Masini, J. and Francalanza, A. (2015). *Xjenza Online*, 3:51–55.

Xjenza Online - Journal of The Malta Chamber of Scientists
www.xjenza.org
DOI: 10.7423/XJENZA.2015.1.07

Research Article

Typing Actors using Behavioural Types

J. Masini, A. Francalanza
Department of Computer Science, University of Malta, Msida, Malta

Abstract. The actor model of computation assists and disciplines the development of concurrent programs by forcing the software engineer to reason about high-level concurrency abstractions. While this leads to a better handling of concurrency-related issues, the model itself does not exclude erratic program behaviours. In this paper we consider the actor model and investigate a type-based static analysis to identify actor systems which may behave erraticly during runtime. We consider the notion of behavioural types and consider issues related to the nature of the actor model including non-determinism, multi-party communication, dynamic actor spawning, non-finite computation and a possibly changing communication topology, which we contrast with existing works.

1 Introduction

The actor model (Hewitt, Bishop & Steiger, 1973) is becoming increasingly prevalent in the development of highly-concurrent systems and constitutes the underlying model of several mainstream technologies including programming languages such as Erlang (Armstrong, 2007; Cesarini & Thompson, 2009) and Scala (Odersky, Spoon & Venners, 2011), and frameworks such as AKKA ("AKKA", 2015) and Cloud-Haskell (Haskell, 2015). In particular, this programming model aides and disciplines the development of concurrent systems, facilitating the handling of concurrency-related programming issues *i.e.,* race conditions, dead/livelocks and starvation, whilst shielding from intricacies that can easily lead to errors (Haller & Sommers, 2012). Computation in the actor model is carried out by a number of *single-threaded* entities called *actors* executing *concurrently* within their own *local memory*. The absence of shared memory forces actors to interact solely by means of *asynchronous message-passing*. In Figure 1 we show the basic structure of an actor, composed of three distinct elements i) a *unique* actor name used (as an address) for communication; ii) an *expression* which describes the actor's behaviour; and iii) a *mailbox* that stores received messages in order (of time of arrival). Whereas each actor may be sent messages from several actors at the same time (*multi-participation*), input may only be done from the actor's own mailbox, unlike channel-based message passing (Haller & Sommers, 2012).

Figure 1: An actor.

The actor also have the ability to dynamically change the topology of their system by *spawning* additional actors during runtime. Communication topology may also be changed by the *delegation* of actor names in message parameters, allowing output to previously unknown actors. Actor computation may also be *non-terminating* in order to implement functionality such as that of servers or web-services. Due to the concurrent nature of the programming model, multi-participation and asynchrony, the messages received in an actor's mailbox may be different for each execution, causing *non-deterministic* actor behaviour. As a result, actor-model inspired technologies such as Erlang and AKKA extend input assisted with a *pattern-matching* mechanism (Armstrong, 2007; Cesarini & Thompson, 2009; Odersky et al., 2011), which allows messages to be retrieved from the mailbox in an order other than the one received in. We emphasise that our presentation of the actor model is derived from a high level description as described by Agha (1986), Clinger (1981) and instantiations of this model within programming languages such as Erlang (Armstrong, 2007; Cesarini & Thompson, 2009).

The rest of this paper highlights the actor model of computation (Section 2) followed by an analysis of the issues (and related existing works) to adapt behaviour types for actors, (Section 3). We conclude by analysing related work in behavioural types that concern actors specifically and close with some final remarks (Section 4).

Correspondence to: J. Masini (joseph.masini09@um.edu.mt)

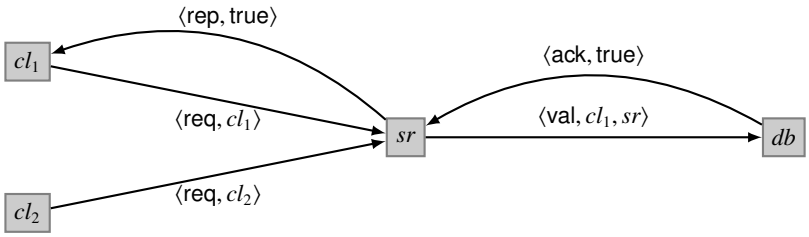

Figure 2: Banking system.

