Composable Actor Behaviour

Sam van den Vonder, Joeri de Koster, Wolfgang de Meuter

Composable Actor Behaviour

Sam Van den Vonder, Joeri De Koster, and Wolfgang De Meuter

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
{svdvonde,jdekoste,wdmeuter}@vub.be

Abstract Code reusability is the cornerstone of object-oriented programming. Reuse mechanisms such as inheritance and trait composition lay at the basis of a whole range of software engineering practices with the goal to improve software quality and reliability. In this paper we investigate code reuse mechanisms for actors, and find that it is currently difficult to specify the behaviour of an actor out of reusable parts. We discuss different kinds of code reuse mechanisms in different kinds of actor model, and we motivate why these mechanisms are currently unsatisfactory. As a possible solution we define a new reuse mechanism based on delegation-based trait composition. In a nutshell, the mechanism allows programmers to compose the behaviour of actors, and every time a compound behaviour is spawned into an actor, it will cause multiple actors to be spawned (one for each independent behaviour). Some messages will be automatically delegated to the actor that implements the desired functionality. We provide an implementation of this model in a prototype Active Object language called Stella, and we formalise a subset of Stella using a small-step operational semantics to unambiguously define the different steps involved in our reuse mechanism.

Keywords: actors · delegation · active objects · code reusability

1 Introduction

In object-oriented programming, the principle of "programming against an interface" helps to foster code reuse and reduce complexity, thus increasing the reliability of individual components [20]. Essentially it is beneficial for the overall complexity of the program to design components as black boxes, because it is then the sole responsibility of each individual component to ensure the functionality it offers through its interface is correct. This principle manifests itself in many reuse mechanisms, such as inheritance [15] and trait composition [12] for class-based languages, and delegation for prototype-based languages [19]. In actor-based programs, using components as black boxes is equally important but for reasons other than just modularity and code reuse.

The *behaviour* of an actor is usually a combination of its internal state and its interface, which is the set of messages that an actor can process [18]. The only way to communicate with an actor is to send it a message that matches an entry in its interface, which is important for two reasons. First, it makes it easier for actors to protect their internal state from race conditions via *interface control* [17] (essentially by asynchronously processing messages one by one).

Second, a message-passing communication model is beneficial for (among others) concurrency, fault tolerance, and distribution over a network.

Despite the principle of "programming against an interface" being ingrained in the actor model almost by definition, it is rarely leveraged to facilitate modularisation and code reuse among actors. More specifically, currently there is limited language support for composing the behaviour of an actor, i.e. its interface, out of reusable parts.

Since code reuse is an important aspect of software engineering, we argue that actor-based programs can benefit from a simple and well-defined code reuse mechanism to control and reduce their complexity. To this end we introduce Stella, a prototype language that implements an actor composition mechanism based on asynchronous message delegation. The main contributions of this paper are the design and definition of Stella, and a formalisation of a subset of Stella that captures the precise semantics of the reuse mechanism.

In Section 2 we discuss the requirements of a suitable code reuse mechanism for actors, and we discuss reuse mechanisms in a number of state-of-the-art actor languages. In Section 3 we define Stella, and in Section 4 we define an operational semantics for a subset of Stella.

2 Code Reuse in Actor-Based Languages

Before we look into existing reuse mechanisms, we define the term actor *behaviour* and we specify the requirements of a reuse mechanism suitable for actors. We adopt the terminology of [18] that defines the behaviour of an actor as the description of its interface and state. The *interface* defines the list (and possibly types) of messages that an actor can process, as well as the program logic to process them. The state of an actor is defined as all the program state that is synchronously accessible by the actor.

From a software engineering point of view it is beneficial to split up a monolithic behaviour into multiple "reusable components" that can be composed using a composition mechanism. We devise 2 goals or requirements for such a mechanism that is suitable for the actor model, which we base on well-established design principles for object-oriented programming [24, Chapter 14].

Extensibility. The interface of behaviours can be extended to accept new messages, and a behaviour can specialise the acquired components to adapt them to its own needs. Relating this to object-oriented programming, this is similar to how a class can add, override, and specialise methods of a superclass, or to how traits can add methods to a class which may also be further specialised.

Reusability. Pre-existing behaviours can be reused by extending, adapting or specialising them via new behaviour definitions, without modifying the original behaviour. In object-oriented programming this is similar to how a class may be specialised via subclassing, while it remains independently instantiatable regardless of new class definitions that rely on it.

Over the years a number of reuse mechanisms have been proposed in different kinds of actor languages that implement different kinds of actor model. In the following sections we discuss inheritance, trait composition, function composition, and reuse via prototypes in the Communicating Event-Loops model.

2.1 Inheritance

The relation between inheritance and concurrent object-oriented programming has been thoroughly researched. Part of this research is focussed specifically on inheritance in actor-based languages such as Act3 [4] and ACT++ [16], which are based on the work of Agha [3]. In these languages, a `become` statement (fundamental to the model) is used to replace the behaviour of the current actor with a new behaviour. This statement causes severe reusability issues due to the *Actor-Inheritance Conflict* [17]. Consider a behaviour as being similar to a class, then the conflict describes a fundamental issue where adding a new method to a subclass may invalidate many superclass methods.

Classes in combination with inheritance fulfil the requirements of extensibility and reusability. However, inheritance is known to have reusability issues when used as the sole reuse mechanism [12]. Furthermore, nowadays it is generally accepted as a good design principe in object-oriented programming to favour object composition over class inheritance [7, 14]. For these reasons we do not consider inheritance by itself to be a suitable reuse mechanism for actors.

