AErlang

Empowering Erlang with Attribute-based Communication

Name1 Name2
Affiliation 1/2
Email 1/2

Name3 Name4
Affiliation 3/4
Email 3/4

Abstract

Attribute-based communication provides a novel mechanism to dynamically select groups of communicating entities by relying on predicates over their exposed attributes. This paradigm represents an interesting alternative to broadcasting and one-to-one communication, and has potential applications in modeling and analysing complex dynamic systems, such as collective adaptive systems. 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. By comparing the runtime performance of functional-style implementations in Erlang and AErlang of a hard matching problem, we show that the overhead introduced by the new communication primitives is acceptable. Moreover, by considering the same problem, we show that our prototype can compete performance-wise with an ad-hoc parallel version, based on adaptive search and implemented in X10.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications —Concurrent, distributed, and parallel languages; D.1.3 [*Programming Techniques*]: Concurrent Programming —Distributed programming

Keywords Attribute-based communication, Erlang, Concurrency, Distributed programming, Collective Adaptive Systems.

1. Introduction

Ant colonies, as well as stock markets, social media networks, robot swarms, can be seen as specific instances of collective adaptive systems (CAS). Typically, these systems are large conglomerates of components which are not fully aware of themselves as members of a collectivity. CAS components can join and leave the community at any moment, and they only rely on a limited mutual knowledge and on local rules that determine how to interact, indirectly triggering global system evolution.

Components of collective adaptive systems are relatively simple in isolation, however simplicity tends to progressively fade out at coarser-grained levels of abstraction. For example, components can congregate to pursue possibly conflicting goals; groups can in turn dynamically change, and so the local behaviour may have to adapt accordingly. Often, the global behaviour of the system ends up being quite sophisticated, hardly predictable or even seemingly chaotic. In fact, extracting behavioural patterns by naively observing the system is non-trivial. Predicting the global behaviour may likewise be difficult, even if the local behaviour of each component is perfectly clear. In general, due to non-linearity, non-determinism, or other sources of complexity, assessing specific properties, such as stability and convergence, or forecasting an emerging behaviour is really challenging.

Advancements on the understanding of collective adaptive systems are therefore very sought after, more so considering the variety of fields these systems pertain to. Many formalisms to describe them have been proposed. Among these formalisms, particularly interesting appears to be the one that relies on *attribute-based communication* to model component interaction. With this approach, components are modelled as processes with attributes, and process interaction is captured by predicates over them. 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. Attributes correspond to exposed features depending on the problem domain or on local or global behaviours of interest. In this way, collectives are dynamically formed at the time of interaction by considering the set of receiving components that satisfy sender's predicates.

In our view, interaction based on predicates over attributes allows naturally capturing many interesting system features that are usually tricky to catch otherwise, such as anonymity, dynamicity, and openness and is a perfect linguistic abstraction to program and reason about CAS. Preliminary work in this direction has shown how the new communication primitives can be used both to specify and verify some key properties of a simple case study [2]. Here, we provide a new implementation, along with evidence of its effectiveness by evaluating its efficiency and scalability.

As an example of effectiveness of attribute-based communication, consider users in a typical social media network aiming at forming groups for language exchange. Initially one could only take into account the language she wishes to learn and the one their potential partners are interested in, but, in case of multiple alternatives it might be convenient to prefer people with similar age and interests, or even knowledgeable of a second language in common. Relevant attributes for this application could be spoken languages, age, interests, and the language one wishes to learn. Predicates would be the conditions already discussed. Interesting properties to prove or disprove could be the guarantee that more than a given percentage of the members are able to join a group of interest to them.

Another example could be a disaster recovery scenario where a swarm of robots with different equipments cooperate in rescuing a victim. To this purpose, some robots have to gather in a targeted location and different equipment or abilities might be needed by taking into account that only some of the robots might have the required tools, and they might be busy. The group in this case has to be formed considering that a certain amount of robots is needed to perform the task and by taking into account the robots' attributes like position, available equipment, energy level, priority of their current task, and so on. Now, a desirable property would be the guarantee that if two robots are equipped with the same tool and are both idle, then the one closer to the victim moves to the required location.

Attribute-based communication appears to be a more direct language supporting the non-trivial component interaction, however it is still not clear whether this would allow to effectively implement real systems in practice. In fact, on a realistic system with a large number of components, even routine operations whose cost may otherwise be considered negligible, such as scheduling, can suddenly become quite expensive and generate significant performance overhead. Similarly, scalability may be difficult to achieve, for instance due to the complicated bookkeeping needed to track possibly frequently changing attributes across multiple processes, having to deal with synchronisation, inconsistency, and so on.

In this paper, we address the above concerns by embedding attribute-based communication in a functional programming setting. Specifically, our contribution is twofold.

As a first contribution, we instantiate the discussed programming abstractions on top of $Erlang$. We have designed and implemented our prototype extension, namely $AErlang$, as a middleware enabling attribute-based communication among $Erlang$ processes, with the aim of preserving $Erlang$'s excellent scalability. Our middleware 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. We have targeted $Erlang$ because it has been designed to make concurrent and distributed programming easy and has solid foundations relying on the concurrency model based on actors [3] which, in principle, avoids thread-and-lock problems. Moreover, the $Erlang$ concurrency model fits very well with the AbC process calculus [1], since both consider processes as basic units of computation and rely on asynchronous message passing.

As a second contribution, we provide a preliminary evaluation of our prototype in terms of performance overhead and scalability and identify specific potential hindrances to the use of the new communication paradigm on real instances. We assess the effectiveness of our prototype by using it to program a solution for the *Stable Marriage Problem* (SMP) [17] that can be seen as a special case of the above considered problems.We focus on the well studied simpler problem to compare performance of our solution with those implemented in other languages; specifically plain $Erlang$ and $X10$, a language specifically designed to scale with the number of cores [13]. Results are indeed encouraging. Experiments show that the overhead resulting from using the new communication primitives is acceptable, and our prototype successfully preserves $Erlang$'s scalability. Moreover, the comparison of the solution of the above matching problem to an ad-hoc parallel version based on adaptive search implemented in $X10$ [24] shows that our prototype does not currently reach a great scalability when increasing the number of cores. However, our prototype does scale considerably better on very large instances.