2 Actor Systems

Consider the actor system depicted in Figure 2 consisting of two clients, cl_1 and cl_2, which require credential validation from a server, *sr*, where the latter in turn requestions the assistance of a database service, *db*. Each actor is identified by its unique actor name, and output operations are represented as directed lines with messages as labels (and we abstract away from the actors' expression and mailbox structures).

In Figure 2, we show the validation process for one client; the client sends a request to the server in the form of a *tuple* tagged by the *label* req containing its own address for validation purposes, shown as ⟨req, cl_1⟩ (for $cl_1$and similarly ⟨req, cl_2⟩ for cl_2). Labels are employed to tag distinct messages for selective input from an actor's mailbox (through pattern-matching). Upon input, the server processes the client's request by extracting the client's address and creates a new message tagged with the label val containing the client's address and its own (as a reply address), ⟨val, cl_1, sr⟩. The server sends the latter message to the database service and awaits a reply from the server. After the database service inputs the server's request, validation is carried on the client's name and the result is sent to the requester's address (in this case the server) in the form of ⟨ack, true⟩ specifying the validation result. Once the server retrieves the message, it creates a new message, ⟨rep, true⟩ and sends it to the client currently being handled. It is important to note that the server is initially *unaware* of the client(s). Since it acts at the intermediary between it and the database service (which is aware of the client in order to verify its identity), when the server inputs this message it is temporarily made aware of the client by the address in its request in order to carry out the validation task with the database service and reply back to the client with the result. Yet the system specified in Figure 2 may run into problems; the server represents a servicing *bottleneck* since it serializes every request and temporarily halts servicing other clients.

Consider an improved arrangement of our banking system shown in Figure 3; instead of handling the task locally, the server assigns the service request to a new actor which it spawns acting as a task handler, *th*. This allows the server to remain responsive to other client requests by removing the computational load of redirecting messages back to the respective clients.

In this new arrangement we use the actor's ability to *spawn* additional actors during runtime and have the spawned task handle credential validation (of that particular client) with the database handler and subsequently complete the service request interaction with the client. This is achieved by the newly spawned task handler delegating its address instead of the server's in the request to the database service, ⟨val, *cl*, *th*⟩. It is important to note that the client is unaware that the service request was completed by the task handler instead of the server.

As both banking examples show, the actor model disciplines the implementation of a concurrent system; it forces the software engineer to avoid the mechanisms (and possible pitfalls) of shared memory by abstracting reasoning on the processing components as distinct entities executing within their own environment. In spite of this, the actor model does not guarantee the absence of erratic behaviour during the execution of such a system. In particular, the aforementioned systems are still susceptible to *execution errors i.e.,* system crash. Consider a client which sends a boolean value instead of its address for validation, as shown in Figure 4. When the database service attempts validation of the client credentials

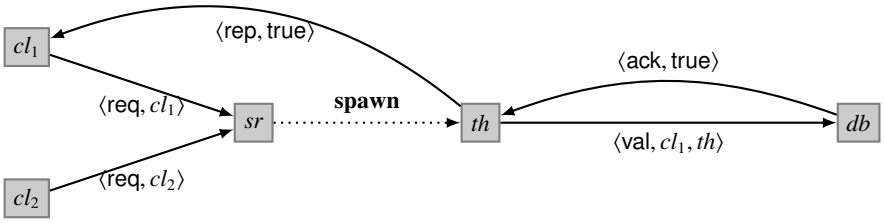

Figure 3: Improved bank system.

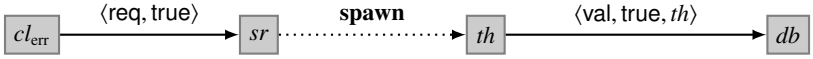

Figure 4: Erratic client 1.

(not shown in the figure), it will require an address but is provided with a boolean value, thereby crashing and halting the database service.

In Figure 5 we show another type of erratic behaviour where the database service would receive a genuine request, but never reply back. In such cases, the said system should be analysed against a form of *liveness* property to discard systems which *get stuck*.

We would like to formalise a static type system (Pierce, 2002; Cardelli, 2004) for the actor model, which would guarantee the absence of a select set of errors in actor systems. However, the development of a type-based static analysis for an actor system is non-trivial due to a number of reasons. In order to verify each actor's behaviour, we require to analyse the mailbox which essentially defines the actor's behaviour (Agha, 1986). The messages and the order in which they are received are the main variant on how an actor executes upon input *e.g.,* in our banking example, the server's next operation is dictated by the messages inputted from its mailbox, which in turn depends on the current state of its mailbox. However, since actor mailboxes may receive different types of messages, traditional type systems for concurrency are inapplicable, as these assume invariance with respect to the values communicated (Marlow & Wadler, 1997).