2.2 Trait Composition

The Active Objects model is based on the work on ABCL/1 [27], and has modern implementations in languages such as Pony [9] and Encore [8]. Here, actors are typically instances of an *active object class* which describes mutable fields and a set of methods that can be asynchronously invoked via message passing. Pony and Encore support neither a `become` statement (which caused the Actor-Inheritance Conflict from the previous section) nor reuse via inheritance. In the case of Pony it is mentioned that composition is preferred to inheritance [1].

Instead of inheritance, Pony and Encore support stateless traits [12]. Traits can be composed with behaviours and other traits to add new methods to the composer. However, they do not fulfil our 2 requirements. Extensibility is only partially fulfilled because, while traits can be used to extend a behaviour with new functionality, they have a number of drawbacks. Most notably, stateless traits are likely to be an incomplete implementation of some functionality unless it is completely stateless [6]. The follow-up work on stateful traits also has some drawbacks such as possibly breaking black-box encapsulation, and difficulties regarding a linear object layout in memory [6]. Reusability is unfulfilled because the trait composition mechanism cannot be used to compose behaviours.

2.3 Function Composition

Popular languages and libraries such as Erlang [5], Elixir [26] and Akka [23] closely link behaviours and functional programming. In Erlang and Elixir, a

blocking `receive` statement is used as part of a function body to dequeue 1 message from the mailbox of the current actor, and local actor state is encapsulated via lexical scoping. In Akka (a library for Scala), the behaviour of an actor is represented as a Scala *partial function* that is continually applied to messages by the receiving actor. Consequentially, behaviour composition is based on function composition. For example, in Akka, the Scala `andThen` and `orElse` function combinators compose two behaviours by respectively chaining two functions (pass the output of 1 into the next) or switching over 2 functions (if the given argument is not in the domain of the first, try the second).

We do not consider function composition to be a suitable reuse mechanism because it does not support extensibility. Logically switching over behaviours can be used to emulate some features of extensibility, e.g. the behaviour that is the result of the composition `(behaviourA orElse behaviourB)` will accept the union of messages accepted by both behaviours. However, the end result is highly susceptible to the composition order; messages matched by both behaviours will always be processed exclusively by `behaviourA`. Furthermore, there is no mechanism to deal with conflict resolution, for example when `behaviourA` accidentally captures messages that should be processed by `behaviourB`.

2.4 Communicating Event-Loops

The Communicating Event-Loops model (CEL) originated in the E [22] language and was later adopted by AmbientTalk [11]. Here, an actor is not described by a behaviour. Instead, an actor is a *vat* of plain objects that are said to be owned by the actor. Objects owned by one actor can have *near references* to objects within the same vat and *far references* to objects in another vat. Method calls via a near reference are synchronous; method calls via a far reference are asynchronous, are sent to the actor that owns the object, which will eventually invoke the method. In this model, the behaviour of an actor depends on which of its objects are accessible via far references, since those determine which messages are accepted.

Both E and AmbientTalk define a prototype-based object model, which relies on functions and lexical scoping for object instantiation and information hiding. A problem occurs when two similar actors attempt to share a behaviour, which in this model amounts to sharing an object. If two actors could reference the same behaviour, they would have access to shared mutable state either via the shared lexical scope, or via the shared object. Therefore, a CEL model in combination with a prototype-based object model does not offer a suitable reuse mechanism because, idiomatically, behaviours cannot be freely reused by different actors.

A possible avenue to explore could be to design a class-based CEL language which can eliminate shared mutable state. While we consider this to be a viable approach to our problem, in this paper we opt for a different approach that we consider to be simpler and more applicable to other actor languages.

2.5 Problem Statement

In the previous sections we discussed different kinds of reuse mechanisms in different kinds of actor model. In Section 2.1 we discussed the relationship between inheritance and actors, and concluded that inheritance by itself is not a suitable reuse mechanism for actors. For Active Objects (Section 2.2) we discussed trait composition and how it does not fulfil our requirements, because traits have a number of drawbacks and cannot be used to compose behaviours themselves. We discussed actor languages where behaviours are encapsulated by functions (Section 2.3), where we motivated that function composition is not a suitable composition mechanism. Finally, for the Communicating Event-Loops model (Section 2.4) we discussed that a prototype-based object model would lead to shared mutable state between actors if behaviours could be reused.

The problem that we tackle in this paper is to define a code reuse mechanism for behaviours that fulfils the requirements of extensibility and reusability. The mechanism defines (1) how the interface of behaviours can be extended with functionality defined by different components (extensibility), and (2) how behaviours themselves can be reused to define new behaviours (reusability).

3 Delegation-based Actor Composition in Stella

In this section we introduce Stella, a prototype language based on the Active Objects model, where behaviours can be composed with a mechanism based on delegation-based trait composition [10]. We opted for a language-based approach (rather than a library) to convey the mechanism in a clear and concise manner. It also ensures consistent run-time semantics, particularly with respect to the definition of behaviours and message sending between actors. We first give a motivating example that benefits from reusable behaviours, and in the sections thereafter we gradually introduce the different aspects of Stella. For brevity we only implement parts of the motivating example to introduce the base language and to explain behaviour composition (the precise semantics of which are formalised in Section 4)[1].

