

AErlang: Empowering Erlang with Attribute-Based Communication

Rocco De Nicola^{1(✉)}, Tan Duong^{2(✉)}, Omar Inverso^{2(✉)},
and Catia Trubiani^{2(✉)}

¹ IMT Institute for Advanced Studies Lucca, Lucca, Italy
`rocco.denicola@imtlucca.it`

² Gran Sasso Science Institute, L'Aquila, Italy
`{tan.duong,omar.inverso,catia.trubiani}@gssi.it`

Abstract. Attribute-based communication provides a novel mechanism to dynamically select groups of communicating entities by relying on predicates over their exposed attributes. In this paper, we embed the basic primitives for attribute-based communication into the functional concurrent language Erlang to obtain what we call AErlang, for attribute Erlang. To evaluate our prototype in terms of performance overhead and scalability we consider solutions of the *Stable Marriage Problem* based on predicates over attributes and on the classical preference lists, and use them to compare the runtime performance of AErlang with those of Erlang and X10. The outcome of the comparison shows that the overhead introduced by the new communication primitives is acceptable, and our prototype can compete performance-wise with an ad-hoc parallel solution in X10.

Keywords: Attribute-based communication · Erlang · Concurrency · Distributed programming · Collective adaptive systems · Stable marriage

1 Introduction

Collective adaptive systems (CAS) are typically large conglomerates of components which are not entirely aware of themselves as members of a collectivity and interact according to limited mutual knowledge and local rules, indirectly triggering global system evolution [14]. Eventually, despite the simplicity of the components in isolation, the global behaviour of the system may end up being quite sophisticated, hardly predictable to seemingly chaotic.

These classes of systems pose challenges at many levels. Reasoning about them is difficult. In fact, assessing specific properties, such as stability and convergence, or forecasting emerging behaviour is usually really hard due to non-linearity and non-determinism. Further sources of complexity are in the following distinguishing features:

- Anonymity: the identity of components is not known;
- Open-endedness: new components may enter or leave at any time;
- Adaptivity: rôles and interests of components may change;
- Scalability: the number of components might grow very fast and large.

All these features are fairly visible, for example, in ant colonies, as well as stock markets, robot swarms, and social networks. In the presence of these features, programming may be difficult. For instance, in an anonymous and open-ended environment traditional mechanisms such as point-to-point communication are hardly acceptable. Similarly, the limited expressiveness of mainstream programming paradigms often makes it inconvenient to describe adaptive behaviour. In addition, the size of the system exacerbates this situation and on a practical standpoint is cause of concern about performance.

Some of the above issues can be addressed with adequate descriptive formalisms. Among these, *Attribute-based Communication* [2, 12] is a particularly appropriate one. With this approach, components are modelled as processes exposing attributes, i.e., relevant features according to the problem domain and to the local or global behaviours of interest, and process interaction is driven by predicates over these. Communication takes place in an implicit selective multicast fashion, and interactions among components are dynamically established by taking into account “connections” as determined by predicates over their attributes. A command $send(v)@\pi$ expresses the intention of sending v to all entities satisfying predicate π while $receive(x)@\pi'$ indicates willingness to receive messages by entities satisfying predicate π' while binding the received values to x . Components can update their attributes via assignments, $[a := v]$. In this way, collectives are dynamically formed at the time of interaction by considering the sets of receiving components that satisfy the sending predicates.

Let us now consider a social network scenario where users aim at forming groups for language exchange. Such groups may be formed by only considering the language users wish to learn and the one potential partners are interested in. However in case of multiple alternatives it might be desirable to prefer people with similar age and interests, or even knowledgeable of a second language in common. Relevant attributes would then be spoken languages, age, language of interest, and so on. Predicates are built by specifying conditions on the attributes, e.g., $age \leq 25 \wedge language = English$. Note that here the identity of users is irrelevant for forming the groups, and no change in the predicates is required when users join or leave the system, thus anonymity and open-endedness are no longer a concern. In general, attribute-based programming allows to naturally capture the essence of a system to a very good level of abstraction without having to worry about a number of details that normally have to be taken under due consideration when using more traditional alternatives.

In this paper, we seek to leverage the benefits of attribute-based communication at a programming level while addressing the performance concerns at the same time. Our contribution is twofold.

As a first contribution, we combine attribute-based communication with functional-style programming by instantiating attribute-based programming

abstractions on top of the *Erlang* programming language. We targeted *Erlang* as the host platform for our prototype language extension due to its native support for concurrency and distribution, its scalability, and the inherent modularity of functional-style programming. Its concurrency model is very lightweight and has solid foundations based on Actors [1], thus, in principle, avoids thread-and-lock problems. Moreover, it fits very well with the *AbC* process calculus [2], since both consider processes as basic units of computation that communicate via asynchronous message passing. Our prototype language extension, *AErlang*, is a middleware enabling attribute-based communication among *Erlang* processes, with the aim of preserving *Erlang*'s excellent scalability. *AErlang* plays the role of global process registry which allows processes to register and update their attributes. It also takes charge of forwarding messages from senders to receivers by evaluating the predicates they supply. In this way, programmers are relieved from the burden of working out details such as the explicit handling of attributes, the evaluation of predicates, and so on. Our attribute-based programming technique can naturally model the main distinguishing features of collective adaptive systems at no extra effort, whereas under a traditional programming setting such a task would require major and time-consuming operations.

As a second contribution, we provide a performance evaluation of our prototype in terms of efficiency and scalability. We assess the effectiveness of our prototype by using it to program a solution to *Stable Marriage* [17] that aims at matching members according to their preferences. For this problem, we consider implementations of different variants in different languages. Namely, we first consider a variant explicitly based on (predicates over) attributes and provide an implementation in *AErlang*. Then, we derive preference lists from the predicates and implement the classical algorithm in *AErlang*, *Erlang*, and *X10*, a language specifically designed to scale with the number of cores [10]. The different implementations are instrumental to compare performances of our solutions. The experimental results show that the overhead resulting from using the new communication primitives is acceptable, and our prototype successfully preserves *Erlang*'s scalability. Moreover, on very large instances the *AErlang* program for the attribute-based solution turns out to scale considerably better than a state-of-the-art parallel version based on adaptive search and implemented in *X10* [30].