The rest of the paper is organized as follows. We briefly review the AbC process calculus and the main features of $Erlang$ in Section 2. We describe how to extend $Erlang$ with attribute-based communication constructs in Section 3. Example programs of $AErlang$ are presented in Section 4. In Section 5 we evaluate our prototype in terms of efficiency and scalability. Related works are discussed in Section 6, conclusions and future research directions are provided in Section 7.

2. Background

In this section we first provide a brief description of the origin and on the distinguishing operators of the AbC calculus by focusing on the input and output actions that are the core components of our extended Erlang implementation, then we describe some of the main feature of $Erlang$.

The attribute based paradigm was first introduced in [14] where $SCEL$, a language for autonomic computing was presented, and then studied in [4] where the AbC calculus distilled from $SCEL$ has been put forward to study the theoretical impact of the new communication paradigm. Attribute-based communication has also been studied on a more practical standpoint. The basic communication primitives introduced have been used to add new programming abstraction corresponding to this new communication to $Java$ [5]. A similar exercise may be done in any programming language that supports the implementation of at least the three main programming abstractions corresponding to the following operators of AbC:

- attribute-based input, $\Pi(x)$, used to receive messages from any component whose attributes satisfy the predicate Π and to bind the received values to variable x;
- attribute-based output, $(E)@\Pi$, that first evaluates the expressions E and then sends the returned value to all components whose attributes satisfy predicate Π;
- update operation, $[a := E]$, that sets the value of attribute a to the evaluation of expression E.

This approach inherits as immediate benefits the previously argued advantages of modelling interaction using predicates. Indeed, the more direct language resulting from introducing the new programming abstractions allows to easily capture non-trivial component interaction.

Erlang [6, 27] is a concurrent functional programming language originally designed for building large-scale telecommunication systems [9] and recently successfully adapted to broader contexts (for example, in large-scale distributed messaging platforms [1] [2]) following its open source release. It supports concurrency [7] and inter-process communication at language-level through a compact set of powerful primitives.

Erlang's standard environment includes a runtime system with built-in support for transparent distribution and fault tolerance, and a collection of robust open-source components that have already been tested and successfully used in a number of industrial applications [12]. OTP is set of Erlang libraries and design principles supporting the development, it includes a robust fault-tolerant distributed database as well as many design patterns, called OTP behaviours, which take care of non-functional parts that are commonly used in many applications, for instance dynamic-code upgrade.

The Erlang concurrency model is based on the Actor Model, an alternative approach to concurrency that in principle avoids thread-and-lock problems, initially proposed by C. Hewitt [18] and formalized by G. Agha [3]. An actor is a fundamental unit of computation that contains three elements: processing, memory and communication. An actor can create more actors, send messages to other actors, and determine its future behavior upon receiving new messages. Communication is asynchronous, i.e., the sender does not wait that the sent message is received before progressing. Message exchange is the only mean to communicate as actors do not share any state.

In Erlang, actors are processes that can send messages to each other using send and receive actions. The ordering between outbound messages is preserved. Each process has a private mailbox for storing incoming messages, which can grow and shrink dynamically. Messages are retrieved from the mailbox by relying on pattern matching: if a message matches a particular pattern, the corresponding expression is evaluated and the message is removed from the mailbox.

The efficient concurrency model (which seems to be lightweight and potentially very scalable) and the functional-style programming which in principle guarantees great modularity [20, 21], make Erlang particularly appropriate for building massively scalable distributed systems.

3. Implementation

In this section we describe the implementation details of AErlang. In Figure 1 we report the programming API available to Erlang programmers. A top-level process starts AErlang by invoking the `start` function which initializes the message-passing environment for attribute-based communication.

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 that is represented using either a proper list or a map, e.g., $\#\{\mathtt{a}_1 => \mathtt{v}_1, \mathtt{a}_2 => \mathtt{v}_2, \ldots\}$, where $\mathtt{a_i}$ is an atom denoting the attribute name and $\mathtt{v_i}$ is the corresponding value that can be a number, atom, or a list. The attribute environment encodes aspects of the application domain. The programmer declares the environment, and this implies that attribute names are supposed to be public among AErlang processes. After the registration, processes can manage their local environment by using the `setAtt(s)` and `getAtt(s)` functions. Processes may actively `unregister`, and when a process unregisters, then it is unable to use attribute-based communication actions.

Registered AErlang processes communicate by using attribute-based send and receive. Differently from standard Erlang, this pair of communication primitives replaces 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`. Predicates are represented as Erlang strings (also see Section.3.2).

[1] `https://www.facebook.com/notes/facebook-engineering/facebook-chat/14218138919/`

[2] `https://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success`

```
% initialization
aerl:start()

% registration and unregistration
aerl:register(Env)
aerl:unregister()

% environment handling
aerl:setAtts(TupleList)
aerl:getAtts(NameList)

% attribute-based send and receive
to(Pred) ! Msg

from(Pred),
receive
    Pattern_1 -> Expression_1;
    ...
    Pattern_n -> Expression_n
end

% send and receive with counting
to_c(Pred) ! Msg

from(Pred,Count),
receive
    Pattern_1 -> Expression_1;
    ...
    Pattern_n -> Expression_n
end
```

Figure 1: AErlang interface.

In order to improve the communication capabilities, we added the possibility for AErlang processes to count with how many partners they are currently interacting. Although these primitives are not originally described in the AbC calculus, we implement variants of attribute-based send and receive as shown in Figure 1. In particular, the primitive `to_c(Pred)` performs an attribute send similar to `to`, but in addition it returns the number of selected receivers at the communication time. The attribute-based multi-receive takes as input an extra integer argument and blocks until the given amount of incoming messages is received. In practice this operation performs a loop to process multiple messages sent by processes satisfying the receiving predicates. This is especially relevant in the context of a highly dynamic and anonymous environment performing communication-intensive tasks (as discussed in 3.1.2).

