Static analysis of communications for Erlang

Fabien Dagnat
Laboratoire Informatique des Télécommunication
ENST de Bretagne, Technopôle Brest Iroise, BP 832
29285 Brest, France
Fabien.Dagnat@enst-bretagne.fr

Marc Pantel
Institut de Recherche en Informatique de Toulouse
LIMA / ENSEEIHT, 2 rue Camichel
31071 Toulouse, France
Marc.Pantel@enseeiht.fr

ABSTRACT

In this paper, we present an insight of the two major contributions of works made to build a static analyzer of ERLANG programs. First, we introduce a general framework based on a process calculus (the *configurations*). This formalism describes concurrent aspects and abstracts functional ones. Obtaining the ERLANG semantics is then just instantiating this framework with an adequate functional setting. The second contribution is a sophisticated type system for ERLANG. This type system infers types and subtyping constraints for a program and ensures that the collected constraints have at least one solution. This system detects usual functional errors but also some of the communication errors. More precisely, for each process, it cumulates all received messages and all handled messages and ensures that the first is included in the second. To do this, it borrows concepts to the object (or record) usual typing in ML.

1. INTRODUCTION

The development of telecommunications industry and the generalization of network use bring concurrent, distributed and mobile computing into the limelight. In that context, programming is a hard task and, generally, the resulting applications contain many more *bugs* than usual sequential centralized software. Indeed, the indeterminism resulting from the unreliability of networks and the size of the code of such applications makes it difficult to validate any distributed functionality using informal approaches. Our work focuses on using static analysis, a kind of formal methods to ease development.

As Erlang software are mainly used in telecommunication equipment that do not tolerate failure, their development must be certified. More precisely every step toward the final application must be *validated* (ideally automatically). Our aim is to participate to this hard task, by building static analysis of communications using type inference techniques.

To give an abstract model to ERLANG programs, we use the actor model developed by Agha in [1]. It is based on a network of autonomous and cooperative agents (called actors and similar to ERLANG processes), which *encapsulate* data and programs. They communicate using an *asynchronous point to point* protocol and store each received message in a mailbox. When idle, an actor handles the first message it can in its mailbox. Besides those conventions (which are also true for concurrent objects), an actor can dynamically (at run-time) change its interface. This property allows to modify the set of messages an actor can handle, yielding a more accurate and widely usable programming model. For example, it can give an abstract model to applets and dynamic code loading.

In a first approach, we defined type systems for the CAP calculus described in [8], a primitive actor calculus derived from asynchronous π-calculus and Cardelli's Calculus of Primitive Objects. Two type systems were developed. The first one [9], based on usual object type abstractions, catches all usual functional and communication errors (erroneous parameters) but only a subset of messages which will never be handled. The second [7], detects all (safety) messages not understood but requires a much more complex type abstraction and a new programming discipline. These systems were proved to be correct. In order to validate their practical use, the need for a programming language implementation arose. In a first approach, we developed a lab language ML-ACT integrating *à la ML* programming with actor primitives and including a sophisticated type system extending the previous work on CAP (see [11]). Then, we studied ERLANG, as it appears that, thought its functional aspects have a strongly different semantics (and typing) than ML-ACT one's, their concurrent semantics and typing were similar. Therefore, we developed a framework abstracting the parts of both languages having semantics (and typing) differences (for example, functional aspects or mailbox semantics). It became possible to build systematically the semantics, the typing and some properties about the typing, once provided the functional setting. Furthermore, this functional setting can use a well known classical one. For example, ML-ACT use the ML functional semantics and typing.

This article gives an introduction to this abstraction and its application to ERLANG. The first section provides a better insight of the form of communication errors we wish to detect and the ones our system captures. Then, we introduce a simplified version of ERLANG and its formal semantics based on configurations, an asynchronous π-calculus like process algebra. Then, we define our type system and illustrate its use on examples. Finally, we discuss scaling this system to the full language and some possible extensions to our work.

2. COMMUNICATION ERRORS

In an usual concurrent setting, a process `P` may receive a message `m` (`P ! m`, in ERLANG). Supposing `P` is idle, there are two possibilities, either `P` can handle `m` or it cannot. Our works focus on the early detection of requests that may not be handled (the second case). This problem is related to the *method not understood* errors of object oriented programming. In the actor context, a message that may not be understood by its receiver is called an *orphan*.