The rest of the paper is organized as follows. We describe how to extend *Erlang* with attribute-based communication constructs in Sect. 2. Example programs of *AErlang* are presented in Sect. 3. In Sect. 4 we evaluate our prototype in terms of efficiency and scalability. Related works are discussed in Sect. 5, conclusions and future research directions are provided in Sect. 6.

2 AErlang

AErlang instantiates attribute-based communication on top of the *Erlang* programming language.

Erlang [4,35] is a concurrent functional programming language originally designed for building telecommunication systems [7] and recently successfully adapted to broader contexts, such as large-scale distributed messaging platforms [29,31]. It supports concurrency [5] and inter-process communication natively through a compact set of powerful primitives. The *Erlang* concurrency model is based on the Actor Model [1,20]. Actors are processes that can asynchronously exchange messages while preserving the order of outbound messages. Each process has its own unlimited mailbox for storing incoming messages that are retrieved via pattern matching. The lightweight and scalable concurrency model and the modularity of functional-style programming [22,23] make *Erlang* particularly appropriate for building massively scalable distributed systems.

AErlang lifts *Erlang*'s send and receive communication primitives to attribute-based reasoning. In *Erlang*, the send primitive `!` requires an explicit destination address (e.g., registered name, process identifier) for message passing. In contrast, *AErlang* processes are not aware of the presence and identity of each other, and communicate using predicates over attributes.

AErlang aims at relieving programmers from the burden of working out details such as the explicit handling of attributes, the evaluation of predicates, and so on, while at the same time preserving *Erlang*'s excellent scalability. Our prototype extension is implemented as a middleware that plays the role of global process registry and takes charge of forwarding messages from senders to receivers by evaluating the predicates they supply.

2.1 Programming Interface

The programming interface of our prototype is presented in Fig. 1. Processes joining the system need to register their details (e.g., process identifier, attributes) using function `register`, which takes as input a process attribute environment `Env` in form of a either a proper list or a map, i.e., pairs of attribute names and their associated values. After the registration, processes can manage their local environment by using the `setAtt` and `getAtt` functions. Processes leaving the system may actively `unregister`, and when a process unregisters, then it is no longer able to use attribute-based communication.

% initialization	% attribute-based	% attribute-based
<code>aerl:start()</code>	% send and receive	% send and receive with counting
	<code>to(Pred) ! Msg</code>	<code>Count = to(Pred) ! Msg</code>
% join and leave		
<code>aerl:register(Env)</code>	<code>from(Pred),</code>	<code>from(Pred,Count),</code>
<code>aerl:unregister()</code>	<code>receive</code>	<code>receive</code>
	<code>Pattern_1 -> Expression_1;</code>	<code>Pattern_1 -> Expression_1;</code>
	<code>...</code>	<code>...</code>
% environment handling	<code>Pattern_n -> Expression_n</code>	<code>Pattern_n -> Expression_n</code>
<code>aerl:setAtt(List)</code>	<code>end</code>	<code>end</code>
<code>aerl:getAtt(List)</code>		

Fig. 1. *AErlang* programming interface.

Registered *AErlang* processes interact via attribute-based send and receive actions. Differently from standard *Erlang*, this pair of communication primitives replace source and destination identifiers with arbitrary predicates over the declared attributes. In particular, attribute-based send is used to send a message `Msg` to all processes whose attributes satisfy predicate `Pred`. On the other hand, attribute-based receive is used to receive messages sent by using attribute-based send. The receipt of a message is conditioned by the attribute values satisfying predicate `Pred`. A receive operation has the effect of retrieving from the receivers' mailbox any message that matches the receiving predicate.

Predicates are strings containing Boolean expressions. They can be over attribute names (*Erlang* atoms), constants (written with a prefix underscore, e.g., `_constant`), process-local references to attributes (written as `this.a`), and process-local variables representing values (written as `$X`). Apart from comparison operators and logical connectives, it is possible to use arithmetic operators, such as `+`, `*`, `/`, `-` between predicate terms. Furthermore, predicates can contain the operator `in`, which denotes the membership relation between an element and a list, and allow the use of user-defined functions.

AErlang provides the possibility for processes to count with how many partners they are currently interacting and to parallelize the communication, so to increase both flexibility and performance without affecting expressiveness. Although these primitives are not originally described in the *AbC* calculus, we provide variants of attribute-based send and receive actions as shown in the right-most column of Fig. 1. In particular, `to(Pred)` can return the number of selected receivers (whom the middleware forwards the message to) at communication time. On the other hand, the attribute-based multi-receive `from(Pred,Count)` takes this as an extra argument and blocks until the given amount of incoming messages is received. Internally the receive operation processes multiple incoming messages satisfying the receiving predicates, up to the given count. This is helpful when the sender is interested to hear back from its communication partners. `Count` is the number of selected receivers at communication time, which is always greater than the number of actual receivers, which in turn bounds the number of receivers willing to answer. Therefore the sender knows the maximum number of expected incoming messages before moving to the next action.

2.2 Prototype Architecture

There are two main components in *AErlang*: (i) a process registry that keeps track of process details such as the process identifier and the current status, and (ii) a message broker that undertakes the delivery of outgoing messages.

Process Registry. It is a generic server that accepts requests regarding process (un)registration and internal updates. It stores process identifiers and all the information used by the message broker to deliver messages.