3.1 Prototype Architecture

The design of AErlang follows a centralized architecture which consists of two main components: (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 decides how and where to deliver outgoing messages.

3.1.1 Process registry

The process registry is a generic server which accepts requests regarding to process (un)registration and internal updates. It maintains process identifiers and necessary information that will be retrieved by the message broker to compute predicates used in communication actions. Synchronous and asynchronous handling of incoming messages is supported, and is inherited from the `gen_server` design pattern.

Our prototype currently uses as a storage back-end Mnesia, Erlang's built-in distributed database. When a process joins the system, the `register` function does several things. On behalf of the process, it stores the process environment into an ETS table for local access, and the table reference into the process dictionary. Note that, since we only use use the process dictionary to store the references to the table, which never change, the known unwanted side effects due to using the dictionary are avoided. The next step performed during the registration is call to the process registry which monitors the process itself and inserts process details into Mnesia. The process registry performs dirty read and write operations on a Mnesia table, that can be replicated over a cluster of Erlang nodes. The process identifier is used as the primary key of this table (which is possible because of the location transparency feature of Mnesia). A table record storing information per process also includes the attribute environment, a reference number obtained from monitoring action, and a receiving predicate initialized as `true`. On the other hand, invoking the `unregistration` procedure causes the process registry to remove all the above information.

3.1.2 Message Broker

AErlang's message broker is responsible for delivering messages between processes. It is an Erlang `gen_server` listening for interactions from attribute-based send. A sending action is characterized by a sending predicate (`Ps`), a message (`Msg`) and sender's environment (`Envs`). All these elements are wrapped up into a single message and passed to the message broker. When such a message arrives, the message broker spawns a handler process for the actual message delivery. In general, the handler takes care of (1) predicate parsing, (2) database records selection and (3) message forwarding. Its exact behaviour depends however on the specific operating mode chosen at the moment of initializing AErlang, as will be shortly described. In the AbC calculus, receiving predicates can be specified over both the message content and the attributes of the sender. In AErlang we do not worry about explicitly handling predicates on the message content, since Erlang's pattern matching already allows to filter messages according to the message content.

Each receiver performs two kinds of checks to actually receive a message:

1. the sender predicate `Ps` is checked by considering the receiver's environment `Envr`,
2. the receiver predicate `Pr` is checked by considering the sender's environment `Envs` embedded in the sent message.

Both these checks can be performed in different ways according to the specific message forwarding policy. In the sequel, we discuss available strategies in AErlang with their advantages and drawbacks.

Broadcast One of the most straightforward strategy to realize attribute-based communication is to broadcast the sent message to all other processes in the system. In this way, AErlang selects all processes in the process registry, except the sender, and forwards the message to them. It is then the responsibility of receivers to decide whether to actually receive the message or not, by performing the necessary checks on both the sending and the receiving predicates.

The main advantage of this communication pattern is that messages are always delivered to all interested receivers, and the design of the central server is kept simple. However, this choice might lead to high communication overhead because messages are always forwarded to all registered processes, without considering whether or not they are actually waiting to perform a receive action.

Pulling A way to reduce the broadcast overhead is to update the receiving predicates in the process registry. This methodology filters the messages to be delivered by considering the receiver's interests. In order to receive a message the receivers should upload their interests (i.e., predicates) to the central server. AErlang then can use these predicates to preselect groups of senders. This early server-side check reduces the number of forwarded messages because those processes that are not interested in receiving messages from some groups of senders will not receive any.

The drawback of this method is that the actual predicate update triggered by a receive action may take place after a sending action. This would cause loss of the message. Tolerance of this problem might depend on the specific application. For example, the problem can be ignored in case of receivers inputting from a potentially large group of interested senders, this is the case of wireless sensor networks where message loss is somehow expected. Any application where processes do not often update predicates may also benefit from this strategy. For example, in a messaging application that allows users to specify predicates as policies in order to receive messages of interest only. Summarising, the pull strategy tends to be effective when changes of user policies do not take place often, or when loss of messages during a predicate update can be acceptable.

Pushing We consider an alternative strategy which enriches the broker with information about the attribute environment of processes. Clearly, whenever a process updates any of its attributes in the local environment, it has to report the changes to the broker.

When updating attributes, problems might arise due to the interval passing from the local update of the environment of a process and the one of the corresponding environment in the broker. As before, this problem can cause message loss. A way out is to introduce atomicity of the sequence consisting of the two updates, for example using transactions. To guarantee good performances, the current implementation does not consider this issue. We think that the pushing approach is suitable when considering scenarios where processes attributes do not change frequently.

PushPull This strategy combines Pushing and Pulling approaches. It means that both the receiver and the sender messages are checked by the message broker. This implies that messages are forwarded only if there are suitable receivers, thus minimizing the overall number of messages circulating in the network. As advantage, we are able to count the delivered messages and store the number of actual communicating entities, as shown in Figure 1. This strategy may be particularly convenient on communication-intensive rather than computation-intensive scenarios.

Summarising, there is no best strategy satisfying both reliability and efficiency criteria. We have implemented all strategies and we leave users the possibility to choose one of them when initializing the central server, based on trade-offs coming from their application domain.

3.2 Predicate Evaluation

In this section we describe how AErlang deals with predicates and evaluates them within an environment. Predicate terms can be attribute names, references to attribute values, constants or variables. A basic predicate is a comparison between two terms. Compound predicates are constructed from simpler ones using logical connectives In practice, predicates are represented as Erlang strings, where apart from connective and comparison operators (that are rendered as Erlang comparison and logical operators), predicates can contain attribute names (represented as Erlang atoms), constants (start with a underscore, e.g., `_constant`), local references to attribute values (written as `this.a`), and also local variables (written as `$X`). Furthermore, it is also possible to use arithmetic operators, such as $+, *, /, -$ and the operator `in` to represent the relationship between an element and a list. The use of Erlang functions (e.g. built-in or user-defined functions), though powerful is disallowed.