3.1 Motivating Example

A modern approach to building "real-time" or "live" applications are stream-based frameworks such as ReactiveX, which describes the API of a class of streaming frameworks in over 18 languages [2]. These frameworks provide abstractions for data streams together with an extensive collection of built-in operators to transform and combine them. Consider a temperature monitoring application that visualises live measurements of many heterogenous sensors. Depending on units of measurement and user preferences, measurements may have to be transformed from one unit to another. This can be done by mapping a

[1] The code for the complete example is available and can be run at http://soft.vub.ac.be/~svdvonde/examples/DAIS19/.

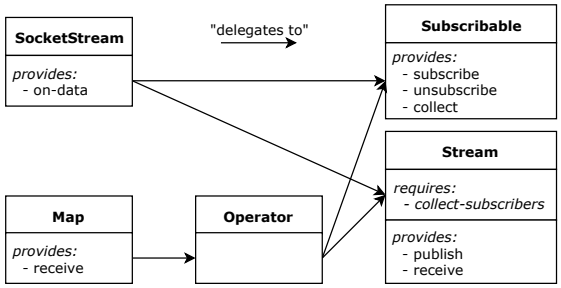

Figure 1: Composition of behaviours in an actor-based streaming framework.

conversion function over a stream of measurements using some built-in `map` operator, resulting in a new stream of data.

Streaming frameworks are often designed sequentially, i.e. new input data is first propagated to all connected streams before the next input can be accepted, and parallelising this process is non-trivial. With composable behaviours we can design a simple framework where streams and operators are actors, such that multiple computations can run in parallel.

Figure 1 depicts the different behaviours involved in our streaming framework. Every behaviour lists the methods that it provides, and for clarity we also list when a behaviour expects a certain method to be present in its composer that it does not implement itself. The framework provides a `SocketStream` behaviour to abstract over a typical socket connection as a data stream, and also 1 built-in `Map` operator to map a function over a stream. Common functionality for operators is factored out into an `Operator` behaviour (which also behaves like a stream), and common functionality of streams is factored out into `Stream` and `Subscribable`. `Stream` implements functionality for publishing and receiving values, `Subscribable` simply keeps a list of other streams (actors) that should receive new publications.

3.2 The Base Stella Language

In this section we introduce the base Stella language without behaviour composition. Similar to other Active Object languages, Stella has an active layer of active object classes and actors, and a passive layer of regular classes and objects We omit the details of the passive object layer since its definition is irrelevant to the problem of behaviour composition.

A program written in Stella is a set of top-level behaviour definitions and regular class definitions. Every program must define a `Main` behaviour that is instantiated as the first actor of the program. Listing 1 implements two behaviours `Stream` (Lines 1-5) and `Subscribable` (Lines 7-14). `Stream` implements generic stream functionality for publishing and receiving data. It has two methods called `publish` and `receive` with 1 formal parameter `data` (Lines 2-4 and 5). Publishing data to a stream simply amounts to sending a `receive` message to all subscribers. The logic of sending that message is contained within local definition

```
1  (actor Stream
2    (def-method (publish data)
3      (def f (lambda (subscriber) (send subscriber 'receive data)))
4      (send self 'collect-subscribers f))
5    (def-method (receive data) 'do-nothing))
6
7  (actor Subscribable
8    (fields subscribers)
9
10   (def-constructor (init) (set! subscribers '()))
11
12   (def-method (subscribe subscriber)
13     (set! subscribers (add subscribers subscriber)))
14   (def-method (collect f) (for-each subscribers f)))
```

Listing 1: Implementation of the `Stream` and `Subscribable` behaviours.

`f` (Line 3) that is bound to a `lambda` function[2]. When the lambda is invoked, it sends the `receive` message to `subscriber` with the `data` to be published as single argument. Iterating over subscribers of the stream is done by sending a `collect-subscribers` message to the current actor via pseudo-variable `self` with `f` as argument. The default `receive` method on Line 5 simply returns the symbol `'do-nothing`.

The `Subscribable` behaviour stores a list of subscribers to a stream. Its definition is analogous to `Stream` but shows the use of fields (local actor state) and constructors. In this case there is 1 field `subscribers` (Line 8), a constructor named `init` (Line 10), and 2 methods `subscribe` and `collect` (Lines 12-13 and 14). A constructor is a special method that is called exactly once when an actor is spawned. Behaviours without a constructor will be initialized by a `default` constructor. In this case the `init` constructor initializes the local field `subscribers` to an empty list via the special form `set!` (assignment).

Bodies of constructors and methods contain either special forms (like `set!`) or synchronous method invocations on regular objects. Here, we use the following syntax where `methodName` is invoked on object `target` with the given argument expressions.

$$(methodName\ \ target\ \ arg_1\ \ \ldots\ \ arg_n)$$

In that vein, the invocation of `send` in `Stream` (Listing 1 Line 3 and 4) is simply the invocation of the `send` method on an object that represents a reference to an actor. Similarly, the `add` and `for-each` methods (Line 13 and 14) are invocations on a list object.

Actors can be spawned via a `spawn` special form that returns a reference object that can be used to send asynchronous messages to the newly spawned actor. For example, the following expression spawns an actor with the `Subscribable` behaviour that is initialized by calling the `init` constructor.

```
(spawn Subscribable 'init)
```

[2] Stella does not have functions. Using a process similar to Lambda Lifting, a `lambda` statement is transformed to an object with an `apply` method

```
1  (actor Operator
2    (delegate-to Stream)
3    (delegate-to Subscribable (rename 'collect 'collect-subscribers))
4
5    (def-constructor (init stream)
6      (spawn-delegate Subscribable 'init)
7      (spawn-delegate Stream)
8      (send stream 'subscribe self)))
```

Listing 2: Implementation of the `Operator` behaviour.

```
1  (actor Operator
2    (delegate-fields Subscribable Stream)  // run-time syntax
3
4    (def-constructor (init stream)
5      (spawn-delegate Subscribable 'init)  // populate special field
6      (spawn-delegate Stream)              // populate special field
7      (send stream 'subscribe self))
8
9    (def-method (subscribe subscriber)
10     (delegate Subscribable 'subscribe subscriber))
11   (def-method (collect-subscribers f)     // renamed method
12     (delegate Subscribable 'collect f))
13
14   (def-method (publish data) (delegate Stream 'publish data))
15   (def-method (receive data) (delegate Stream 'receive data)))
```

Listing 3: Compile-time expanded version of the `Operator` behaviour.