Our prototype uses as the main storage back-end Mnesia, *Erlang*'s built-in distributed database. When a process joins the system, the `register` function does several things. First, the process environment is stored into an ETS table and the reference to this table is stored into the process dictionary. This information is local to the process. Second, a service request to the process registry is performed, to insert process details, including the attribute environment, into an Mnesia table. We currently store attributes in separate columns for increased performance and at the expense of some extra memory. All the above information is removed when the `unregistration` procedure is invoked.

Message Broker. It is responsible for delivering messages between processes. It is implemented as an Erlang server process listening for interactions from attribute-based send. To address potential bottlenecks arising in the presence of a very large number of processes, the message broker can be set up to run in multiple parallel threads. Similarly to the process registry, *Erlang*'s runtime system provides distribution for the message broker.

A sending action is characterized by a sending predicate, a message and sender's environment. All these elements are wrapped up into a single message and passed to the message broker. When such a message arrives, the message broker performs the following steps:

1. parse the predicates and converts them into a database query;
2. select the receivers by applying the query to the process database;
3. forward the message to all the selected receivers.

The exact behaviour depends, however, on the specific operating mode chosen at the moment of initializing *AErlang*. More specifically, there are two kinds of checks that need to be evaluated for a receiver to receive a message:

- the sending predicate is checked against the receiver's environment,
- the receiving predicate is checked against the sender's environment.

The current prototype implemented the following message forwarding strategies for the message broker: (i) *broadcast*, i.e., the broker forwards any outbound message to every components in the system, then these filter the received messages according to both the sending and receiving predicates; (ii) *pushing*, i.e., the broker only checks the sending predicates and forwards messages to selected receivers that will use the receiving predicates to decide whether to accept any incoming message; (iii) *pulling*, i.e., the broker only checks the receiving predicates and only forwards messages from selected senders; the forwarded messages are then filtered by the receiver according to the sending predicates; (iv) *push-pull*, i.e., the message broker checks for both the sending and receiving predicates before forwarding any message. The choice of one message forwarding policy over the others depends on the specific class of problems under consideration. For example, broadcast can guarantee consistency in highly dynamic systems, but it is quite expensive due to the large number of forwarded messages. On the

other hand, if consistency is not the major concern, then pushing is more suitable when attributes do not change frequently, while pushpull works well when even the predicates are quite static.

3 Programming with A Erlang

In this section we present attribute-based programming in A Erlang. By adopting as a case study the well-known problem of Stable Marriage (SM), we begin with describing a program that implements the classical solution for this problem, and then consider progressively more elaborate variants, with the purpose of showing the convenience of using attributes and a suitable programming technique. At the end of the section we discuss how the proposed approach can be generalised to model realistic examples of collective adaptive systems such as social networks.

3.1 Stable Marriage

SM consists in finding a matching between sets of men and women, where each person has a preference list of members of the opposite gender [17]. A *matching* is a set of one-to-one assignments between men and women. Each assignment is denoted by a *pair* (m, w) , where m and w indicate the two matched *partners*. A pair is *blocking* if, according to their respective preference lists, both the matched man and woman prefer someone else to their partners. A matching is *stable* if there is no blocking pair. A matching is *complete* when everybody is matched, *incomplete* otherwise.

SM has many practical applications [19], and has been intensively studied in the literature, together with its variants [24]. In the classical form, the preference lists are strictly-ordered and complete. For this, Gale and Shapley gave an efficient algorithm to find a stable matching [17]. It can be informally summarized as follows. Each man actively proposes himself to his most favourite woman according to his preference list. Whenever a man is rejected, he tries again with the next woman in the list. On the other hand, each woman continuously waits for incoming proposals. A woman without a partner immediately accepts any proposal, otherwise she compares the proposer with her current partner. She then rejects the man whom she likes less, according to her preference list. The algorithm terminates when every man has a partner.

Variations of this algorithm consider other kinds of preference list: incomplete (SMI), with ties (SMT), or both (SMTI). While the first two variants can be solved similarly to the classical case, SMTI is hard [25]. In this paper we investigate a new variant of the algorithm where the matching happens by taking into account the mutual interests of partners characteristics, rather than preference lists of identifiers. We call this variant stable marriage with attributes (SMA). Note that SMA can always be cast into SM by converting preferences over attributes to preferences over identifiers. This can be done by assigning a weight to each attribute and summing up all the attributes exposed by the identifiers to obtain the preference list.

Table 1. Correspondence between preference lists and predicate lists

men's lists	women's lists	men's id	men's predicates	women's id	women's predicates
$m_1: w_1 w_2$	$w_1: (m_1 m_2)$	m_1	$id = w_1, id = w_2$	w_1	$id = m_1 \text{ or } id = m_2$
$m_2: w_1 w_2$	$w_2: m_1$	m_2	$id = w_1, id = w_2$	w_2	$id = m_1$

3.2 Stable Marriage with Preference List

We now consider a variant of SM known as SMTI [25], in which the preference list is incomplete and partially ordered, i.e., a man or a woman may like several people at the same level. The preference list is thus a list of sets rather than single elements and we refer to such sets as *ties*.

We model this problem in AErLang by introducing an attribute `id` to represent people identifiers and predicates over these to specify the preferences. As an example, Table 1 shows the predicate lists induced from a SMTI instance (on the left) where ties are enclosed by parentheses. To implement preference list we use predicates over the attribute `id`, where ties are modelled by predicates with logical disjunction on equality comparison. We refer to the newly derived lists on the right table as predicate lists.

We then solve the problem under this new representation of preferences by using a simple solution which is similar to the classical Gale-Shapley algorithm described in Sect. 3.1, but uses a slightly different protocol and is converted to message-passing style.