Predicates are processed into three steps. The source transformation detects the syntax of attribute-based send and receive primitives in user's code, and pre-processes variables appearing in predicates if there any. The preprocessing step includes building a list of variable bindings and storing them in a process dictionary for latter retrieving. In the second step, predicates are evaluated under the local environment with the help of a scanner generated from Erlang leex. The evaluation replaces the occurrences of terms `this.a` by their corresponding values stored in the attribute environment. In addition, variable names and constant notations (if any) are replaced by their actual values. The partially evaluated predicates are sent to the message broker. In the pushpull and pushing modes, it translate sending predicates into Mnesia queries and selects the set of receivers accordingly. In other cases, a predicate is translated into a higher oder function with the help of an interpreter based on Erlang yecc parser tool. The function takes an external environment as input and return the truth value of the original predicate.

4. Programming with AErlang

We now use AErlang to implement possible solutions to the well-known stable marriage problem (SMP) [17]. The problem consists in finding a matching between two disjoint sets of men and women, where each person has a preference list of members of the opposite sex. A *matching* is a one-to-one assignment between the men and the women, and it is said to be *stable* if there exists no pair (m, w) such that man m prefers woman w to his matched partner and vice versa. A matching is said to contain a *blocking* pair (m, w) if there exist a man m' and a woman w' such that man m prefers w' over w' and w prefers m' rather than m. A matching is *incomplete* if there exists at least a man or a woman who has no partner.

Gale and Shapley proposed an algorithm to find a stable matching in [17]. The algorithm 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. Each woman on the other hand continuously waits for incoming proposals. A free woman immediately accepts the proposal and becomes engaged, 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 all men are engaged.

We consider two variants of the above problem. In the first variant, known as SMI problem, the preference list is strictly ordered but incomplete, i.e., the preference list of a man (resp. woman) does not necessarily include all women (resp. men). In the second variant, known as SMTI problem, the preference list is still incomplete but no longer strictly 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. In the rest of the paper, we refer to such sets as *ties*, and to the SMI and SMTI problems as the version with no-ties and ties, respectively.

The SMI problem can be solved using the classical Gale-Shapley algorithm. Here we implement it using message passing (Fig. 3). In this section we use this version only to show how naturally the novel programming abstractions provided by AErlang can describe algorithms such as this one. We use it later in Section 5 to evaluate the performance overhead w.r.t. the native programming language by comparing the runtime performance of two equivalent AErlang and Erlang programs implementing the same algorithm. We also show a possible Erlang implementation (Fig. 2) for comparison, however we omit the description because it is very similar.

We follow the approach based on the new attribute-based programming abstraction. Men and women are modelled as interacting processes associated with their own attributes: identifier `id`, preference list `prefs`, current `partner` and `gender`. These processes are spawn from functions `man()` and `woman()`. Process environments are handled by using getter and setter functions. A `propose` message from a man means that he is proposing himself to some woman, while a `no` message from a woman means that she is either rejecting a proposal, or breaking up with her current partner. At each iteration, a man extracts his favourite woman from the top of his preference list and sends her a `propose` message. Note that in this version the man assumes that the woman will silently accept his own proposal, or send an explicit reject message otherwise. The attribute-based send is illustrated in line 4, where the sending predicate is directed at `"id = this.partner"`, which has the effect of sending a message to processes whose `id` equals to the value of attribute `partner`. In lines 5-8, the man uses an attribute-based receive using the same predicate to receive a message from his partner only.

On the other hand, women continuously wait for incoming proposals from any man, using as receiving predicate `"gender = _man"` at line 10[3]. A woman takes her decision by using a helper function `bof` that compares two men and returns `true` if the first one appears before the second one in her preference list. The woman rejects her current partner (line 16) in case she found somebody she likes more than him, otherwise she rejects the proposer and keeps her current partner (line 19)[4]

The above program uses attributes, but in essence the communication is still point-to-point because the sending predicates always map to unique receivers. To show the expressive power of AErlang's communication primitives, we now use them to program a solution for the SMTI problem. This program will be used later in Section. 5 to evaluate AErlang's scalability by comparing it against a state-of-the-art and ad-hoc solution based on adaptive search.

In this version, men and women are encoded as AErlang processes with attributes exactly as before, but their behaviour is slightly more convoluted. In particular, the behaviour of a man `m` now includes two phases, proposing and waiting.

[3] Note that the underscore is introduced to distinguish an attribute identifier from a constant.

[4] The sending predicate includes a variable name where it is written with a $ prefix.

```
man([H|T],Id) ->
 Pid = global:whereis_name(H);
 Pid ! {propose,Id},
 receive
   no -> man(T,Id)
 end.

woman(Prefs,Partner,Id) ->
 receive
   {propose,Man} ->
     Pman = global:whereis_name(Man),
     Ppart = global:whereis_name(Partner),
     case bof(Prefs,Man,Partner) of
       true ->
         Ppart ! no,
           woman(Prefs,Man,Id)
         false ->
           Pman ! no,
           woman(Prefs,Partner,Id)
     end
 end.
```

Figure 2: Classical stable marriage in Erlang.

```
man() ->
 [[H|T],Id] = aerl:getAtts([prefs,id]),
 aerl:setAtts([{partner,H},{prefs,T}]),
 to("id = this.partner") ! {propose,Id},
 from("id = this.partner"),
 receive
   no -> man()
 end.

woman() ->
 from("gender = _man"),
 receive
   {propose,Man} ->
     Partner=aerl:getAtt(partner),
     case bof(Man,Partner) of
       true ->
         to("id = this.partner") ! no,
         aerl:setAtt(partner,Man);
       false ->
         to("id = $Man") ! no
     end,
     woman()
 end.
```

Figure 3: Classical stable marriage in AErlang.