Typed object oriented languages determine the set of methods an object `P` understands (**typeof**(`P`)) and ensures that each method invocation `P.m` is correct by verifying that `m` is part of the type of `P` (`m` $\in$ **typeof**(`P`)). Furthermore, as the type of an object does not change, the verification can be done when the method is invoked. Adapting this technic to ERLANG (`P` becoming a process and `P.m` becoming `P!m`) raises two problems leading to a much more complex typing: a) the computation of the set of messages a process can handle is dynamic and more complex and b) as the time between sending a message and its reception by its target may be important (the message may travel through large networks), the verification must be done upon reception.

The usual approach for actor languages is to **dynamically** check for *message not understood* errors. A process knows the messages it can (immediately) handle and if a received message does not conform to this interface, it raises a message not understood error (see the initial actor model [1] or the Vasconcelos and Tokoro object calculus [26]). But this approach reduces consequently the set of programs that one may build. In fact, the programmer must adopt a sort of synchronous programming discipline to be sure that messages arrive in *right* states. We think that this strategy is too restrictive. For example, consider a printer device that has two states: `working` (it accepts printing requests) and `stopped` (it waits for initialization). A client must wait that an initialization message has been sent to the printer before printing. It would be much more flexible to enqueue all requests received when the printer is `stopped` and to process all pending requests when it is initialized (possibly independently by another process) which is the usual behavior of unices print spoolers.

The second and opposite approach never rejects a message. When a process receives a message that it cannot handle, it silently enqueues it. Notice that, in this context, a message may stay indefinitely in a mailbox (their size is unbound). This semantics has been chosen by the blue calculus [4], the join calculus [14] and ERLANG.

We believe that a combination of both approaches may be much more appropriate. Such a system would reject programs that contains *message* ***never*** *understood* and would accept all other messages warning the programmer that they may never be handled. To achieve this goal, we use a powerful behavioral[1] type system to enforce the rejection of such messages. Our type system detects all messages that are not in the set of messages the receiver may handle during its execution. This means that **typeof**(`P`) cumulates all the `receive` that `P` could execute. To do this the system must follow the flow of functions called by `P`. It is clear that, in general, our analysis will answer $\top$ (top) to express the fact that a process may assume an externally defined receive and therefore understands virtually everything. But, we think that the results are generally already helpful and we are working on extending our techniques to those open programs as will be discussed later.

[1]By opposition with a more usual class name type system as in C++ or Java.

For example, a process `P` executing the first function of the program below (`ping`) has a type containing `ping`, `change` and all messages accepted by all possible behaviors `F`. This means that sending a message `{change, pong}` to `P` adds `pong` to the type of `P`.

```
ping() -> receive ping -> ping();
                  {change, F} -> apply(F,[])
          end.
pong() -> receive pong -> pong() end.
```

3. A SIMPLIFIED VERSION OF ERLANG

Following a common use in the definition of static and dynamic semantics, we simplify the ERLANG language by suppressing syntactic sugar and ignoring constructions that are typed orthogonally to our work (for example, exceptions, lists or records). Furthermore, we do not address the semantics of the real time part of the language which is complex but do not add any specific problem to the type system. An effort has been made to define precisely a small (but still too big) language named CORE ERLANG ([5] or [6]). Therefore, we use a smaller version of the language named μErlang:

$$
\begin{array}{lll}
prg & ::= & c\texttt{;}\ldots\texttt{;}c\texttt{.} \;\mid\; c\texttt{;}\ldots\texttt{;}c\texttt{.}\; prg \\
c & ::= & s\texttt{(}p\texttt{,}\ldots\texttt{,}p\texttt{)} \texttt{ -> } e \\
p & ::= & \texttt{\_} \mid V \mid s \mid i \mid \texttt{\{}p\texttt{,}\ldots\texttt{,}p\texttt{\}} \\
e & ::= & V \mid s \mid i \mid \texttt{\{}e\texttt{,}\ldots\texttt{,}e\texttt{\}} \mid \texttt{(}e\texttt{)} \mid e\texttt{,}e \mid e\texttt{!}e \\
 & \mid & e\texttt{(}e\texttt{,}\ldots\texttt{,}e\texttt{)} \mid \texttt{case } e \texttt{ of } f \texttt{ end} \mid \texttt{receive } f \texttt{ end} \\
f & ::= & p \texttt{ -> } e \mid p \texttt{ -> } e\texttt{;}f
\end{array}
$$

A μErlang program is a set of function definitions including a function named `main`. This main function is launched to start the execution of the program. The rest of the language is very close to ERLANG. Each function is composed of clauses separated by semi-colons and terminated by a dot. All clauses (s`(`p`,`...`,`p`) ->` e) must refer to the same function name s and have the same arity. Notice that this language does include guards to simplify the semantics and the type system for this paper. A pattern may be a joker (always succeeding), a variable V (always succeeding and binding the variable[2]), an atom s, an integer i or a tuple. An expression may be any of those values and add parentheses, sequencing (`,`), message sending (`!`), function call, choice (`case`) and message handling operation (`receive`). The choice (resp. the receive operation) matches an expression (resp. the mailbox of the current process) using a set of filters composed of a pattern and an expression (f is named **interface**). Finally, some atoms represents built-in functions, as for example, `spawn` and `self`.