The *AErLang* program for STMI is shown in Fig. 2. Function `man()` takes as arguments the preference list `Prefs` of a man and his identifier `Id`. The first element in `Prefs` is bound to variable `H` by pattern matching on list (line 2). A man goes through a proposing phase from lines 3 to 14. First, he sends a `propose` message using "`id in $H`" as the sending predicate (line 3) which has the effect of contacting all women whose `id` belongs to the list `H`. He then waits for enough answers from the women he contacted using the attribute-based receive construct with counting (line 4), with the same predicate used when sending. Inside the body of this receive operation, the man is only interested in `yes` messages. He becomes aware of his status by checking attribute `partner` (line 7) to take a decision. If he has no partner, he sends a `confirm` message to the first woman who said `yes` by using her identifier `W` attached in the reply message. He then considers this woman as his current partner (line 10), and informs any other interested women that he is no longer available by sending them a `busy` message (line 12).

After the proposing phase, a man can either be alone or engaged (checked by line 15). In the first case, he does not consider any woman in the current predicate `H` and tries to `propose` himself again to the women in the remaining part of his preferences (line 16). In the second case, he takes no action unless he receives a `goodbye` message from his partner (lines 17–19), in which case he tries proposing himself again using his current predicate unchanged (line 21). The man keeps the predicate unchanged as it may include other women.

```

1  man(Prefs,Id) ->
2    [H|T]=Prefs,
3    Count=to("id in $H") ! {propose,Id},
4    from("id in $H",Count),
5    receive
6      {yes,W} ->
7        case aerl:getA(partner) of
8          none ->
9            to("id=$W") ! confirm,
10           aerl:setA(partner,W);
11          _ ->
12            to("id=$W") ! busy
13        end
14    end,
15    case aerl:getA(partner) of
16      none -> man(T,Id);
17      _ -> from("id=this.partner"),
18          receive
19            goodbye ->
20              aerl:setA(partner,none),
21              man(Prefs,Id)
22          end
23    end.

24 woman(Prefs,Id,P) ->
25   from("bof($Prefs,$P,id)",
26   receive
27     {propose,M} ->
28       to("id=$M") ! {yes,Id},
29       from("id=$M"),
30       receive
31         confirm ->
32           to("id=$P") ! goodbye,
33           woman(Prefs,Id,M)
34         busy ->
35           woman(Prefs,Id,P)
36       end
37   end.

```

Fig. 2. Stable marriage with preference lists in A Erlang (SM-aerl).

Function `woman()` takes as arguments a preference list `Prefs`, an identifier `Id`, and the partner's identifier `P`. A woman always waits for proposals from men who are better than her current partner. This comparison is performed with the `bof` function (line 25) that checks if a proposer preceeds the current partner `P` in the woman's preference list. If this is the case, then the woman sends back a `yes` message and waits for an `confirm` message from the new man `M`. After `M` confirms to her, the woman gets engaged to him by keeping `M` in the recursive call (line 33), after rejecting her current partner `P`. Otherwise, she keeps listening for other proposals (line 35).

3.3 Stable Marriage with Attributes

In this variant each person has a set of attributes describing their own characteristics and some preferences over the attributes of their potential partners. Each attribute has a finite domain, while preferences are represented by logical expressions over the attributes of the partners. For simplicity, in this section we only consider simple predicates where preferences are conjunctions of equality comparisons.

Table 2 shows an example of SMA instance of size four where each person has two attributes, which in turn have two possible values. This example points out the expressive power of attribute-based communication. In fact, our program for SMA (Fig. 3) is very similar to the program proposed in previous section (Fig. 2), and the differences are mostly accommodated by altering the predicates. In addition, men can progressively adapt their preferences to increase the chances to find a partner. For example, there is no woman in Table 2 satisfying the requirements of man `m1`, hence he looks for partners partially matching his

Table 2. Attributes and preferences for men and women.

Id	Wealth	Body	Preferences	Id	Eyes	Hair	Preferences
m1	rich	strong	eyes=amber \wedge hair=red	w1	amber	dark	wealth=poor \wedge body=weak
m2	rich	weak	eyes=green \wedge hair=dark	w2	amber	dark	wealth=rich \wedge body=strong
m3	poor	strong	eyes=green \wedge hair=red	w3	green	red	wealth=rich \wedge body=strong
m4	poor	weak	eyes=amber \wedge hair=red	w4	green	dark	wealth=rich \wedge body=weak

Table 3. Predicate lists for men.

Id	Relaxation of preferences		
m1	eyes=amber \wedge hair=red	eyes=amber	hair=red
m2	eyes=green \wedge hair=dark	eyes=green	hair=dark
m3	eyes=green \wedge hair=red	eyes=green	hair=red
m4	eyes=amber \wedge hair=red	eyes=amber	hair=red

initial preferences. This adaptive behaviour is achieved by transforming plain preferences into predicate lists, as shown in Table 3. For example, when man **m1** relaxes his preferences and look for women with amber eyes only, then there are women **w1** and **w2** satisfying such predicate. We assume that the ordering of attributes within a predicate indicates their priority.

Figure 3 shows a possible AErland implementation for SMA. Function **man()** takes as arguments the predicates list **Prefs** of a man, his **Id** and characteristics **Atts**. The first element in **Prefs** (i.e., the most demanding predicate) is bound to variable **H** by pattern matching on list (line 2). The proposing phase of a man is implemented by lines 3–14 and follows the same behaviour described in previous section.

```

1  man(Prefs,Id,Atts) ->
2    [H|_]=Prefs,
3    Count=to(H) ! {propose,Id,Atts},
4    from(H,Count),
5    receive
6      {yes,W} ->
7        case aerl:getA(partner) of
8          none ->
9            to("id=$W") ! confirm,
10           aerl:setA(partner,W);
11          _ ->
12            to("id=$W") ! busy
13        end
14    end,
15    case aerl:getA(partner) of
16      none -> man(T,Id,Atts);
17      _ -> from("id=this.partner"),
18          receive
19            goodbye ->
20              aerl:setA(partner,none),
21              man(Prefs,Id,Atts)
22          end
23    end.

24  woman(Prefs,Id,P,PA) ->
25    from("bof($Prefs,$PA,wealth,body)",
26    receive
27      {propose,M,MA} ->
28        to("id=$M") ! {yes,Id},
29        from("id=$M"),
30        receive
31          confirm ->
32            to("id=$P") ! goodbye,
33            woman(Prefs,Id,M,MA)
34          busy ->
35            woman(Prefs,Id,P,PA)
36        end
37    end.