3.3 Delegation-based Behaviour Composition in Stella

In this section we introduce a new composition mechanism for behaviours inspired by delegation-based trait composition in AmbientTalk [10]. In a nutshell, a behaviour can statically acquire the methods of other behaviours, and spawning an actor from a compound behaviour creates multiple actors that each run part of the compound behaviour. We will refer to these actors as the *delegate actors*. Messages that match an acquired method are automatically delegated to the corresponding delegate actor. To explain the different aspects of behaviour composition, we implement the `Operator` behaviour from Figure 1.

The `Operator` behaviour implements common functionality for all streaming operators in our motivating example, which in this case only amounts to ensuring that every instance of an operator behaves like a stream of data. Its implementation is shown in Listing 2. A `delegate-to` statement (Line 2-3) is used to declare that (at compile-time) all methods from the behaviours `Stream` and `Subscribable` are acquired. A conflict may occur when acquiring two or more methods with the same name. These must be explicitly resolved by aliasing or excluding certain methods using a `rename` or `exclude` statement respectively. In this case, the `collect` method from `Subscribable` is renamed to `collect-subscribers` for clarity rather than solving a conflict.

Before we can explain the run-time semantics of acquired methods (which is different from traditional trait composition), we first show the effects of a `delegate-to` statement at compile-time. Listing 3 shows the compile-time expanded version of the `Operator` behaviour of Listing 2, which incorporates the acquired methods. The added lines of code are Line 2 – a pseudocode statement

to explain the run-time semantics – that declares 2 new (special) fields, Lines 9-12 which are the acquired methods from the `Subscribable` behaviour (note that the `collect` method is renamed), and finally Lines 14-15 which are the acquired methods from the `Stream` behaviour.

The 2 new fields on Line 2 are generated by the compiler and carry the name of the delegate behaviours. They are populated by the `spawn-delegate` statements in the constructor (Lines 5-6), which is a special version of a regular `spawn`. Instead of returning the address of the new actor, it is stored in the corresponding (generated) field that carries the name of the spawned behaviour. Thus, when `Operator` is spawned, it also spawns 2 delegate actors, and by storing their addresses in generated fields we can guarantee that the contents of these fields cannot be directly modified or retrieved. Consequentially, because the address of delegate actors can never be shared with other actors, we keep the process of spawning delegates completely transparent to users of a behaviour.

In contrast with regular trait composition for object-oriented programming, the implementation of acquired methods is not copied over. Instead, a `delegate` statement is generated that serves 2 purposes. First, `delegate` retrieves the delegate actor (from the generated fields) referenced by its first argument. Second, it sends a special message to the delegate actor that, when the message is executed, changes the `self` pointer of the delegate actor to that of the sender. This is a crucial mechanism of trait composition that allows the delegate to communicate with its delegator, which is similar to the unchanged `this` pointer for regular trait composition in object-oriented programming [10,12]. The effect is that, any time the delegate actor sends a message to `self`, the message is actually received by its delegator. An example of where this mechanism is necessary is the `Stream` behaviour of Listing 1 Line 4, where a `collect-subscribers` message is sent to `self` to iterate over a list of subscribers stored in another behaviour.

4 Operational Semantics of Stella

In this section we formalise a subset of Stella via an operational semantics. The formalisation entails the necessary details about actors, behaviours and delegation. For brevity we omit the sequential class-based object-oriented subset of the language, since this concern is orthogonal to actors and behaviours. The goal of this formalisation is to describe the precise semantics of the composition mechanism such that it can be reproduced in other languages. Our semantics is based on the formalisation of JCoBox [25] and AmbientTalk [11].

4.1 Syntax

The abstract syntax of Stella is shown in Figure 2. Capital letters denote sets, and overlines denote sequences (ordered sets). We may implicitly treat single elements as sequences or sets of size 1 (e.g. $\mathcal{A}(\ldots)$ is equivalent to $\{\mathcal{A}(\ldots)\}$). Most of the syntax is shown in Section 3. Note that in this section we talk about (active object) classes instead of behaviours.

$$
\begin{array}{rcl}
p \in \textbf{Program} & ::= & B \\
B \subseteq \textbf{ClassDecl} & ::= & (\text{actor } n\ D\ (\text{fields } \overline{f})\ H) \\
D \subseteq \textbf{DelegationDecl} & ::= & (\text{delegate-to } n\ (\text{exclude } \overline{m})\ \overline{(\text{rename } m\ m')}) \\
H \subseteq \textbf{MethodDecl} & ::= & (\text{def-method } (m\ \overline{x})\ e) \\
e \in E \subseteq \textbf{Expression} & ::= & \begin{array}{l} x \mid self \mid (\text{get } f) \mid (\text{set! } f\ e) \mid (\text{spawn } n) \mid \\ (\text{delegate } d\ m\ \overline{e}) \mid (\text{send } e\ m\ \overline{e}) \end{array}
\end{array}
$$

$$x \in \textbf{Variable}, f, d \in \textbf{FieldName}, n \cup Main \in \textbf{ClassName}, m \in \textbf{MethodName}$$