Notice that as CORE ERLANG, we adopt lexical scoping of variables to ease the presentation. Our prototype uses ERLANG strategy mixing dynamic and lexical scoping. Therefore, the real system uses systematically an input and an output environment for each expression. Again for sake of simplicity, μErlang does not include lists that are replaced in application and spawning by tuples.

[2]This is not true for ERLANG, but our system can easily adopt ERLANG policy.

4. FORMAL SEMANTICS OF ERLANG

Our work focuses on static analysis and more precisely on typing. In order to prove the correctness of our type system, we need a formal semantics of ERLANG. To our knowledge, few works have addressed such a hard task. Indeed, as ERLANG is a full fledge functional, concurrent, distributed and mobile language, its semantics is complex. Some efforts have been made to give an informal, but clear and systematic description of its semantics ([3] and [6]). But, this is not sufficient to build and prove some static verification system. It seems that only two papers ([12] and [15]) try to build such a formal semantics. These two papers define two *Labeled Transition System* that does not suit our need (proving the correctness of a type system). Inspired by those approaches and our previous works on semantics for actors, we built our own formal semantics by instanciating a general framework called *configurations* previously build on a lab language extending ML to actors (ML-ACT). This framework defines a general syntax for concurrent actions and abstracts (in the sense of taking as parameter) the functional part of the studied language. With this approach, we can reuse existing semantics and typing from the functional world. The μErlang semantics is obtained by instantiating this framework with an adequate functional semantics.

We are not going to give all the formal definitions and justifications of this model that may be found in [10]. We are only going to give insights on configurations to deduce the μErlang semantics. Most rules are given in appendix for the interested reader.

Configuration

A configuration is a term that represents a concurrent system at a given time. Its definition is parameterized by three sets : the **name** set $a \in \mathbb{A}$, the message set $m \in \mathcal{M}ess$ and the expression set $e \in \mathcal{E}xp$ with $\mathbb{A} \subset \mathcal{E}xp$ and $\mathcal{M}ess \subset \mathcal{E}xp$. The set of configurations noted W is built from the following grammar:

$$\begin{array}{rcl} w & ::= & \epsilon \mid \mathbf{Err} \mid \nu a.w \mid w \parallel w \mid a \triangleleft m \mid \alpha \triangleright e \\ \alpha & ::= & \star \mid \langle a \,|\, \widetilde{m} \rangle \end{array}$$

A configuration looks like a π-calculus term with a send operation, noted $a \triangleleft m$ (a is the receiver and m the message), and a process, noted $\alpha \triangleright e$ (α is the identity and e is the executed expression). The identity of a process is either unspecified $\star$ to model toplevel computations[3] or, $\langle a \,|\, \widetilde{m} \rangle$ a pair composed of a name (*pid* in ERLANG tradition) and a mailbox (the tilde notation denotes sequence). As it is usual in process calculi, we use a name binder ν to simulate the name creation and suppose that the corresponding notion of free names and substitution are defined.

In the context of μErlang, $\mathcal{E}xp$ represents the syntax introduced in the previous section, addresses are built automatically when the built-in function `spawn` is called and a message can be any value (atom, integer or tuple).

A congruence is defined to state which configurations are equivalents:

- $(W, \parallel, \epsilon)$ is a commutative monoid, the order of subconfigurations is not important and we can suppress all occurrence of ϵ.
- $w \parallel \mathbf{Err} \equiv \mathbf{Err}$ and $\alpha \triangleright \mathbf{Err} \equiv \mathbf{Err}$, errors are propagated until the program evaluation stops.
- $\nu a.w \equiv w$ if a is not free in w, $\nu a.w \equiv \nu b.[b/a]w$ if b is not free in w and $\nu a_1.\nu a_2.w \equiv \nu a_2.\nu a_1.w$; those three usual properties allow to forget the bindings of unused names, to rename a bounded name and to modify the order of restrictions.
- the restriction rule, $\nu a.w_1 \parallel w_2 \equiv \nu a.(w_1 \parallel w_2)$ if a is not free in w_2, allows to enlarge the scoping of a name. Combined with the previous rule, it enables (up to a renaming of a in w_1) to extend the scoping and to simulate name propagation in the medium.
- $\star \triangleright v \equiv \epsilon$ and $\nu a.(\langle a \,|\, \varnothing \rangle \triangleright v) \equiv \epsilon$ if v is a value (it cannot be reduced) ; therefore, a global computation (or a process) which reduce to a value can be destroyed by a garbage collector. Notice that the process must have an empty mailbox and be inaccessible to the outside world.