```

Fig. 3. Stable Marriage with Attributes in AErland (SMA-aerl).

Function `woman()` takes as arguments the preferences `Prefs`, an identifier `Id`, in addition to arguments `P` and `PA` to keep the current partner's information. A woman waits for proposals from men whose attributes are better than her current partner. This comparison is performed with the `bof` boolean function (line 25) that checks if a proposer is characterized by attributes `wealth`, `body` better than the partner `P` characterized by the variable `PA`. If this function provides true as output, then the woman sends a `yes` message back and waits for an acknowledge message `confirm` from this man `M`. If `M` confirms to her, the woman gets engaged to him by keeping `M` and his characteristics `MA` in the recursive call (line 33), after rejecting her current partner `P`. Otherwise, she keeps listening for other proposals (line 35).

3.4 Social Networking with Attributes

By abstracting `SM`, we are able to deal with the more realistic setting of social networking. In fact, this domain nicely fits with our new programming abstractions that can be naturally used to express attribute-based interaction. In particular, a generalization of stable marriage can be applied to open-ended systems where many-to-many matchings are allowed and the stability requirement is dropped. This appears indeed to be quite a common case in large-scale social networks, as we are going to discuss shortly.

In the social networking domain, attributes can represent characteristics of the users, such as their hobbies, musical preferences, current location, age, spoken languages, personality, mood, groups they belong to, their contact list (if they decide to make it public). Note that some of these attributes, for example location and mood, can change dynamically.

Possible interactions between users could happen when the interests of two or more users match. For example, people could mutually look for other people to jointly participate in a certain activity according to some specific criteria which could be expressed using a predicate over the given attributes. More concretely, let us consider a language exchange scenario where initially one could only look for the language she wishes to learn and the one their potential partners are interested in. In addition, however, it might be convenient to prefer somebody with similar age and interests, or even knowledgeable of a second language in common. Possible attributes for one user joining the system are: the `language` that a user already knows, the language of `interest`, `age`, `hobby` and so on. Interaction might be naturally expressed by the following code snippets, where users advertise their own interest by sending their proposal:

```
to("language = this.interest") ! {Language, Id}
```

Another user may set up a receive waiting for somebody knowing the language that she is interested in, conditioned to the matching of the hobby and only if the potential partner is at most five years older than the user:

```
from("this.age - age < 5 and this.hobby = hobby and language =
this.interest"), receive
  {Language, Buddy} ->
    to("id = $Buddy") ! {ok, Id}
end
```

The language exchange scenario above demonstrates the high flexibility and expressiveness provided by attribute-based communication over traditional actor-to-actor communication [1]. The interactions among components flexibly arise from the sending and receiving predicates whose expressiveness allows to suitably select the communicating entities. The handling of attributes inevitably introduces a performance overhead that results to be acceptable (see Sect. 4).

Besides this we also consider the case when new pairs can join or leave the group of entities aiming at finding matching partners. Such situation can easily be dealt with when the partner selection is predicate-based, but clearly requires significant work when preferences are expressed via lists of identifiers, as these have to be recalculated whenever the set of users in the system changes.

4 Performance Evaluation

In this section we present the performance evaluation of AErlang. The conducted experimentation focuses on two main aspects: (i) the efficiency in terms of runtime overhead, see Sect. 4.1; (ii) the scalability in terms of size of the instances and hardware resources, see Sect. 4.2. Experimental results are reproducible since our prototype is publicly available¹.

4.1 Efficiency

To evaluate the efficiency of AErlang, we compared the runtime performance of SMA-aerl, SM-aerl, and SM-erl, an Erlang program implementing the same matching protocol used in SM-aerl. All three programs were used to solve the SMA problem instances. We used the pushing message forwarding policy (see Sect. 2.2) for this part of the experiments.

Firstly, we generated multiple random input instances by considering problem sizes from 100 to 500. We considered two attributes for women and two for men, each attribute having a domain of two values (like in Table 2), with a probability of occurrence ranging from 0.1 to 0.9. We used the same ranges for preferences. We selected 24 different combinations in the given probability ranges, and generated 10 instances for each combination. Since SM-erl and SM-aerl take preference lists as input, we have also converted the problem instances to use preference lists. Finally, we ran each instance 10 times and took the average execution times. The hardware environment is a machine consisting of 4 CPUs AMD Opteron 6376 2.3 GHz, 2 MB Cache, 64 GB RAM. The versions of OS and Erlang were Linux 4.4.0-62-generic and 19.1, respectively.

Table 4 reports the runtime ratio of the SM-aerl and SMA-aerl programs with respect to SM-erl. Here, columns list the instance size, whereas rows enumerate the compared variants. We observe that the ratio is always within the same order of magnitude, more precisely we found a maximum ratio of 2.99 (observed for SM-aerl vs SM-erl with 100 instance size), as highlighted by the bold entry in

¹ <https://github.com/ArBITRAL/AErlang>.

Table 4. Runtime ratio AERlang vs Erlang.

	size				
	100	200	300	400	500
SM-aerl vs SM-erl	2.99	1.73	1.92	1.98	2.20
SMA-aerl vs SM-erl	2.21	1.36	1.36	1.43	1.65
SMA-aerl vs SM-aerl	0.73	0.71	0.72	0.72	0.75

Table 4, and a minimum one of 1.36 (observed for SMA-aerl vs SM-erl with 200 and 300 instance size). This suggests that the new programming abstractions introduce an acceptable performance overhead (always within the same order of magnitude) which is minimized when attributes are considered for predicate evaluation. In fact, in Table 4 we can notice that SMA-aerl always shows lower ratios with respect to SM-aerl. This is not affected by the instance size, i.e., with larger instance sizes the ratio remains within the min-max values observed for rather small instance sizes.