Figure 2: Abstract syntax.

$$
\begin{array}{rcll}
k \in K \in \textbf{Configuration} & ::= & \mathcal{K}(A, C) & \text{Configurations} \\
C \subseteq \textbf{ActorClass} & ::= & \mathcal{C}(n, \overline{f}, \overline{d}, M) & \text{Actor Classes} \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}(i, M, Q, F, F_d, e) & \text{Actors} \\
Q \subseteq \textbf{Queue} & ::= & \overline{msg} & \text{Queues} \\
F, F_d \subseteq \textbf{Field} & ::= & \mathcal{F}(f, v) & \text{Fields} \\
msg \in \textbf{Message} & ::= & \mathcal{M}sg(i, m, \overline{v}) & \text{Messages} \\
M \subseteq \textbf{Method} & ::= & \mathcal{M}(m, \overline{x}, e) & \text{Methods} \\
v \in \textbf{Value} & ::= & i \mid null & \text{Values} \\
e \in E \subseteq \textbf{Expression} & ::= & \ldots \mid v & \text{Runtime Expressions}
\end{array}
$$

$$i \in \textbf{ActorId}$$

Figure 3: Semantic entities.

- A program `p` is a set actor class declarations, one of which we assume will be called `Main`.
- For simplicity, classes have no constructor.
- Because there are no constructors, there is no explicit `spawn-delegate` statement required in the syntax because delegate actors can now be created ex nihilo (i.e. without initializing them with run-time values).
- Methods have just one expression as their body, and there are no variable definitions (e.g. via a `let` statement).
- Fields are accessed explicitly via `get` and `set!` statements.

4.2 Semantic Entities

The static and dynamic semantics are formulated as a small-step operational semantics whose semantic entities are listed in Figure 3. Calligraphic letters such as $\mathcal{K}$ and $\mathcal{C}$ are used as "constructors" to distinguish different semantic entities syntactically.

The state of a program is represented by a configuration `k` which contains a set of concurrently executing actors and a set of classes.

A class has a unique name `n`, fields $\overline{f}$, delegate fields $\overline{d}$ (these are the generated fields to store references to delegate actors, see Section 3.3), and a set of methods M. In Section 4.4 we show how a class is produced from the abstract syntax.

An actor has a unique identifier i that we use as its address, a set of methods M that can be invoked by the actor, a queue Q that holds a sequence of messages

to be processed, a set F that maps fields to values, a set F_d that maps delegate fields to delegate actors, and an expression e that the actor is currently executing.

A message msg is a triplet of a `self` address i to be used during execution of the message (either the message receiver or the delegator), the name m of a method to invoke, and a sequence of values $\overline{v}$ which are the method arguments.

A method M is a triplet containing the name of the method m, a sequence of formal parameters $\overline{x}$, and a body e.

Our reduction rules in Section 4.5 operate on "runtime expressions" which are simply all expressions e extended with run-time values v, which can be actor references i and $null$.

4.3 Notation

We use the $\uplus$ (disjoint union) operator to lookup and extract values from sets. For example, $S = S' \uplus s$ splits the set S into element s and the set $S' = S \setminus \{s\}$. When the order of elements is important (e.g. for representing the message queue of an actor) we use the notation $S = S' \cdot s$ to deconstruct a sequence S into sequence $S' = S \setminus \{s\}$ and s which is the last element of S. We denote both the empty set and the empty sequence as $\emptyset$.

4.4 Static Semantics

Our reuse mechanism requires an additional compilation step to transform a class declaration from the abstract syntax into a class that can be used at run-time. In Figure 4 we define a number of auxiliary functions in a declarative style to generate such a run-time class. We sum up their purpose:

gen Generates a set of methods (to be acquired by a class) based on a set of pre-existing methods and a set of `delegate-to` statements. M represents a set of pre-existing (non-acquired) methods of a class, C is a set of compiled run-time classes, and D a set of delegation declarations. For each delegation declaration, lookup the corresponding run-time class and generate a set of methods to be acquired for this particular delegate.

genMethods Given a set of pre-existing methods M, a classname n, a set of excluded methods m_{excl}, a set of aliased methods m_{alias} and a set of methods M' to acquire, return a new set of methods possibly extended with newly acquired ones. Methods in M' with name m are excluded if $m \in \overline{m_{excl}}$, or if a method m already exists in the pre-existing set of methods M. The latter ensures that methods from the base class take precedence over acquired methods (they are not "overridden" by the delegate).

genMethod Generate the method to be acquired by a class. $\mathcal{M}$ is the original method from the delegate class, n is the name of said class, and m_{alias} is a set of tuples to possibly rename the generated method. The body of the generated method is a `delegate` expression where n will refer to the delegate actor.

name Given a method name m and a set of possibly aliased methods m_{alias}, return the (possibly aliased) method name.