Notice that it is possible to add a rule to express the fact that a stopped process waiting for a message, that do not understand any of its mailbox messages and is no more accessible from outside is an error. But, as our type system cannot capture all such messages (for example in a deadlock case), we cannot prove its correctness with this rule.

The appendix contains all the configuration reduction rules. Let us discuss only original rules.

As introduced in the second section of this paper, we try to detect communication errors. To define those errors more precisely, they are introduced in the semantics of configurations. Therefore, when a process receives a message, it can accept it (and put it in its mailbox) or reject it by raising an error:

$$\langle a \,|\, \widetilde{m} \rangle \triangleright e \parallel a \triangleleft m \longrightarrow \begin{cases} \langle a \,|\, m\ \widetilde{m} \rangle \triangleright e & \text{if } \mathcal{P}(m, e) \\ \mathbf{Err} & \text{else} \end{cases}$$

To abstract the choice of reaction, a (communication) *potential* $\mathcal{P}(m, e)$ is defined. This predicate approximates e to determine whether m may be understood or not. This allows the semantics of our framework to behave differently toward such messages. It is possible, for example, to code usual ERLANG semantics with a predicate always true. In the next section on typing, we will discuss more deeply this subject.

Our general semantics includes a rule to specify the interaction between functional and concurrent reduction:

$$\frac{a \notin \mathcal{FN}(\alpha \triangleright e) \qquad a \vdash \alpha,\ e \xrightarrow{w}_e \alpha',\ e'}{\alpha \triangleright e \longrightarrow \nu a.(\alpha' \triangleright e' \parallel w)}$$

Where, we suppose that the functional reduction have the given shape with a being a fresh name ($a \notin \mathcal{FN}(\alpha \triangleright e)$) that may be used during the expression evaluation and w being a configuration describing the concurrent effect of the functional reduction step. In the rest of the paper, if the label of such a reduction is ϵ, it is omitted. Notice that if a is unused, the third congruence rule enable to forget its binding.

[3] Those expressions cannot access `receive` or `self`.

Functional reduction

A μErlang program is a set of function definitions and its execution corresponds to the reduction of the body of the `main` function in a context where all the other functions are defined. By consequence, the first step of the functional semantics builds the function environment (noted $\mathcal{F}$). This process will not be described here, its result is an environment associating an atom and an arity to the body (all the pattern matching converted to a tuple matching) of the corresponding function. For example:

$$\left\{\begin{matrix}\texttt{f(}p_1\texttt{,}p_2\texttt{) ->}\ e_1\texttt{;}\\ \texttt{f(}p_3\texttt{,}p_4\texttt{) ->}\ e_2\texttt{.}\end{matrix}\right. \quad \text{produces} \quad (\texttt{f},2) \mapsto \left\{\begin{matrix}\texttt{\{}p_1\texttt{,}p_2\texttt{\} ->}\ e_1\texttt{;}\\ \texttt{\{}p_3\texttt{,}p_4\texttt{\} ->}\ e_2\texttt{.}\end{matrix}\right\}$$

To simplify our presentation this set is abstracted and supposed to be accessible in all rules. This could be done by tagging each expression with this environment: $e_{\mathcal{F}}$ and by propagating it during reduction.

Functional reduction uses the classic notion of evaluation context. A context noted C[] is an expression with a hole marking the sub-expression subject of the current reduction step. The reduction $\mathrm{C}[e_1] \longrightarrow_e \mathrm{C}[e_2]$ reduce the expression e_1 and replace it by the result e_2. The evaluation context grammar is also given in the appendix, it expresses the fact that the order of evaluation is undefined when evaluating a tuple, a message sending or an application. On the contrary, evaluation of a sequence (resp. a choice) starts with the first expression (resp. the tested value). In addition we suppose that an error cause the end of the evaluation process: $\mathrm{C}[\mathbf{Err}] \triangleq \mathbf{Err}$.