It is worth to notice that the SMA-aerl variant always outperforms SM-aerl, as showed in the last row of Table 4. I due to the different cost of predicate evaluation, in fact the former uses sending predicates whose complexity is independent from the input size (e.g., "hair=blonde and eye=amber") whereas the corresponding predicates of the latter need to check membership of identifiers within ties and therefore may be as large as the size of the tie itself (e.g., "id=w1 or id=w2 or ..."). Note that this also holds at the receiver side.

4.2 Scalability

The scalability of our prototype is demonstrated while increasing: (i) the size of the input instances from 1k to 5k and comparing AERlang with AS-X10; (ii) the number of cores from 2 to 48 and comparing AERlang with its Erlang counterpart.

Comparison with AS-X10. In [30], the authors proposed adaptive search as an efficient approach to solve the SMTI and SMI problems. They model SMTI as a permutation problem and try to resolve blocking pairs until an acceptable size of the matching is achieved. Their framework, implemented in the X10 programming language (AS-X10) can handle instances up to the size of 1000 pairs with good performance and scalability on a large number of cores thanks to a fine-tuned cooperation mechanism between many parallel solvers.

In this experiment we used the inputs originally described in [18], which are generated by using their tool² that takes three parameters as input: (p_1) size of the instance, (p_2) probability of incompleteness, (p_3) probability of ties.

² <https://github.com/dannymrock/SMTI-AS-X10>.

For this comparison we optimized for performance the SM-aerl program shown in Fig. 2. In particular, we did tailor the selection mechanism of the message broker to exploit the structure of the sending predicate of men (i.e., checking the membership of an identifier within a tie is eventually expanded to disjunctions of identities checks). Note that our prototype allows arbitrarily complex expressions and function calls, however their repeated parsing and evaluation affects performance. A way to avoid this is to set the receiving predicate to true and to evaluate the function locally. Our prototype currently does not implement such mechanism, but we simulated it by simply moving the comparison function (`bof`) from the predicate to the local code for women. Furthermore, we used the pushpull message forwarding policy (see Sect. 2.2) as it performed best in this specific case.

We have generated two classes of instances while considering instance sizes up to 5 thousands pairs of elements and the following sets of parameters: (i) 80% of incompleteness and no-ties instances (i.e., $p_2 = 0.8$, $p_3 = 0$); (ii) 95% of incompleteness and 80% of no-ties instances (i.e., $p_2 = 0.95$, $p_3 = 0.8$). These parameters were intentionally selected to be in line with those chosen in the evaluation of the adaptive search approach, for a fair comparison [30].

This part of the experiments was run on an idle local workstation equipped with 128 GB of memory, a dual Intel Xeon processor E5-2643 v3 (12 physical cores in total) clocked at 3.40 GHz, and running a 64-bit generic Linux kernel version 4.4.0, Erlang/OTP version 19.1, and X10 version 2.4.2.

Figure 4 shows that X10 performs faster than AERlang only on small instances with 1 thousands of pairs of elements. However we do notice that when increasing the size of the instances the AERlang program turns out to scale considerably better. This gap tends to increase with size, making the AERlang program very suitable to larger instances.

Comparison with Erlang. We wrote an Erlang program for the classical algorithm by Gale-Shapley, and used it to compare runtime performance with the AERlang program for SMTI. In this experiment, AERlang is configured with pushpull message dispatching policy (see Sect. 2.2). We also used the same input generator to generate problem instances for both SMTI and SMI problems.

We ran the AERlang program for SMTI and the Erlang program for SMI to safely exclude any hidden complexity due to the management of the ties. The size of the instances is fixed to the largest available option, i.e., 10 thousands of pairs of elements, and by ranging the number of cores from 2 to 48. We ran 10 instances, 10 times each, and collected the average execution times as previously. This experiment was performed on a computing cluster [34] where we had access to nodes with 64 Intel CPUs clocked at 2.3 GHz and 110 GB of memory running a scientific Linux distribution.

The results are presented in Fig. 5, where the x-axis denotes the number of cores and the y-axis reports the execution time in seconds on a logarithmic scale. Interestingly, the pronounced fluctuations in the running times are consistent for both AERlang and Erlang programs. This suggests that performance glitches within the Erlang subsystem end up affecting our AERlang prototype too.

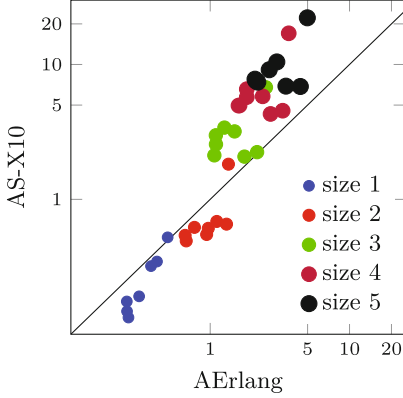


Fig. 4. AERlang vs. AS-X10

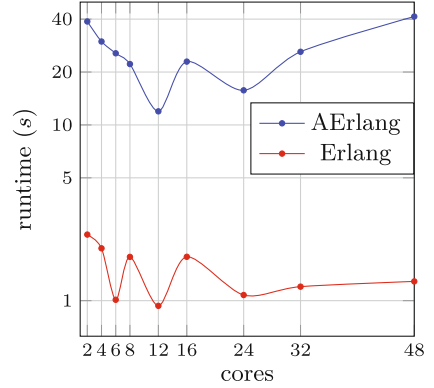


Fig. 5. AERlang vs. Erlang

Summarizing, we can conclude that introducing the attribute-based programming abstraction introduces a reasonable performance overhead. The experimental results confirm that nevertheless the scalability provided by the underlying runtime system is not significantly affected. In practice it is still possible to challenge and outperform ad-hoc state-of-art distributed algorithms conceived for large-scale systems.