3 Typing Actors

Behavioural types (compared to traditional types), allow the analysis of *how* computation occurs by providing notions of causality and choice ("Behavioural Types for Reliable Large-Scale Software Systems - The Foundations of Behavioural Types: http://www.operationalsemantics.net/behaviouralwiki", n.d.). These have been extensively used to statistically analyse a wide range of computing aspects, ranging from ensuring party compatibility in protocols (Mostrous & Vasconcelos, 2011) to correct method call ordering in object-oriented systems (Gay, Vasconcelos, Ravara, Gesbert & Caldeira, 2010). Despite numerous existing works, proposed approaches are not directly applicable to analyse the actor systems. We aim to develop our own notation of behavioural types, whilst retaining the expressivity of the actor model. However, this is far from trivial as we require to consider several issues including concurrency and asynchrony, non-terminating behaviour, dynamic actor spawning and name delegation.

Concurrency and Asynchrony Concurrency is one main source of complication; due to potential non-determinism, we are forced to consider all possible execution interleavings amongst actors. Actor communication is also asynchronous, hence actors send messages irrespective of the execution state of the target actor. Due to this combination, actors that are sent messages by several other actors at one time (multi-participation) do not always guarantee the specific order of messages in the recipient's mailbox. Recall the server in Figure 3; client cl_2 can send a message to the server while it is handling a previous request from client cl_1. At the same time, the server may receive a reply for the database service for the request of cl_1. Figure 6 shows three instances of this server's state described by its name, expression and mailbox contents receptively. Figure 6 (*a*) describes the server's mailbox containing a request by the client, cl_2, and an acknowledgement sent back by the database service for the current client request being processed, cl_1. The server's mailbox may, from this state: i) increase in size with the reception of an additional message, Figure 6 (*b*), say from another client, cl_3; or ii) decrease in size with the server inputting from its mailbox, Figure 6 (*c*). In the latter case, since the server requires to process two different forms of messages (specifically client requests in the form of $\langle \mathsf{req}, \mathit{client\_name} \rangle$ and database service replies, $\langle \mathsf{ack}, \mathit{boolean} \rangle$) the server employs pattern-matching to extract messages other than the order present in the mailbox, which complicates our static analysis of the mailbox for each actor.

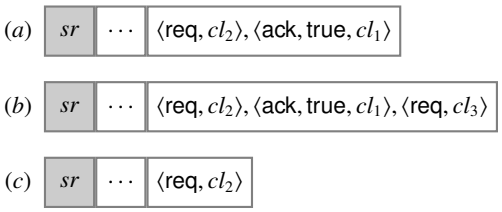

Figure 6: Mailbox communication structure.

Existing works, such as the work done by Honda, Yoshida and Carbone (2008), extend the notion of (binary) session types to a multi-party setting over a π-calculus with asynchronous communication semantics. In contrast, communication is carried by means of channels which may be used

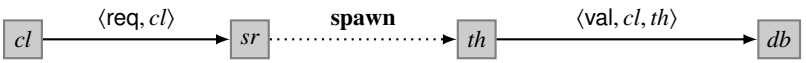

Figure 5: Erratic client 2.

for input and output by several entities but only up to two at one time, while in the actor model, each mailbox may only be inputted from by the mailbox owner and may be outputted by several entities at any time. Furthermore, input from a channel is on a first-in-first-out basis, unlike the pattern-matching input in an actor setting which allows input in a different order than from the one received, thereby having looser tolerances. They employ a notion of types inspired from the paradigm of *global programming* - interaction types are specified by a *global description* (or *global types*) of the overall communication behaviour between peers and are subsequently *projected* to extract the individual endpoint *local* sessions of each peer. However, this requires the knowledge of the communicating entities *a priori*, whereas in the actor model the topology may change during runtime by the dynamic spawning of actors. This is further aggravated by the possibility of delegation, as the communication topology may dynamically change to include actors unknown at runtime.