- (Proposing) man `m` extracts a tie from the top of his preference list, and sends `propose` messages to all women in this tie, say `k` women. As soon as the man receives a first `yes` message from a woman `w`, he sends her a `confirm` message and considers himself engaged to her. Note that the man in any case waits for exactly `k` reply messages from the women he contacted, rejecting any other `yes` message. However the man keeps track of those women who replied `yes`, so in case he breaks up, then he can contact them again. Apart from this, all `no` answers are ignored.
- (Waiting) after the proposing phase, a man can be either alone or engaged. In the first case, he continues to propose himself again to the women in the next tie according to his preference list. In the second case he takes no action unless he receives a break-up message from his partner, in which case he starts trying again.

The behaviour of a woman is similar to the classical case, except that she needs an extra confirmation message from a man she has decided to accept. In this case, a woman first says `yes` to the man (but without considering herself engaged to him yet), then waits for a `confirm` message. Once received such a message, she breaks the current engagement and gets the new man. In any case, a woman always keeps listening for incoming proposals. Note that the above protocol based on counting is introduced to make it possible for men to understand when it is time to move on to the next tie in the preference list.

The code implementing the above protocol between men and women is given in Fig. 4. We omit further description of the behaviour of women as it is very similar to the one discussed in Fig. 3.

Note that in order to deal with multiple receives at once, in the man code we use a variant of attribute-based send and receive. In line 4, the process sends a message to any process whose attribute `id` belongs to a list `H`. AErlang then returns a value indicating the number of receivers selected at the communication time, which is `Count`. In line 5, men use this number inside attribute-based receive which has the effect of recursively receiving `Count` messages. Note that the unique reference `Ref` is exploited to distinguish the proposing and waiting phases of the man. Furthermore, in line 14, the `update` function stores the women who said `yes` in the preference list so that in the next round a man can try again to propose himself to them.

By abstracting the SMP problem, we are able to deal with more realistic settings 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 the stable marriage can be applied to open 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 media networks, as we are going to discuss shortly.

```
man() ->
 [[H|T],Id] = aerl:getAtts([prefs,id]),
 Ref = make_ref(),
 Count = to_c("id in $H") ! {Ref,propose,Id},
 from("gender = _woman",Count),
 receive
   {Ref, yes,W} ->
      case aerl:getAtt(partner) of
           none ->
              to("id = $W") ! {confirm, Id},
              aerl:setAtt(partner,W);
           _ ->
              to("id = $W") ! {no, Id},
              update(prefs,W)
       end;
   {Ref, no, _} -> ok
 end,
 Partner = aerl:getAtt(partner),
 case Partner of
    none -> man();
     _ -> receive
           {no, Partner} ->
              aerl:setAtt(partner,none),
              man()
       end
  end

woman() ->
   [Id,Partner] = aerl:getAtts([id,partner]),
   from("gender = _man"),
   receive
     {Ref, propose,Man} ->
        case bof(Man,Partner) of
           true ->
              to("id=$Man") ! {Ref, yes, Id}
              receive
                 {confirm, Man} ->
                    to("id=$Partner")!{no,Id}
                    aerl:setAtt(partner,Man);
                 {no, Man} -> ok
              end;
           false ->
              to("id = $Man") ! {Ref, no, Id}
        end,
        woman()
   end.
```

Figure 4: SMTI in AErlang.

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 dynamically change, in fact the system is open and anonymous.

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. This application would fit very well the social media scenario with few extra attributes, such as spoken languages, age, interests, and the language one wishes to learn.

Possible attributes for one user joining the system are: (i) `language`, i.e., the language that she already knows; (ii) `interest`, i.e., the language that she would like to learn; (iii) `age`, i.e., the age of user; (iv) `hobby`, i.e., the user's hobbies which might be represented as a list, such as outdoor activities, musical or food preferences.

Interaction might be naturally expressed by the following code snippet, where users advertise their own interest by sending their proposal:

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

Another user may put a receive waiting for somebody knowing the language that she is interested in, conditioned to the matching of the hobby and only if she is not older than five years.

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

Note that in the above snippets `Language`, `Id`, and `Buddy` are variable identifiers.

The communication based on matching attributes opens to many relevant investigations. For example, one interesting question might be whether musical or food preferences somehow propagate among friends. The corresponding property to check would be to check whether the probability of any two people liking the same song is higher if they are friends.

5. AErlang Evaluation

In this section we discuss the effects of using the attribute-based programming abstractions in terms of efficiency and scalability of the resulting programming artefacts. Our prototype is publicly available[5].

The experimental results are grouped in two separate parts. In the first part, we evaluate the efficiency by comparing an Erlang program for the classical solution to the SMI problem against the corresponding AErlang program presented in Section 4. The insights are summarised in Section 5.1. In the second part, we evaluate the scalability by comparing our AErlang program for the STMI problem presented in Section 4 with respect to a state-of-the-art ad-hoc solution based on adaptive search [24]. The observations are reported in Section 5.2.

We generated multiple random input instances with the support of a tool[6] that takes three parameters as input: (p_1) size of the instance, (p_2) probability of incompleteness, (p_3) probability of ties. For our analysis we generated two classes of instances while considering instance sizes up to 8 thousands pairs of elements: (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 [24].

Initial experiments were 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.

Further experiments were performed on a computing cluster 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.

5.1 Efficiency

We ran the Erlang and AErlang programs to solve the SMI variant of stable matching while ranging the size of the instances from 1 to 8 thousands of pairs of elements and the number of cores from 1 to 12. For each combination of these, we ran ten different instances (i.e., randomly generated with [24]) ten times each. We collected the average execution times measured by only taking into account the actual computations, excluding the time spent on the initialization and the stability check on the output.