Variables once defined have their values propagated by a substitution noted σ that we will not describe here. The matching operator / uses a function **match** to compare a pattern and a value and build the substitution of the variables in the pattern by their corresponding values. This function either returns a substitution or fails. It tries to match the first filter $p \to e$. If $\mathbf{match}(p,v)$ returns σ, / returns $\sigma(e)$. Else, if it did not matched, the process continue with the remaining filters. At the end, if none of the filter have matched, we get an error.

Purely functional evaluation is classic. The most original rules concerns application:

$$a \vdash \alpha,\ \mathrm{C}[v(v_1,...,v_n)] \longrightarrow_e \alpha,\ \begin{cases}\mathbf{Err} & \text{if } (v,n) \notin dom(\mathcal{F})\\ \mathrm{C}[\{v_1,...,v_n\}/\mathcal{F}(v,n)]\end{cases}$$

The called function must be in the current function environment ($\mathcal{F}$). The result corresponds to the matching of its body with the tuple of actual arguments. This rule suppose that the expression describing the function must reduce to a valid atom and therefore, it extends slightly ERLANG semantics.

The functional actions that are connected with concurrent behavior have an original form and must be explained:

- Sending a message impose that the first argument is a name, returns the sent value and is labeled by the configuration sending term:

$$a \vdash \alpha,\ \mathrm{C}[v_1\,!\,v_2] \xrightarrow{v_1 \triangleleft v_2}_e \alpha,\ \begin{cases}\mathbf{Err} & \text{if } v_1 \notin \mathbb{A}\\ \mathrm{C}[v_2]\end{cases}$$

- Spawning impose that its second argument is a tuple, returns the name (guaranteed to be fresh by concurrent reduction) of the future process and is labeled by the configuration describing the newly created process.This is only rules where the fresh name is used.

$$a \vdash \alpha,\ \mathrm{C}[\texttt{spawn}(v,v_1,...,v_n)] \xrightarrow{\langle a\,|\,\varnothing\rangle \triangleright v(v_1,...,v_n)}_e \alpha,\ \mathrm{C}[a]$$

- A call to the built-in function `self` must be done in a process and is replaced by the name of the current process:

$$a \vdash \langle a'\,|\,\widetilde{m}\rangle,\ \mathrm{C}[\texttt{self}()] \longrightarrow_e \langle a'\,|\,\widetilde{m}\rangle,\ \mathrm{C}[a']$$

- Accessing the mailbox is similar to the choice except that the order of matching is different. The process try first to match each message with the first pattern and try next patterns only if none of the mailbox messages successfully matched the first pattern. For this we use a function **matchmailbox** that returns the resulting mailbox and the reaction. Notice that if the mailbox is empty no reduction can take place and by consequence the process is stopped (until a message reaches its mailbox).

$$a \vdash \langle a'\,|\,\widetilde{m}\rangle,\ \mathrm{C}[\texttt{receive}\ f\ \texttt{end}] \longrightarrow_e \langle a'\,|\,\widetilde{m}'\rangle,\ \mathrm{C}[e]$$
$$\text{where } \mathbf{matchmailbox}(f,\widetilde{m}) = \widetilde{m}',\ e$$

5. TYPING μErlang

When building a type system to statically detect errors in programs. The first thing to do is to define precisely what kind of errors, we want to avoid. In a concurrent setting, two families of errors arise: functional errors and concurrent errors. The former family is usual in the sequential world and correspond to the erroneous use of a value (for example, using an undefined variable or using 1 as a function). The latter is rather unusual and has been described in details in the section 2.

A type system can provide several level of precision. Two prototypes have already been built for ERLANG (see [17] and [16]) that concentrates on typing purely functional computation by simplifying the language semantics. Our ambition is to build a more useful system for ERLANG programs that also analyzes concurrent parts. As we use similar technics for collecting and solving constraints, our work may be considered as an extension of those systems.

Type inference and Constraints

Our system allows the synthesis of the types of every program entity without requiring any type annotation from the programmer. To do this, a fresh type variable is associated with each node of the syntactic tree of the program and constraints between those variables are collected. At the end of this collect phase, a resolution tool determines whether the constraint set has solutions. If this is the case, the program is declared well-typed. The schema of figure 1 describes this process.