5 Related Work

Attribute-based communication has been explored in the context of autonomic computing by the research centered around the SCEL paradigm [36]. It has been used to model the dynamic formation of ensembles from interacting autonomic components [12]. Notably, this novel communication paradigm can also be used to model a wide range of adaptation patterns in autonomic systems [9]. In our previous work [11] we provided a preliminary assessment of AERlang, where we used a simple program for a stable marriage variant without ties, to give a hint of what could be done with AERlang. No performance evaluation was considered. In this paper, instead, we develop an extended programming interface allowing counting and parallel message delivery, along with a comprehensive evaluation of the approach in terms of efficiency and scalability.

To the best of our knowledge, only two more efforts have been made on instantiating attribute-based communication, both on top of the Java programming language. The first work enriches the language with the primitives of the AbC calculus [3], but it only supports the broadcasting method. This simplifies the design and implementation of the message broker, it introduces communication overhead, especially in large systems. Being aware of this issue, AERlang's message broker includes three other message-dispatching strategies, allowing users to trade off depending on the application domain. The second work is jRESP [26], based instead on the SCEL paradigm, and more oriented towards autonomic

and adaptive systems. jRESP designates ports with specific roles at nodes (or components) for communication. Nodes agreeing to interact via a port and can use the communication protocol (such as broadcast via a central server, multicast or point-to-point) that the port supports. The main difference with our approach is that we also consider strategies which filter early group of partners by exploiting updated predicates and attributes.

Erlang has been used as the host language for incorporating domain specific abstractions to deal with multi-agents and self-adaptive systems [13, 28, 33]. Among others, we mention ContextErlang [33] which is an extension of Erlang according to Context-Oriented Programming [21]. ContextErlang extends Erlang’s `gen_server` behaviour with `context_agent` whose callback functions can be overridden by (functions implementing) variations at runtime. During operation, a context change triggers the activation of the corresponding variations, which leads to changing the behaviour of `context_agents`. The difference from our approach is in that we exploit exposed attributes, thus processes can adapt their behaviour implicitly using predicate-based message passing. In practice, via attributes that are updated by relying on appropriate sensors, we can model context-awareness.

The use of source-to-source transformation for extending Erlang with new primitives has been demonstrated in JErLang [32]. JErLang provides a receive-like join construct inspired from Join-Calculus [16]. Apart from transformation, their implementation intercepts the Erlang receive algorithm to incorporate the joins resolution mechanism, together with low-level optimizes inside Erlang’s VM. Our AErlang prototype on the other hand focuses on mediating message passing based on predicates with appropriate handling of process attributes, and leading to user-friendly communication primitives.

6 Conclusion

In this paper, we have been experimenting with attribute-based communication and functional-style programming. We have proposed a prototype language extension, namely *AErlang*, that enables attribute-based communication among *Erlang* processes. *AErlang* conveniently combines the benefits of this novel paradigm with the efficiency and scalability of *Erlang*. Our approach copes well with the main sources of complexity of collective adaptive systems, such as anonymity, adaptivity, open-endedness, and their large size. It allows programmers to concentrate on the essence of the system being implemented, by relieving them from the burden of working out low-level details on a case-by-case basis.

We have evaluated the efficiency and scalability of our approach. Experiments compared the runtime performance of functional-style implementations for a known solution to a hard matching problem, and have shown that the overhead resulting from using the new communication primitives is acceptable, and our prototype successfully preserves *Erlang*’s efficiency and scalability.

We have also implemented a variant of the above matching problem that requires a more involved interaction pattern. We compared this variant to an

ad-hoc parallel version based on adaptive search implemented in *X10* [30] that can scale very well when increasing the number of cores. The experimental results have shown that our prototype does not currently scale well when increasing the number of cores. This is possibly partly due to known potential performance drains within the underlying *Erlang* subsystem which are being actively investigated [6, 8, 27]. However, *AErlang* does indeed scale considerably well on large instances, whereas these turn out to be progressively out of reach for the algorithm based on adaptive search implemented in *X10*.

Further experimentation is needed to improve *AErlang* and make it more attractive in practice. An extensive evaluation on arbitrarily large instances that use complex predicates and frequently changing attributes would be useful to assess the overall robustness. An in-depth performance evaluation to understand whether the large size of the system stresses the underlying scheduling mechanisms would be very useful. A systematic evaluation of the cost of predicate handling would be highly beneficial to improve efficiency. Indeed, since predicates can have an arbitrary complexity, their evaluation may add a significant overhead, and efficient predicate evaluation is known to be non-trivial [15]; looking for more efficient ways to handle predicate evaluation is thus very important. Lastly, handling process attributes does require complicated bookkeeping that has to take into account synchronisation, possible data inconsistencies, and so on. A comprehensive experimentation by varying the number of attributes, the size of the domains, the frequency of their updates, and their probability distribution would be very useful to devise different handling strategies according to a finer-grained classification of the attributes.

We plan to apply attribute-based communication to other concurrent languages, such as Go and Scala. Extending our experimentation across different programming environments would certainly allow a deeper investigation on the effectiveness of attribute-based communication.