$$\text{gen}(M, C, \emptyset) = \emptyset$$
$$\text{gen}(M, C, D \cup (\text{delegate-to } n \ (\text{exclude } \overline{m_{excl}}) \ \overline{m_{alias}})) =$$
$$\quad \text{genMethods}(M, n, \overline{m_{excl}}, \overline{m_{alias}}, M') \cup \text{gen}(M, C, D)$$
$$\quad \text{with} \quad \mathcal{C}(n, \overline{f}, \overline{d}, M') \in C$$

$$\text{genMethods}(M, n, \overline{m_{excl}}, \overline{m_{alias}}, \emptyset) = \emptyset$$
$$\text{genMethods}(M, n, \overline{m_{excl}}, \overline{m_{alias}}, M' \cup \mathcal{M}(m, \overline{x}, e)) =$$
$$\begin{cases} \text{genMethods}(M, n, \overline{m_{excl}}, \overline{m_{alias}}, M'), & \text{if } m \in \overline{m_{excl}} \ \vee \ \mathcal{M}(m, \overline{x'}, e') \in M \\ \text{genMethod}(\mathcal{M}(m, \overline{x}, e), n, \overline{m_{alias}}) \ \cup \ \hookleftarrow & \\ \quad \text{genMethods}(M, n, \overline{m_{excl}}, \overline{m_{alias}}, M') & \text{otherwise} \end{cases}$$

$$\text{genMethod}(\mathcal{M}(m, \overline{x}, e), n, \overline{m_{alias}}) = \mathcal{M}(\text{name}(m, \overline{m_{alias}}), \overline{x}, (\text{delegate } n \ m \ \overline{x}))$$

$$\text{name}(m, \overline{m_{alias}}) = \begin{cases} m', & \text{if } (\text{rename } m \ m') \in \overline{m_{alias}} \\ m & \text{otherwise} \end{cases}$$

$$\text{methodsOf}((\text{delegate-to } n \ (\text{exclude } \overline{m_{excl}}) \ \overline{m_{alias}}), C) =$$
$$\quad \{name(m, m_{alias}) \mid (\mathcal{M}(m, \overline{x}, e) \in M) \wedge (m \notin m_{excl})\}$$
$$\quad \text{with} \quad \mathcal{C}(n, \overline{f}, \overline{d}, M) \in C$$

Figure 4: Auxiliary functions for class compilation.

$$\frac{\begin{array}{c} \forall x_1 \in D : \forall x_2 \in D \setminus \{x_1\} : methodsOf(x_1, C) \ \cap \ methodsOf(x_2, C) = \emptyset \\ M = \{\mathcal{M}(m, \overline{x}, e) \mid (\text{def-method } (m \ \overline{x}) \ e) \in H\} \\ \overline{d} = \{n \mid (\text{delegate-to } n \ (\text{exclude } \overline{m}) \ \overline{(\text{rename } m \ m')}) \in D\} \end{array}}{\langle B \cup (\text{actor } n \ D \ (\text{fields } \overline{f}) \ H), C\rangle \rightarrow_c \langle B, C \cup \{\mathcal{C}(n, \overline{f}, \overline{d}, M \cup \text{gen}(M, C, D))\}\rangle}$$

Figure 5: Class compilation reduction rule.

methodsOf Given a delegation declaration and a set of run-time classes C, return the set of all method names that would be acquired by C using the delegation declaration.

Figure 5 defines a reduction $\rightarrow_c$ to compile a set of class declarations B. The reduction is defined as $\langle B, C\rangle \rightarrow_c \langle B', C'\rangle$ where the tuple $\langle B, C\rangle$ initially contains all class declarations B in the program, and C is empty. Compilation fails when the conditions of the rule are not met and no element in B can be reduced. This signifies an error in the program. Another possible error is explicitly formulated by a precondition given on the first line of the premise that prevents method conflicts between delegates, which means that the intersection of the acquired methods for any 2 delegates is empty. A set of delegate fields $\overline{d}$ is created using the classnames of delegates.

$$\text{Field-Get}\ \frac{\mathcal{F}(f, v) \in F}{\mathcal{A}(i, M, Q, F, F_d, e_\Box[(\text{get } f)]) \rightarrow_a \mathcal{A}(i, M, Q, F, F_d, e_\Box[v])}$$

$$\text{Field-Set}\ \frac{F = F' \uplus \mathcal{F}(f, v_0) \quad F'' = F' \cup \{\mathcal{F}(f, v)\}}{\mathcal{A}(i, M, Q, F, F_d, e_\Box[(\text{set! } f\ v)]) \rightarrow_a \mathcal{A}(i, M, Q, F'', F_d, e_\Box[v])}$$

$$\text{Process-Msg}\ \frac{Q = Q' \cdot \mathcal{M}sg(i', m, \overline{v}) \qquad \mathcal{M}(m, \overline{x}, e) \in M}{\mathcal{A}(i, M, Q, F, F_d, v) \rightarrow_a \mathcal{A}(i, M, Q', F, F_d, e[i'/self][\overline{v}/\overline{x}])}$$

Figure 6: Actor-local reduction rules.

4.5 Dynamic Semantics

Evaluation Contexts We use evaluation contexts [13] to abstract over the context of an expression, and to indicate which subexpressions should be fully reduced before a compound expression can be reduced. The expression $e_\Box$ denotes an expression with a "hole" to identify the next subexpression to be reduced. The notation $e_\Box[e]$ indicates that expression e is part of an abstracted compound expression $e_\Box$, and that e should be reduced first before $e_\Box$ can be reduced.

$$e_\Box ::= \begin{array}{c} \Box \mid (\text{set! } f\ e_\Box) \mid (\text{send } e_\Box\ m\ \overline{e}) \mid \\ (\text{send } v\ m\ \overline{v}\ e_\Box\ \overline{e}) \mid (\text{delegate } d\ m\ \overline{v}\ e_\Box\ \overline{e}) \end{array}$$

Evaluation Rules Our evaluation rules are defined in terms of a reduction on sets of configurations $K \rightarrow K'$. For clarity we split the rules defining this reduction in two parts. Actor-local rules are defined in terms of a reduction $a \rightarrow_a a'$ and can be applied in isolation (within one actor). Actor-global rules are defined in terms of a reduction $K \rightarrow_k K'$ and indicate interactions between actors.