Non-terminating Behaviour We have to consider the ability of actor-inspired technologies to specify *non-finite* computation (Armstrong, 2007; Cesarini & Thompson, 2009; Odersky et al., 2011). Similar to the server and database service in our banking scenarios, actors can carry out a specific operational sequence for an unbounded number of times *e.g.,* the database service's operation sequence to receive a request for validation, process the credentials and send back the acknowledgement. We have to ensure safety for each possible operational sequence, and with the addition of non-determinism, we have to consider all possible interleavings for each operational sequence *e.g.,* it might be that in one execution, the request from one cl_1 is processed by the server first (and hence serviced by the database service), whereas in another execution cl_2 might be processed first. Several works have tackled the problem of recursive behaviour (or some form of repetition or replication of behaviour) (Honda et al., 2008; Caires & Vieira, 2010), however in our case we have to consider non-termination as well.

Dynamic Actor Spawning The actors' ability to spawn additional actors during runtime (Agha, 1986) is another issue that complicates the development of our static analysis. Our type system must support actor systems that may dynamically change the number of participants, similar to our bank system in Figure 3, whilst ensuring that these actors are still safe when interacting within the actor system. This is substantially more complex when it is coupled with possibly non-terminating computation. It requires assure safety in systems with (possibly) an unanticipated number of participants which may only be known at runtime *e.g.,* in our banking system, since we may not know *a priori* the number of clients that are going to interact with our system, we cannot know the number of actors spawned by the server. An extension of the work done by Honda et al. (2008) handles the issues of dynamic participation by the parametrization of sessions according to the number of participants (Deniélou, Yoshida, Bejleri & Hu, 2012). However, the abstractions required for the communication constructs are rather different from the ones in the actor model, as input and external choice are modelled separately and messages in channels retain the received ordering. Caires *et al.* present a novel notion of behavioural types referred to as conversation types (Caires & Vieira, 2010) to address multi-party interaction with dynamic interaction. The latter are similar to those found in service-oriented computing, which are conceptually analogous to the communication interactions in the actor model. The type system is based on an extension of the π-calculus that addresses the issue of non-deterministic communication by labelling output and input messages, similar to the communication primitives in our actor calculus. In contrast, the actor mailbox is more dynamic from the channels employed in the conversation calculus (Vieira, Caires & Seco, 2008), whereby our input operation allows (a pattern-matched) selection other than the first value found in the channel. The type system employed in (Caires & Vieira, 2010) uses a generalisation of session types adapted to multi-party interactions. This is achieved through the merging of local and global types which are used to compositionally distribute parts of the protocol between a number of participants, some of which may be unknown at runtime.

3.1 Name Delegation

The ability to *delegate* a name by sending a copy of the actor's address as a message parameter allows dynamic changing of the communication topology. While beneficial and allows communication to previously unknown actors, it decentralises control and introduces further process interleavings *e.g.,* the server in Figure 2 is previously unaware of the clients, until they send a request specifying their address. This is additionally complicated with the possibility of dynamic actor spawning, as we require to analyse possible communication paths between actors which are only known at runtime, *e.g.,* the task handler is spawned during runtime in Figure 3, and the database service is made aware of it from the task handler's request. Combined with non-terminating behaviour, this increases the complexity of the possible communication topology, as the number of participants and their interactions is unknown before runtime. As proposed by Caires and Vieira (2010), conversation types allow the type-checking of dynamic conversations, where a particular slice of the conversation is delegated to an other participant. Actors may also dynamically send addresses to enable interaction with possibly, previously unknown actors. Caires and Vieira (2010) enable dynamic participation by modelling multiparty conversation through name passing. This may be contrasted with the approach taken by Honda et al. (2008), where the authors distinguish between the passing of values and the passing of session to another entity at the level of the calculus. In our case, delegation is more subtle as

we communicate actor names as values, where some of the actor names may even be unknown before runtime.