To type functions and give them widely usable types, ML uses parametric polymorphism. For example, `map` has the type $\forall\alpha,\beta\quad(\alpha \to \beta) \to \alpha\ list \to \beta\ list$ meaning that it can be used with any type α and β. We advocate that in

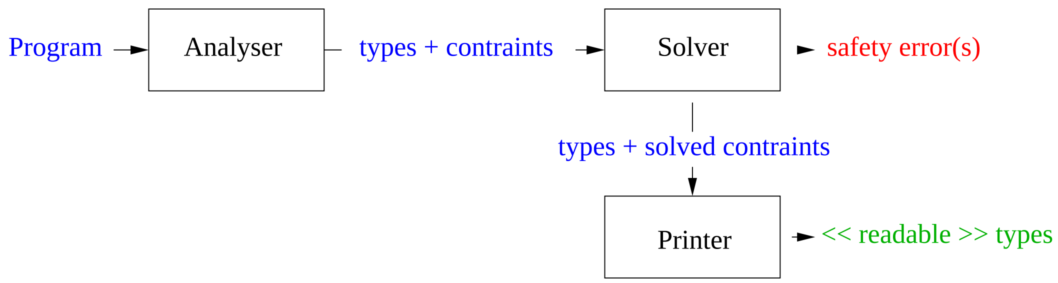

Figure 1: The analyzer schema

the concurrent context, this form of polymorphism becomes too restricting. Our system adopts inclusion polymorphism that intuitively means that the system ensures the correctness only for all values used in the program as real arguments (that is finite intersections rather than infinite ones). Therefore, in our context, we use the *subtyping* relation. A type t_1 being a subtype of a type t_2 ($t_1 \sqsubseteq t_2$) if a value of type t_1 may be used (safely) where a value of type t_2 is required. For ERLANG, the main use of subtyping is on process type: a process that understands more messages and sends itself less messages than another process, can replace this one. Typing an expression e under assumptions A will produce a type t and a subtyping constraint set C: $A \vdash e : t,\ C$, this deduction being valid only if C has at least one solution.

Notice that usual ML type system such as SML or Ocaml can be viewed as following the same process collecting equality constraints. But, when subtyping is needed (as for ERLANG), the constraints become complex and their resolution must use sophisticated and powerful graph algorithm. We refer the interested reader to the works of Pottier [19] or Fänhdrich [13]. Indeed, a constraint set is viewed as a graph where type variables are nodes (with their upper and lower bounds) and subtyping relation defines the edges.

The type of `map` becomes $(\alpha \to \beta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$ and each application with an argument of type t_1 and another of type t_2 produces the constraint set $\{t_1 \sqsubseteq \alpha \to \beta,\ t_2 \sqsubseteq \alpha\ \mathit{list},\ \beta\ \mathit{list} \sqsubseteq t_r\}$ where t_r is the resulting type. This strategy collects all possible argument types and ensures that they can all be used safely:

$$\{\bigsqcup_i t_1^i \sqsubseteq \alpha \to \beta,\ \bigsqcup_i t_2^i \sqsubseteq \alpha\ \mathit{list},\ \beta\ \mathit{list} \sqsubseteq \bigsqcap_i t_r^i\}$$

Potential and Errors

Before going on, let us look at the example below to precise some vocabulary:

```
state1(V) ->
  receive
    {add,V1} -> state1(V1 + V);
    {change,V1} -> state2(V,V1)
  end.
state2(V1,V2) ->
  receive
    {add,V3,V4} -> state2(V1 + V3, V2 + V4);
    {mute,F} -> F()
  end.
state3() ->
  receive
    kill -> true
  end.

main() ->
  case (spawn(state1,1)) of
    P -> P ! {add,1,3}, P ! kill,
         P ! {change,11}, P ! {mute,state3}
  end.
```

A function may contain two forms of interfaces (the filters f of a `receive` f `end`). One called *immediate* that is present in the body of the function or in the body of another called function ignoring received datas (in messages). And the second category corresponds to interfaces received via messages. This notion is extended to processes, the set of immediate interfaces of a process being the set of immediate interfaces of its initializing function. In the example, `state1` calls `state2` and itself and `state2` only calls itself. By consequences, the immediate interfaces set of `P` is:

`{{add,V1} {change,V1} {add,V3,V4} {mute,F}}`

The immediate interfaces may be viewed as the static automaton describing our process and the others as some dynamic part (in the exemple, `kill`).

Our type system captures all orphans that leads to error (in the semantics) using the potential introduced in the previous section. It is possible to give a predicate that collects all immediate interfaces (we refer the interested reader to [10]). Such a potential would approximates the previous set (keeping only labels) and would be defined by:

$$\mathcal{P}(m,e) \triangleq (\mathbf{label}(m) \in \{\texttt{add change mute}\}) \qquad (*)$$

Furthermore, as we do not want to raise an error and forbid the sending of the message `kill`, the potential of a processe calling a received function accepts anything. The real potential of `P` is then an *open potential*: $\mathcal{P}(m,e) \triangleq \mathit{true}$. In fact, the potential defined in $(*)$ would correspond to the same process if we change `state2`'s second filter body (the `mute` reaction) to any code not calling `F`.