Actor-local Evaluation Rules Actors continually dequeue the first message from their message queue, retrieve the correct expression to process this message, and reduce this expression to a value. The next message can only be processed after the expression is reduced to a value. An actor is considered idle when its message queue is empty and its current expression has been completely reduced. This is the only situation in which no rules apply to a particular actor. Otherwise, if an actor is not idle and no rules can reduce its current expression, there is an error in the program. We summarise the actor-local reduction rules in Figure 6.

Field-Get, Field-Set Values of fields are stored in a set F of 2-tuples that map fields to values. A `get` expression is a simple lookup of the field, and `set!` replaces the current association with a new one.

Process-Msg Processing a message is only possible when the message queue Q is not empty and the current expression is reduced to a value. The last entry of the queue is removed and the corresponding method is retrieved. To evaluate the body of the method we substitute the formal parameters $\overline{x}$ and $self$ with the values contained within the message. Note that `self` is either the current actor (when the message was sent via a normal message send) or the delegator (when the message was delegated).

$$\text{Send}\ \frac{A = A' \uplus \mathcal{A}(i', M', Q', F', F'_d, e') \qquad Q'' = \mathcal{M}sg(i', m, \overline{v}) \cdot Q'}{\begin{array}{c}\mathcal{K}(A \uplus \mathcal{A}(i, M, Q, F, F_d, e_\square[(\text{send } i'\ m\ \overline{v})]), C) \rightarrow_k \\ \mathcal{K}(A' \cup \{\mathcal{A}(i, M, Q, F, F_d, e_\square[null]), \mathcal{A}(i', M', Q'', F', F'_d, e')\}, C)\end{array}}$$

$$\text{Delegate}\ \frac{\begin{array}{c}\mathcal{F}(d, i') \in F_d \\ A = A' \uplus \mathcal{A}(i', M', Q', F', F'_d, e') \qquad Q'' = \mathcal{M}sg(i, m, \overline{v}) \cdot Q'\end{array}}{\begin{array}{c}\mathcal{K}(A \uplus \mathcal{A}(i, M, Q, F, F_d, e_\square[(\text{delegate } d\ m\ \overline{v})]), C) \rightarrow_k \\ \mathcal{K}(A' \cup \{\mathcal{A}(i, M, Q, F, F_d, e_\square[null]), \mathcal{A}(i', M', Q'', F', F'_d, e')\}, C)\end{array}}$$

$$\text{Spawn}\ \frac{\text{makeActor}(n, C) = \mathcal{A}(i', M', Q', F', F'_d, e') \cdot \overline{delegates}}{\begin{array}{c}\mathcal{K}(A \uplus \mathcal{A}(i, M, Q, F, F_d, e_\square[(\text{spawn } n)]), C) \rightarrow_k \\ \mathcal{K}(A \cup \{\mathcal{A}(i, M, Q, F, F_d, e_\square[i']), \mathcal{A}(i', M', Q', F', F'_d, e')\} \cup \overline{delegates}, C)\end{array}}$$

$$\text{Congruence}\ \frac{a \rightarrow_a a'}{K \uplus \mathcal{K}(A \uplus a, C) \rightarrow_k K \cup \mathcal{K}(A \uplus a', C)}$$

Figure 7: Actor-global reduction rules.

$$\text{zipFields}(\emptyset, \emptyset) = \emptyset$$

$$\text{zipFields}(d \uplus \overline{d}, \mathcal{A}(i, M, Q, F, F_d, e) \uplus \overline{delegates}) = \{\mathcal{F}(d, i)\} \cup \text{zipFields}(\overline{d}, \overline{delegates})$$

$$\text{makeActor}(n, C) = \{\mathcal{A}(i, M, \emptyset, F, F_d, null)\} \cdot \overline{delegates}$$

with i fresh, $\quad \mathcal{C}(n, \overline{f}, \overline{d}, M) \in C, \quad F = \{\mathcal{F}(f, null) \mid f \in \overline{f}\}$

$\overline{delegates} = \{\text{makeActor}(n', C) \mid n' \in \overline{d}\}$

$F_d = \text{zipFields}(\overline{d}, \overline{delegates})$

Figure 8: Auxiliary functions to create actors.

Actor-global Evaluation Rules We summarise the actor-global evaluation rules of Figure 7.

Send Describes an asynchronous message send to an actor. A new message is added at the front of the queue Q' of the receiving actor i'. The address of the `self` reference passed as an argument in the message is also i'. This means that the receiving actor will execute the message using its own address as `self` parameter. Semantically, all arguments $\overline{v}$ are passed to the other actor via a (deep) copy, but in our case there is no assignment other than local fields, and therefore we do not explicitly create copies in this formalisation. The `send` expression reduces to `null`.

Delegate Describes delegating messages. This rule is almost identical to `Send`, except that the address of the receiver i' is stored in a delegate field d in F_d, and that the address of the sender i is passed in the message instead of i'. Thus, when the receiver eventually processes this message, any messages it sends to `self` during execution will be sent to the delegator i.

