gASP` to `ABS` would require to know statically the number n of synchronisation to perform. From the analysis point of view, this means that we have to collect all the dependencies of the form $(\beta, \beta_1), (\beta, \beta_2), \ldots, (\beta, \beta_{\mathtt{n}-1})$. In [5,8], this collection was done step-by-step by generating a dependency pair for every explicit synchronization. For synchronization on implicit futures, we need to generate a sequence of dependence pair when a value is needed, and this sequence is not bound statically.

1.3 Main contribution.

Addressing adequately implicit futures amounts to define a new type system of the above program and adapt in a non-trivial way the analyzer. The challenge we address is the ability to extend the synchronization point so that an unbounded number of events can be awaited at the same time. Our solution first extends the behavioural type with *fresh future identifiers* and to introduce specific types that identify whether a future is synchronised or not. A method signature also declares the set of actors and futures it creates to handle the potential unbounded number of future and actor creations. Then, we exploit the relation that exists between the number of dependencies of a synchronization and the number of nested method invocations. Instead of associating dependencies to synchronization points, *we delegate the production of the dependencies to method invocations*, each contributing with its own dependency. The sequence of dependencies is unfolded during the analysis. To implement this methods types of `gASP` carry an additional formal parameter, called *handle*, which is instantiated by the actor requiring the synchronization when this happens. The evaluation of behavioural types in the analyzer also carries an environment binding future names to their values (method invocations).

2 Behavioral Types

The deadlock detection technique we present uses abstract descriptions, called *behavioral types*, that are associated to programs by a type system. The purpose of the type system is to collect dependencies between actors and between futures and actors. At each point of the program, the behavioral type gathers informations on

local synchronizations and on actors potentially running in parallel. We perform such an analysis for each method body, gathering the behavioral information at each point of the program.

A *behavioral type program* is a pair $\left(\mathcal{L}, \Theta \cdot \mathtt{L}\right)$, where $\mathcal{L}$ is a *finite set* of *method behaviors* $\mathtt{m}(\alpha, \overline{\mathtt{x}}, X) = (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathtt{L}_{\mathtt{m}})$, with $\alpha, \overline{\mathtt{x}}, X$ being the *formal parameters* of $\mathtt{m}$, $\Theta_{\mathtt{m}}$ the *future environment* of $\mathtt{m}$, $\mathtt{L}_{\mathtt{m}}$ the behavioral types for the *body* of $\mathtt{m}$, and Θ and $\mathtt{L}$ are the *main future environment* and the *main behavioral type*, respectively. A future environment Θ maps future names to future behaviors (without synchronization information) $\lambda X.\mathtt{m}(\alpha, \overline{\mathtt{x}}, X)$. In the method behavior, the formal parameter α corresponds to the identity of the object on which the method is called (the `this`), while X, called *handle*, is a place-holder for the actor that will synchronize with the method. In practice several actors can synchronise with the same future, but only one at a time. $\overline{\mathtt{x}}$ are the type of the method parameters. The binder $(\nu\overline{\varphi})$ binds the occurrences of $\overline{\varphi}$ in $\Theta_{\mathtt{m}}$ and $\mathtt{L}_{\mathtt{m}}$, with φ ranging over future or actor names.

The basic types $\mathbb{r}$ are used for values: they may be either $\square$, to model integers, or any actor name α. The extended type $\mathbb{x}$ is the type of variables, and it may be a value type $\mathbb{r}$ or a *not-yet-synchronized type* $\mathbb{r}_f$ (in order to retrieve the value $\mathbb{r}$ it is necessary to synchronize the future f). The behavioral type $\mathtt{0}$ enforces no dependency, (κ, α) enforces the dependency between κ and α meaning that, if κ is instantiated by an actor β, β will need α to be available in order to proceed its execution. f_κ may represent different behaviors depending on the value of κ: $f_\star$ represents an unsynchronized future f, which is a pointer in the future environment to the corresponding method invocation; f_α represents the synchronization of the actor α with the future f; f_X represents the return of a future f by the method associated to the handler X. The type $\mathtt{L} \,\&\, \mathtt{L}'$ is the *parallel composition* of $\mathtt{L}$ and $\mathtt{L}'$, it is the behavior of two methods running in parallel and not necessarily synchronized. The sum $\mathtt{L} + \mathtt{L}'$ it is the composition of two behaviors that cannot occur at the same time, either because one occurs before the other or because they are exclusive.