Building the rules for such a system is already complex and does not capture all errors that our type system detects. Indeed, if in the example, we send a message `sub` to `P`, it is not rejected because the potential of `P` is opened. Building a more precise predicate (with respect to the captured errors) is hard and in fact corresponds to a slight simplification of the type inference. By consequence, we will not give precise definition of the potential predicate and one can view it as a simplification of the type. Each atom sent in `mute` message is collected and its potential is added to the potential of `P` which becomes:

$$\mathcal{P}(m,e) \triangleq (\mathbf{label}(m) \in \{\texttt{add change mute kill}\})$$

The message `kill` is not declared orphan but the message `sub` causes a type error (it raises a dynamic error if not rejected).

We are currently devising a new definition of errors based on a dedicated arborescent temporal logic (see [25]). However, this approach currently only handle immediate interfaces.

Message and Process Types

An automatic analysis of the ERLANG compiler code, its standard libraries and programs freely available on internet[4] revealed that sent messages and receive interfaces are mainly tuples where one element is an atom. This atom plays the role of a label for messages. Furthermore rule 5.7 from [27] states that all messages should be tagged. Following the pioneer work of [17], we impose to all programs this precept. Notice that the only (less rare) exceptions are the use of jokers or variables to delegate the treatment of the message to a choice instruction or to another process. These two uses do not go against our precept since they just serve as forwarder. Finally, a program not following this principle may easily be adapted manually.

Those labels play a role similar to those of record label in ML or of method names in objects (for example). We borrow the *row* technology, used to type records, to approximate interfaces. Rows are now frequently used for static analysis in ML world (see for example, exception analysis [18] or object typing in Ocaml [20]). In our context, a process type is a row, which is a partial function from labels to pair of types describing arguments the message contains. The first one describes received messages content and the second handled messages content. Indeed, the originality of our types is the fact that they contain both received and handled messages in the type of a process. A process receiving messages labeled m_1 containing datas of type T_1 and handling it with values of type T_2 will have the following type: $@\{m_1 : (T_1,\, T_2),\ i\}$. The (row) variable i expresses the fact that the type of the process is only partially known. The conversion from a tuple type T to a message type $\widehat{T}$ (if it is sent) or $\overline{T}$ (if it is handled) is done in a lazy way and is defined in the appendix. Either the system knows the form of the type and converts it, or its structure is unknown and the system waits. A message reduced to an atom s has the type s and correspond to the message type $\{s : (\mathit{unit},\, \top)\}$ or to $\{s : (\bot,\, \mathit{unit})\}$. Meaning respectively that it is a sent message (the handling part is meaningless[5]) or a handled message (the received part is meaningless). The conversion of tuple message is similar. In the paper [17], the conversion was done for all tuples but we think that this is not really necessary. Back to our example, the process `P` has the following type if α and i are variables:

$$T_{\texttt{P}} \triangleq @\{\texttt{add} : (1{\times}3,\ \mathit{int} \sqcup (\mathit{int}{\times}\mathit{int})),\ \texttt{change} : (11,\ \mathit{int}),\ \texttt{mute} : (\texttt{state3},\ T),\ \texttt{kill} : (\mathit{unit},\ \alpha),\ i\}$$

Where T is the type of the function `F` taken as parameter. Notice that the unknown part i is related to the type T.

The correctness of the system is ensured by generating for each spawn process a fresh interface type i verifying $\diamond i$. This predicate is true if each received message is understood and is mathematically defined by:

$$\diamond\{m_i : (T_i, T_i')\}_{i\in I} \triangleq \forall i \in I\ \ T_i \sqsubseteq T_i'$$

Applied on previous type $T_{\texttt{P}}$, we get:

$$\{1{\times}3 \sqsubseteq \mathit{int} \sqcup (\mathit{int}{\times}\mathit{int}),\ 11 \sqsubseteq \mathit{int},\ \texttt{state3} \sqsubseteq T,\ \mathit{unit} \sqsubseteq \alpha\}$$

We have not yet defined subtyping but intuitively, one can see that the two first constraints are trivial. The complete is discussed resolution after the presentation of types and subtyping.

Types and Subtyping

In ERLANG, one of the difficulties, is that being untyped, an expression may evaluate to values of really different structures (for example, a boolean and a function). Therefore, the type language must include a notion of union $t_1 \sqcup t_2$ meaning that a value of this type may be of type t_1 or t_2. Moreover to get sufficient precision, each constant has its own type (for example, 1 is of type 1 subtype of the integer *int*).