The comparison between AErlang and Erlang is shown in Figure 5, where on the x-axes report the size of instances in thousands of pairs, and on the y-axes the execution times in seconds. The main point here is that the performance overhead tends to decrease when increasing the number of cores.

[5] https://github.com/ArBITRAL/AErlang

[6] https://github.com/dannymrock/SMTI-AS-X10

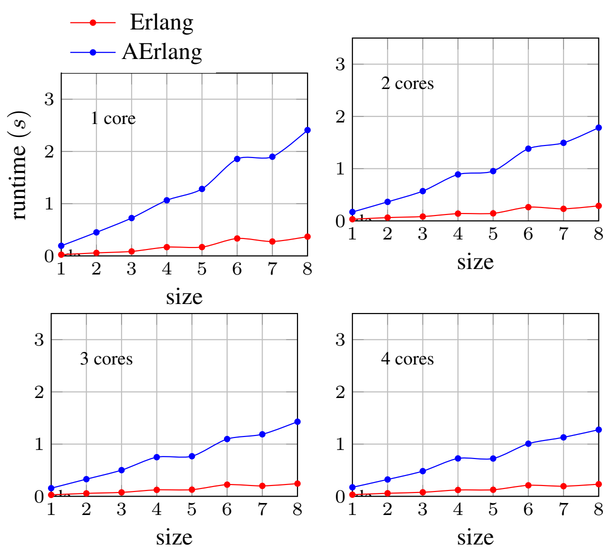

Figure 5: Efficiency: AErlang vs. Erlang (SMI)

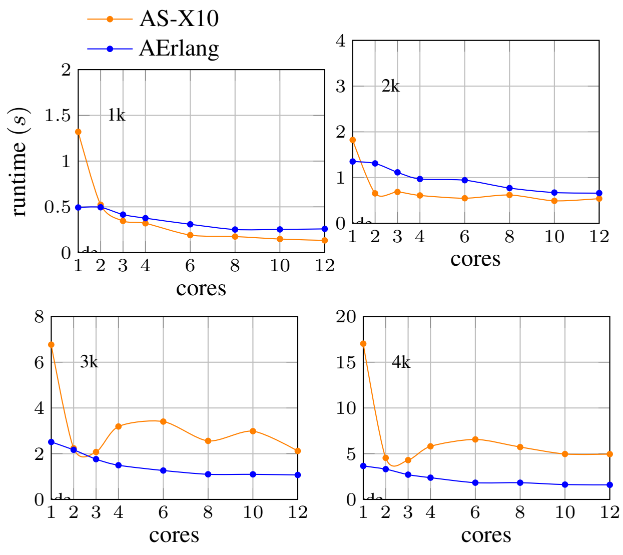

Figure 6: Scalability: AErlang vs. AS-X10 (SMTI)

Table 1 reports the AErlang vs Erlang runtime ratio worked out from the same data points. Here, columns list the instance size in thousands of pairs of elements, whereas rows enumerate the considered number of cores up to 12 cores. We observe that overall cases the ratio is always within one order of magnitude, roughly between 5 and 8. More precisely we found a minimum ratio of 4.78 with instance size equal to 6k and 4 cores, and a maximum one of 8.48 with 3k and 1 core, as highlighted in the bold entries. This suggests that the new programming abstractions introduce an acceptable performance overhead. Note that in this part of the evaluation we have not considered the AS-X10 program based on adaptive search. This program was indeed specifically conceived to take advantage of ties in the input, and therefore it performed poorly on the no-ties instances considered here, timing out on most of them at 500 seconds.

		size							
		1k	2k	3k	4k	5k	6k	7k	8k
cores	1	8.03	7.80	**8.48**	6.35	7.57	5.57	6.88	6.52
	2	5.30	5.76	6.90	6.41	6.63	5.26	6.48	6.19
	3	5.54	5.74	6.63	6.07	6.00	4.89	5.94	5.85
	4	5.41	5.48	6.23	5.94	5.75	**4.78**	5.81	5.48
	6	5.75	6.39	7.13	7.43	6.14	5.06	6.38	6.81
	8	5.35	6.41	7.37	7.72	6.57	5.80	6.63	6.40
	10	5.51	6.66	7.58	7.90	6.87	6.11	7.20	6.69
	12	5.39	6.29	7.51	7.87	6.99	6.04	7.41	6.92

Table 1: Runtime ratio AErlang vs Erlang (SMI)

5.2 Scalability

Similarly to the previous section, we ran all the experiments on the SMTI version on ten instances with ties, ten times each, while considering different variations of the number of cores and the size of instances. However, in comparison to previous section, we restrict the range of instance sizes from 1 to 5 thousands of pairs of elements due to frequent timeouts (500 seconds) of the AS-X10 STMI program beyond that point.

Figure 7 shows the experimental results for this part of the analysis. 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. Such speedup tends to increase with size, making the AErlang program more suitable to larger instances.

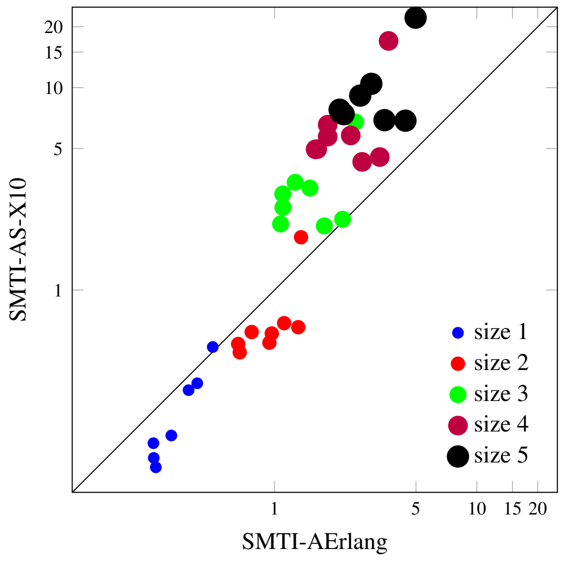

Figure 7: AErlang vs. AS-X10 (SMTI)