In general, a statement has a behavior which is a sum of behaviors. Each term of the sum is a parallel composition of synchronization dependencies and unsynchronized behaviors. We propagate this way the set of methods running in parallel as a set of not-yet-synchronized futures all along the type analysis. The statements that create no synchronization at all (i.e. that do not access a future, nor call a method, nor return from a method) have behavior 0.

Example. The behavioral type associated to the following program is $(\mathtt{fact\_d}(\alpha, \square_f, X) = (\nu\, f')(\Theta_{\mathtt{fd}} \cdot \mathtt{L}_{\mathtt{fd}}), \Theta \cdot \mathtt{L})$.

```
01 Int fact_d(Int n){
02    Int y;
03    if (n == 0) return 1;
04    else { n = n-1; y = this.fact_d(n);
05           y = y*(n+1) ; return y; }}
```

$$\begin{aligned} \Theta_{\mathtt{fd}} &= \{f' \mapsto \lambda X.\ \mathtt{fact\_d}(\alpha, \square, X)\} \\ \mathtt{L}_{\mathtt{fd}} &= (f_\alpha + f'_\star + f'_\alpha)\ \&\ (X, \alpha) \\ \Theta &= \{f'' \mapsto \lambda X.\mathtt{fact\_d}(\alpha, \square, X)\} \\ \mathtt{L} &= f''_\star + f''_{main} \end{aligned}$$

The synchronization f'_α, contained in the behavior $\mathtt{L}_{\mathtt{fd}}$, causes a deadlock. The corresponding method invocation $(\lambda X.\ \mathtt{fact\_d}(\alpha, \square, X))$ is performed on the actor α, which amounts to instantiate the pair (X, α) into (α, α).

3 Future work.

Relaxing constraints. In order to simplify our arguments, we focussed on a sublanguage where futures are either returned or synchronized within a method body. This implies that a synchronization on a method will cause the simultaneous synchronization on every new future it may have directly or indirectly triggered. More specifically, after the synchronization we are guaranteed that every other method invocation triggered by it has terminated. We intend to relax this restriction by admitting method behaviours that trigger unsynchronized tasks. We already investigated this extension in [8] and the application to `gASP` of the solutions therein seems possible. A similar remark concerns the restriction that fields of actors must be ground integers. We can relax it by using records, as we did in [8]. In this case, the problematic issue will be to admit fields that store futures while keeping the precision of the analysis acceptable.

Actor model extension. A possible evolution of our work could be continue the study of deadlock analysis on some actor model extension. Generally an actor runs a single applicative thread, but in [10] a version of the model in which each actor is able to run more than one thread in parallel is presented. This extension both enhances efficiency on multicore machines, and prevents most of the deadlocks of the actors. It is trivial to see that if there are no constraints related to the number of methods that can run in parallel on the same actor, the model results to be deadlock free. However, it could be interesting to study this extension of the actor model enriched by a concept that can be defined as *compatibility between methods*. This compatibility can be a property defined by the programmer that through some kind of annotation can specify if it is *safe* to run in parallel some methods. In this context safe can mean that there are no data races condition if the compatible methods are executed in parallel on the same actor. We think that our technique can be applied on this scenario extending the type system in order to express the compatibility concept.

References

1. G. Agha. The structure and semantics of actor languages. In *REX Workshop*, pages 1–59, 1990.
2. G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
3. J. Armstrong. Erlang. *Communications of ACM*, 53(9):68–75, 2010.
4. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, 2009.
5. E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *Proceedings of IFM 2013*, volume 7940 of *LNCS*, pages 394–411. Springer, 2013.
6. E. Giachino, L. Henrio, C. Laneve, and V. Mastandrea. Actors may synchronize, safely! In *PPDP, Sep 2016, Edinburgh, United Kingdom. (to appear)*, 2016.
7. E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of CONCUR 2014*, volume 8704, pages 63–77.

8. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 2015.
9. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
10. L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In C. Julien and R. De Nicola, editors, *COORDINATION'13*, LNCS. Springer, June 2013.