4 Conclusion

There has been limited work in the area of behavioural types involving the actor model specifically (Crafa, 2012; Mostrous & Vasconcelos, 2011). Mostrous and Vasconcelos (2011) propose type system for a featherweight Erlang calculus, which lacks notions of internal choice and infinite computation. They analyse actor systems to eliminate possible actor impersonation which may cause malicious behaviour. However this is not our current goal as it enforces a specific programming approach thereby limiting actor (language) expressivity. Mostrous and Vasconcelos (2011) employs the notion of session types to model the protocol of specific sequences and forms of messages. This is used to ensure that an actor handles all the messages in its mailbox, and receives all expected messages. However, this goes against our notion of the actor model as i) actors are not hindered by extra messages (virtually) inside their mailbox; and ii) the absence of a message in the mailbox does not constitute ill-behaviour, especially when we are reasoning on non-finite computation. In fact, an actor that does not receive a particular message simply blocks waiting for the desired input, a mechanism intrinsic to the actor model. Crafa (2012) proposes a work closer to what we study, specified over an alternate actor calculus inspired by Odersky et al. (2011). Their work lacks the notion of internal choice and constructs to express some form of repetitive behaviour. Furthermore, we contrast her use of input semantics with (Crafa, 2012), where the author handles communication non-determinism by specifying non-deterministic semantics for actor input, which does not reflect the input specification of the actor model. Also, Crafa (2012) employs an approach inspired by conversation types where the behavioural types define the protocols as a sequence of messages, branching from external choice. In order for typechecking to occur, a path is marked on each protocol describing each actor and the expected messages, and the path markings are distributed amongst each actor compositionally. This approach allows Crafa to handle dynamic actor spawning and ensure adherence of each actor's protocol throughout the entire actor system.

We give an in-depth study of the issues involved in the adaptation of behavioural types to model actor systems. We also contrasted them against the current literature for actors, however we found these to be rather restrictive. We concluded that current type systems are not flexible enough to allow analysis of the actor model and we consider it as a promising area for further research and study.

References

Agha, G. (1986). An Overview of Actor Languages. In *Proc. 1986 sigplan work. object-oriented program.* (pp. 58–67). OOPWORK '86. New York, NY, USA: ACM.

AKKA. (2015). Retrieved from http://www.akka.io

Armstrong, J. (2007). *Programming Erlang - Software for a Concurrent World*. The Pragmatic Bookshelf.

Behavioural Types for Reliable Large-Scale Software Systems - The Foundations of Behavioural Types: http://www.operationalsemantics.net/behaviouralwiki. (nodate).

Caires, L. & Vieira, H. T. (2010). Conversation Types. *Theor. Comput. Sci. 411*(51-52), 4399–4440.

Cardelli, L. (2004). Type Systems. In *Comput. sci. eng. handb.* CRC Press.

Cesarini, F. & Thompson, S. (2009). *ERLANG Programming* (Media, Inc). O'Reilly.

Clinger, W. D. (1981). *Foundations of Actor Semantics*. Cambridge, MA, USA: Massachusetts Institute of Technology.

Crafa, S. (2012). Behavioural Types for Actor Systems. *CoRR*, *abs/1206.1*.

Deniélou, P.-M., Yoshida, N., Bejleri, A. & Hu, R. (2012). Parameterised Multiparty Session Types. *Log. Methods Comput. Sci. 8*(4).

Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N. & Caldeira, A. Z. (2010). Modular Session Types for Distributed Object-oriented Programming. *SIGPLAN Not. 45*(1), 299–312.

Haller, P. & Sommers, F. (2012). *Actors in Scala*. USA: Artima Incorporation.

Haskell, C. (2015). Cloud Haskell website: retrieved from http : / / www . haskell . org / haskellwiki / Cloud % 5C_Haskell

Hewitt, C., Bishop, P. & Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Ijcai* (pp. 235–245). Morgan Kaufmann.

Honda, K., Yoshida, N. & Carbone, M. (2008). Multiparty Asynchronous Session Types. *SIGPLAN Not. 43*(1), 273–284.

Marlow, S. & Wadler, P. (1997). A Practical Subtyping System For Erlang. In *Proc. int. conf. funct. program. (icfp '97* (pp. 136–149). ACM Press.

Mostrous, D. & Vasconcelos, V. T. (2011). Session Typing for a Featherweight Erlang. In *Proc. 13th int. conf. coord. model. lang.* (pp. 95–109). COORDINATION'11. Berlin, Heidelberg: Springer-Verlag.

Odersky, M., Spoon, L. & Venners, B. (2011). *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition* (2nd). USA: Artima Incorporation.

Pierce, B. C. (2002). *Types and Programming Languages*. Cambridge, MA, USA: MIT Press.

Vieira, H. T., Caires, L. & Seco, J. (2008). The Conversation Calculus: A Model of Service-Oriented Computation. In S. Drossopoulou (Ed.), *Program. lang. syst.* (Vol. 4960, pp. 269–283). Lecture Notes in Computer Science. Springer Berlin Heidelberg.