Similarly to Table 1, Table 2 reports the AErlang vs AS-X10 runtime ratio where columns list the size of the instances varying between 1 and 5 thousands of pairs of

		size				
		1k	2k	3k	4k	5k
cores	1	0.37	0.74	0.37	0.21	0.22
	2	0.95	2.00	0.97	0.73	0.65
	3	1.20	1.63	0.85	0.63	0.51
	4	1.18	1.59	0.47	0.41	0.29
	6	1.62	1.72	0.37	0.28	0.29
	8	1.44	1.24	0.43	0.32	0.28
	10	1.71	1.37	0.37	0.33	0.27
	12	1.95	1.22	0.51	0.32	0.30

Table 2: Runtime ratio AErlang vs AS-X10 (SMTI)

elements, whereas rows enumerate the considered number of cores. In the table it is worth to notice that in most of the cases we found a ratio below to 1, i.e., AErlang outperforms AS-X10, as highlighted in the shaded entries of Table 2.

In the previous section we have discussed that introducing the new programming abstraction introduces a reasonable performance overhead. In addition these new experimental results indeed 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.

Figure 6 compares the runtime performance of AErlang and AS-X10 for SMTI. We report on the x-axes the number of cores and on the y-axes the execution times in seconds. Note that the scales for the y-axes are different. For small instances, AS-X10 exhibits very good scalability; for larger instances, however, this does no longer hold. On the contrary, AErlang's scalability appears to be quite modest but more consistent across all the considered instance sizes.

		size				
		1k	2k	3k	4k	5k
cores	1	2.54	3.00	3.46	3.44	3.89
	2	2.91	3.60	3.81	3.73	**4.65**
	3	2.65	3.38	3.51	3.62	4.55
	4	2.18	3.00	3.09	3.28	4.15
	6	1.69	2.55	2.39	2.22	3.77
	8	1.31	1.88	1.79	1.88	2.61
	10	1.22	1.51	1.68	1.52	2.36
	12	**1.21**	1.47	1.59	1.45	2.37

Table 3: Runtime ratio with AErlang (SMI vs SMTI)

Finally, Table 3 reports the runtime ratio observed while comparing the two versions of the problem, i.e., SMI and SMTI, both implemented in AErlang. The SMTI program turns out to be slower within a relatively small ratio, roughly ranging within 1 and 5, which tends to increase with the size of the instances. The minimal and maximal values are boldfaced in the table. A reason for the performance gap is that in the SMTI AErlang program we had to introduce a more complex component interaction that relies on extra acknowledgment messages (Sect. 4) being exchanged between pairs of elements.

To deeper investigate on the modest scalability of our prototype in terms of the number of cores, we performed

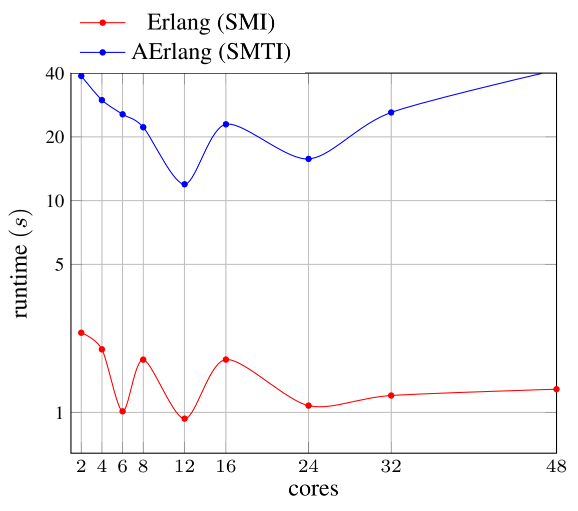

Figure 8: Scalability: AErlang (SMTI) vs. Erlang (SMI)

further experiments on the aforementioned computing cluster. 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 within 2,4,6,8,12,16,32, and 48. We ran 10 instances, 10 times each, and collected the average execution times as previously. The results are summarised in Figure 8. Here, the x-axis denotes the number of cores and the y-axis we report the runtime in seconds on a logarithmic scale. Interestingly, the pronounced fluctuations in the running times are shown for both AErlang and Erlang programs. This suggests that performance glitches within the Erlang subsystem may end up affecting our AErlang prototype too.

6. Related Work

Attribute-based communication has been explored in the context of autonomic computing by the research centered around the SCEL paradigm [28]. It has been used to model the dynamic formation of ensembles from interacting autonomic components [14]. Notably, this novel communication paradigm can also be used to model a wide range of adaptation patterns in autonomic systems [11].

To the best of our knowledge, two 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 [1]. This implementation only supports the broadcasting method discussed in Section 3. While the broadcast strategy 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 effort is jRESP [22], based instead on the SCEL paradigm, and more specifically 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 can use the communication protocol that the port supports. In this way, various communication protocols can be implemented, such as broadcast via a central server, multicast or P2P. The main difference with our approach is that we also consider strategies which filter early group of partners by exploiting updated predicates and attributes.

In the context of autonomic computing and Erlang, ContextErlang [26] has been proposed as an extension of Erlang for programming context-aware applications according to Context-Oriented Programming [19]. 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_agent*s. This is the key mechanism that ContextErlang uses for achieving dynamic software adaptation. 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.

Finally, in [25] JErlang has been presented, and it is an Erlang extension to Join-Calculus [16]. JErlang provides a receive-like join construct for synchronising multiple messages with different patterns in the mailbox of a receiving process. To further improve performance of this construct, their implementation intercepts the Erlang receive algorithm to incorporate the joins resolution mechanism, together with low-level optimizations inside Erlang's VM. On the contrary, AErlang implementation focuses on mediating message passing based on predicates with appropriate handling of process attributes, and leading to user-friendly communication primitives.