Spawn This rule describes the spawning of an actor given a classname n. Spawning an actor may add multiple actors to the program, namely the actor with the spawned behaviour and all of its delegates (and all of their delegates, ...). To create these actors in a single evaluation step we define an auxiliary function `makeActor` in Figure 8 that, given a classname n and all classes C, returns

$$\frac{\text{makeActor}(Main, C) = \mathcal{A}(i, M, Q, F, F_d, null)}{\langle \emptyset, C \rangle \rightarrow \mathcal{K}(\mathcal{A}(i, M, \{\mathcal{M}sg(i, start, \emptyset)\}, F, F_d, null), C)}$$

Figure 9: Program initialization.

a sequence of newly created actors. The first element of this sequence is the actor spawned from behaviour n, whose address i' is the value of the reduced spawn expression. All newly created actors are added to the configuration.

Congruence This rule relates the local and global reduction rules such that reductions of local rules also progress the global system.

Finally, the rule in Figure 9 bridges compilation and evaluation, and shows how to reduce a fully compiled program represented by a tuple $\langle \emptyset, C \rangle$ into the first configuration of the program. The first actor is an instance of the `Main` class and contains a `start` message in its mailbox.

5 Conclusion

Code reusability is an important aspect of software engineering that can improve software quality and reliability of actor-based systems. We approach this topic in Section 2 by discussing different kinds of code reusability mechanisms in different kinds of actor models. The discussed mechanisms do not fulfil 2 requirements that we find essential for programming actor-based systems: extensibility and reusability.

We introduce a prototype language in Section 3 called Stella with a behaviour composition mechanism based on delegation-based trait composition. Here, a compound behaviour essentially describes a collection of actors that are composed at runtime such that some messages are implicitly delegated from one actor to another. It fulfils the requirement of extensibility because behaviours can be easily extended with new methods defined elsewhere, and it fulfils self-containment because every part of a composed behaviour can, by itself, also be used to create new actors.

Acknowledgements

We would like to thank Thierry Renaux for his insightful comments on drafts of this paper. Sam Van den Vonder is supported by the Research Foundation - Flanders (FWO) under grant No. 1S95318N.

References

1. Pony tutorial: What about inheritance? https://web.archive.org/web/20180717115657/https://tutorial.ponylang.org/types/classes.html, accessed: 2018-07-17

2. ReactiveX: An api for asynchronous programmingwith observable streams. http://web.archive.org/web/20180717115824/http://reactivex.io/ (2018), accessed: 2018-07-17
3. Agha, G.: Concurrent object-oriented programming. Commun. ACM **33**(9), 125–141 (1990)
4. Agha, G.A.: A model of concurrent computation in distributed systems. the MIT Press (1986)
5. Armstrong, J., Virding, R., Williams, M.: Concurrent programming in ERLANG. Prentice Hall (1993)
6. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Computer Languages, Systems & Structures **34**(2-3), 83–108 (2008)
7. Bloch, J.J.: Effective Java, 2nd Edition. The Java series ... from the source, Addison-Wesley (2008)
8. Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E.B., Pun, K.I., Tarifa, S.L.T., Wrigstad, T., Yang, A.M.: Parallel objects for multicores: A glimpse at the parallel language encore. In: Bernardo, M., Johnsen, E.B. (eds.) Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures. Lecture Notes in Computer Science, vol. 9104, pp. 1–56. Springer (2015)
9. Clebsch, S., Drossopoulou, S., Blessing, S., McNeil, A.: Deny capabilities for safe, fast actors. In: Boix, E.G., Haller, P., Ricci, A., Varela, C. (eds.) Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015. pp. 1–12. ACM (2015)
10. Cutsem, T.V., Bergel, A., Ducasse, S., Meuter, W.D.: Adding state and visibility control to traits using lexical nesting. In: Drossopoulou, S. (ed.) ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5653, pp. 220–243. Springer (2009)
11. Cutsem, T.V., Boix, E.G., Scholliers, C., Carreton, A.L., Harnie, D., Pinte, K., Meuter, W.D.: Ambienttalk: programming responsive mobile peer-to-peer applications with actors. Computer Languages, Systems & Structures **40**(3-4), 112–136 (2014)
12. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. **28**(2), 331–388 (2006)
13. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2), 235–271 (1992)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley (1994)
15. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of object-oriented programming **1**(2), 22–35 (1988)
16. Kafura, D.G.: Concurrent object-oriented real-times systems research. SIGPLAN Notices **24**(4), 203–205 (1989)
17. Kafura, D.G., Lee, K.H.: Inheritance in actor based concurrent object-oriented languages. Comput. J. **32**(4), 297–304 (1989)
18. Koster, J.D., Cutsem, T.V., Meuter, W.D.: 43 years of actors: a taxonomy of actor models and their key properties. In: Clebsch, S., Desell, T., Haller, P., Ricci, A. (eds.) Proceedings of the 6th International Workshop on Programming Based

on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016. pp. 31–40. ACM (2016)
19. Lieberman, H.: Using prototypical objects to implement shared behavior in object oriented systems. In: Meyrowitz [21], pp. 214–223
20. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10), 40–51 (1992)
21. Meyrowitz, N.K. (ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings. ACM (1986)
22. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency among strangers. In: Nicola, R.D., Sangiorgi, D. (eds.) Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3705, pp. 195–229. Springer (2005)
23. Roestenburg, R., Bakker, R., Williams, R.: Akka in action. Manning Publications Co. (2015)
24. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.E., et al.: Object-oriented modeling and design, vol. 199. Prentice-hall Englewood Cliffs, NJ (1991)
25. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6183, pp. 275–299. Springer (2010)
26. Thomas, D.: Programming Elixir. Pragmatic Bookshelf (2018)
27. Yonezawa, A., Briot, J., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Meyrowitz [21], pp. 258–268