References

1. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (1986)
2. Abd Alrahman, Y., De Nicola, R., Loret, M.: On the power of attribute-based communication. In: Albert, E., Lanese, I. (eds.) *FORTE 2016*. LNCS, vol. 9688, pp. 1–18. Springer, Cham (2016). doi:[10.1007/978-3-319-39570-8_1](https://doi.org/10.1007/978-3-319-39570-8_1)
3. Alrahman, Y.A., Loret, M.: *AbCuS: a run-time environment of the AbC calculus* (2016). <https://github.com/lazkany/AbC>
4. Armstrong, J.: *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf (2007)
5. Armstrong, J.: Erlang. *Commun. ACM* **53**(9), 68–75 (2010)
6. Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., Venetis, I.E.: A scalability benchmark suite for Erlang/OTP. In: *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, pp. 33–42. ACM (2012)
7. Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G.: AXD 301: A new generation ATM switching system. *Comput. Networks* **31**(6), 559–582 (1999)

8. Boudeville, O., Cesarini, F., Chechina, N., Lundin, K., Papaspyrou, N., Sagonas, K., Thompson, S., Trinder, P., Wiger, U.: RELEASE: a high-level paradigm for reliable large-scale server software. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 263–278. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-40447-4_17](https://doi.org/10.1007/978-3-642-40447-4_17)
9. Cesari, L., Nicola, R., Pugliese, R., Puviani, M., Tiezzi, F., Zambonelli, F.: Formalising adaptation patterns for autonomic ensembles. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 100–118. Springer, Cham (2014). doi:[10.1007/978-3-319-07602-7_8](https://doi.org/10.1007/978-3-319-07602-7_8)
10. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: ACM SIGPLAN Notices, vol. 40, pp. 519–538. ACM (2005)
11. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: AErlang at work. In: Steffen, B., Baier, C., Brand, M., Eder, J., Hinchey, M., Margaria, T. (eds.) SOFSEM 2017. LNCS, vol. 10139, pp. 485–497. Springer, Cham (2017). doi:[10.1007/978-3-319-51963-0_38](https://doi.org/10.1007/978-3-319-51963-0_38)
12. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **9**(2), 7 (2014)
13. Díaz, Á.F., Earle, C.B., Fredlund, L.Å.: eJason: an implementation of Jason in Erlang. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS, vol. 7837, pp. 1–16. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-38700-5_1](https://doi.org/10.1007/978-3-642-38700-5_1)
14. Ferscha, A.: Collective adaptive systems. In: ACM International Joint Conference on Pervasive and Ubiquitous Computing, pp. 893–895. ACM (2015)
15. Fontoura, M., Sadanandan, S., Shanmugasundaram, J., Vassilvitski, S., Vee, E., Venkatesan, S., Zien, J.: Efficiently evaluating complex boolean expressions. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 3–14. ACM (2010)
16. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 372–385. ACM (1996)
17. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. *Am. Math. Monthly* **69**(1), 9–15 (1962)
18. Gent, I.P., Prosser, P.: An empirical study of the stable marriage problem with ties and incomplete lists. In: Proceedings of the 15th European Conference on Artificial Intelligence, pp. 141–145. IOS Press (2002)
19. Harrenstein, P., Manlove, D., Wooldridge, M.: The joy of matching. *IEEE Intell. Syst.* **28**(2), 81–85 (2013)
20. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, pp. 235–245. Morgan Kaufmann Publishers Inc. (1973)
21. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. *J. Object Technol.* **7**(3), 125–151 (2008)
22. Hu, Z., Hughes, J., Wang, M.: How functional programming mattered. *Ntl. Sci. Rev.* **2**(3), 349–370 (2015)
23. Hughes, J.: Why functional programming matters. *Comput. J.* **32**(2), 98–107 (1989)
24. Iwama, K., Miyazaki, S.: A survey of the stable marriage problem and its variants. In: International Conference on Informatics Education and Research for Knowledge-Circulating Society, ICKS 2008, pp. 131–136. IEEE (2008)

25. Iwama, K., Miyazaki, S., Morita, Y., Manlove, D.: Stable marriage with incomplete lists and ties. In: Wiedermann, J., Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 443–452. Springer, Heidelberg (1999). doi:[10.1007/3-540-48523-6_41](https://doi.org/10.1007/3-540-48523-6_41)
26. jRESP: Java Runtime Environment for SCEL Programs. <http://jresp.sourceforge.net/>
27. Klaftenegger, D., Sagonas, K., Winblad, K.: On the scalability of the Erlang term storage. In: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, pp. 15–26. ACM (2013)
28. Krzywicki, D., Turek, W., Byrski, A., Kisiel-Dorohinicki, M.: Massively concurrent agent-based evolutionary computing. *J. Comput. Sci.* **11**, 153–162 (2015)
29. Letuchy, E.: Facebook Chat (2008). <http://web.archive.org/web/20160303044321/https://www.facebook.com/notes/facebook-engineering/facebook-chat/14218138919/>
30. Munera, D., Diaz, D., Abreu, S., Rossi, F., Saraswat, V., Codognet, P.: Solving hard stable matching problems via local search and cooperative parallelization. In: 29th AAAI Conference on Artificial Intelligence (2015)
31. O’Connell, A.: Inside Erlang, The Rare Programming Language Behind WhatsApp’s Success (2014). [http://web.archive.org/web/20160715132942/](http://web.archive.org/web/20160715132942/http://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success), <http://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success>
32. Plociniczak, H., Eisenbach, S.: JERlang: Erlang with joins. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 61–75. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-13414-2_5](https://doi.org/10.1007/978-3-642-13414-2_5)
33. Salvaneschi, G., Ghezzi, C., Pradella, M.: ContextErlang: a language for distributed context-aware self-adaptive applications. *Sci. Comput. Program.* **102**, 20–43 (2015)
34. Stalio, S., Di Carlo, G., Parlato, S., Spinnato, P.: Resource management on a VM based computer cluster for scientific computing. arXiv preprint [arXiv:1212.4658](https://arxiv.org/abs/1212.4658) (2012)
35. Thompson, S., Cesarini, F.: Erlang programming: a concurrent approach to software development (2009)
36. Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.): Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998. Springer, Cham (2015)

Coordination Models and Languages

19th IFIP WG 6.1 International Conference,

COORDINATION 2017, Held as Part of the 12th

International Federated Conference on Distributed

Computing Techniques, DisCoTec 2017, Neuchâtel,

Switzerland, June 19-22, 2017, Proceedings

Jacquet, J.-M.; Massink, M. (Eds.)

2017, XII, 279 p. 94 illus., Softcover

ISBN: 978-3-319-59745-4