7. Conclusion

In this paper, we have been experimenting with attribute-based communication and functional-style programming.

The general benefits of using predicate-based message passing were already clear, in particular the resulting increased expressiveness that allows capturing complex process interaction using predicates. However, due to the inherent complexity of the systems under consideration, scalability and performance overhead introduced with the new communication primitives were a concern.

We have addressed those concerns by initially instantiating the novel programming abstractions on top of the Erlang functional programming language. We chose $Erlang$ for its concurrency model based on actors that fits very well with the AbC process calculus.

We have discussed the design and some implementation details of our prototype extension, $AErlang$, but we are aware that it is still far from being generally effectively usable in practice. An extensive evaluation on arbitrarily large instances that use complex predicates and frequently changing attributes would be needed to assess the overall robustness. Nevertheless, the experimental results are encouraging and address the above mentioned concerns on performance overhead and scalability.

We have evaluated our prototype in terms of performance overhead with respect to the native language by comparing the runtime performance of functional-style implementations for a known solution to a hard matching problem. Experiments have shown that the overhead resulting from using the new communication primitives is acceptable, and our prototype successfully preserves $Erlang$'s 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 $X10$ [24] 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 [8, 10, 23]. 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 $X10$.

As mentioned, in order to further improve robustness of our prototype we will need an in-depth performance evaluation, to introduce more efficient evaluation strategies. Firstly, we need to understand whether the large size of the system stresses the underlying scheduling mechanisms. Secondly, we have to assess the cost of predicates handling. 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. It could be beneficial to devise different strategies that by relying on a fine-grained classification of attributes allows us, e.g., to handle differently the frequently-changing attributes and the totally static ones.

Finally, we remark that this is our first experiments of embedding AbC primitive in a fully-fledged language such as Erlang. We plan to introduce attribute-based communication in other concurrency languages, e.g., Go[7] and Scala[8] as we think that their extension can also be done in a similar way to Erlang. This wider experimentation across different programming languages will allow us to deeply investigate the effectiveness of attribute-based communication.

References

[1] Abacus: A run-time environment of the abc calculus. http://lazkany.github.io/AbC.

[2] Paper to appear. title of the paper and authors are intentionally not reported due to double blind process.

[3] Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.

[4] Alrahman, Y. A., De Nicola, R., and Loreti, M. (2016a). On the power of attribute-based communication. In *IFIP International Conference on Formal Techniques for Distributed Objects, Components, and Systems FORTE*, pages 1–18. Full technical report can be found on `http://arxiv.org/abs/1602.05635`.

[5] Alrahman, Y. A., De Nicola, R., and Loreti, M. (2016b). Programming of CAS systems by relying on attribute-based communication. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA*. to appear.

[6] Armstrong, J. (2007). *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf.

[7] Armstrong, J. (2010). Erlang. *Commun. ACM*, 53(9):68–75.

[8] Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., and Venetis, I. E. (2012). A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 33–42. ACM.

[9] Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., and Wicklund, G. (1999). AXD 301: A new generation ATM switching system. *Computer Networks*, 31(6):559–582.

[10] Boudeville, O., Cesarini, F., Chechina, N., Lundin, K., Papaspyrou, N., Sagonas, K., Thompson, S., Trinder, P., and Wiger, U. (2012). Release: a high-level paradigm for reliable large-scale server software. In *International Symposium on Trends in Functional Programming*, pages 263–278. Springer.

[11] Cesari, L., De Nicola, R., Pugliese, R., Puviani, M., Tiezzi, F., and Zambonelli, F. (2013). Formalising adaptation patterns for autonomic ensembles. In *Formal Aspects of Component Software*, volume 8348, pages 100–118.

[12] Cesarini, F. and Vinoski, S. (2015). *Designing for Scalability with Erlang/OTP*. O'Reilly.

[13] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM.

[14] De Nicola, R., Loreti, M., Pugliese, R., and Tiezzi, F. (2014). A formal approach to autonomic systems programming: The

[7] https://golang.org/

[8] http://www.scala-lang.org/

SCEL language. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):7.

[15] Fontoura, M., Sadanandan, S., Shanmugasundaram, J., Vassilvitski, S., Vee, E., Venkatesan, S., and Zien, J. (2010). Efficiently evaluating complex boolean expressions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 3–14. ACM.

[16] Fournet, C. and Gonthier, G. (1996). The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385. ACM.

[17] Gale, D. and Shapley, L. S. (1962). College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15.

[18] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *International joint conference on Artificial intelligence*, pages 235–245.

[19] Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology*, 7(3).

[20] Hu, Z., Hughes, J., and Wang, M. (2015). How functional programming mattered. *National Science Review*, 2(3):349–370.

[21] Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

[22] jRESP. Java Runtime Environment for SCEL Programs. `http://jresp.sourceforge.net/`.

[23] Klaftenegger, D., Sagonas, K., and Winblad, K. (2013). On the scalability of the erlang term storage. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pages 15–26. ACM.

[24] Munera, D., Diaz, D., Abreu, S., Rossi, F., Saraswat, V., and Codognet, P. (2015). Solving hard stable matching problems via local search and cooperative parallelization. In *29th AAAI Conference on Artificial Intelligence*.

[25] Plociniczak, H. and Eisenbach, S. (2010). JErlang: Erlang with joins. In *International Conference on Coordination Languages and Models*, pages 61–75.

[26] Salvaneschi, G., Ghezzi, C., and Pradella, M. (2015). ContextErlang: A language for distributed context-aware self-adaptive applications. *Science of Computer Programming*, 102:20–43.

[27] Thompson, S. and Cesarini, F. (2009). Erlang programming: A concurrent approach to software development.

[28] Wirsing, M., Hölzl, M., Koch, N., and Mayer, P. (2015). *Software Engineering for Collective Autonomic Systems: The ASCENS Approach*, volume 8998 of *Lecture Notes in Computer Science*. Springer.