In ERLANG, any expression can execute a `receive` (*i.e*, access the mailbox of the current process). Therefore, the system use an indirect effect calculus inspired by [24] to collect, in the type of `self`, all interfaces matched against the mailbox. This effect is then included in the type of a function. When a process is spawned the effect of its initial function is added to the process type. In our example, `state3` has the following function type where the effect is the superscript of the arrow:

$$\mathit{unit} \xrightarrow{\{\texttt{kill} : (\bot,\, \mathit{unit})\}} \texttt{true}$$

The language of types needed for μErlang is built by the following grammar:

$$
\begin{array}{rcll}
T & ::= & \bot \mid \top \mid t \mid T \sqcup T \mid T \sqcap T & \\
 & \mid & i \mid \mathit{int} & \text{integers} \\
 & \mid & s \mid \mathit{atom} & \text{atoms} \\
 & \mid & \mathit{unit} \mid T \times \ldots \times T \mid \mathit{tuple} & \text{tuples} \\
 & \mid & T \xrightarrow{I} T & \text{functions} \\
 & \mid & @I & \text{processes} \\
I & ::= & \{\} \mid \top_I \mid i \mid \{m : (T,\, T),\ I\} & \text{interfaces type}
\end{array}
$$

Subtyping is defined in the formula appendix, only three rules are unusual:

- Process types are contravariant because a process may replace another one only if its interface is larger, $@I \sqsubseteq @I'$ is equivalent to $I' \sqsubseteq I$.
- Function types are contravariant on arguments as usual and covariant on effect and on result. Indeed, if a function must replace another one, it must have a smaller

[4] This represent 200 000 code lines.

[5] The sens of the $\top$ or $\bot$ will become clear when subtyping will be defined. The intuition is that it is *nothing*.

concurrent effect: $T_1 \xrightarrow{I} T_2 \sqsubseteq T'_1 \xrightarrow{I'} T'_2 \iff T'_1 \sqsubseteq T_1 \wedge I \sqsubseteq I' \wedge T_2 \sqsubseteq T'_2$

- Interface subtyping is covariant on received type, contravariant on handled type and compose covariantly.

$$\{m : (T_1, T_2),\ I\} \sqsubseteq \{m : (T'_1, T'_2),\ I'\} \iff T_1 \sqsubseteq T'_1 \wedge T'_2 \sqsubseteq T_2 \wedge I \sqsubseteq I'$$

The intuition behind this rule is that the system must keep the largest type T_r of received messages and the lowest type T_u of handled messages. The correctness predicate $\diamond$ leads to $T_r \sqsubseteq T_u$ and any received content of type T is guaranteed to be understood by any receiver state T' because $T \sqsubseteq T_r \sqsubseteq T_u \sqsubseteq T'$.

Attentive readers may have remarked that the subtyping on interfaces is defined only for rows beginning by the same message label. A complete algebraic theory exists and proves that it is the only needed rule. If one label of the left side row is absent from right side row, the subtyping is clearly false and once all left side labels are treated, the system reduces to $\{\} \sqsubseteq I$ which is an axiom.

Another example

Before going into further discussion on this type system, consider a function that realizes a timer waiting for a message `cancel` or the end of a time specified at its creation to throw an alarm:

```
timer({Pid, Time, Alarm}) ->
    receive {cancel,Pid} -> true
    after Time -> Pid ! Alarm
    end.
```

A `timeout` function spawns such a timer process using the *pid* of the current process and returns the pid of the timer. The same process may cancel this timer using the returned *pid*:

```
timeout({Time, Alarm}) ->
    spawn(timer, {self(),Time,Alarm}).
cancel(Timer) ->
    Timer ! {cancel,self()}.
```

Supposing arguments of `after` (`Time`) are integers, our system infers:

$$\texttt{timer}: \quad \widehat{\alpha} \times int \times \alpha \xrightarrow{\{\texttt{cancel}:(\bot,\ @\widehat{\alpha})\}} \texttt{true} \sqcup \alpha$$

$$\texttt{timeout}: \quad int \times \alpha \xrightarrow{\widehat{\alpha}} @\{\texttt{cancel} : (\bot,\ @\widehat{\alpha})\}$$

$$\texttt{cancel}: \quad @\{\texttt{cancel} : (@\phi,\ \top)\} \xrightarrow{\phi} \texttt{cancel} \times @\phi$$

meaning that:

- The `timer` function takes three arguments: an address (receiving the third argument), an integer and a value (a message). The result is either `true` or this value and the current process receives a `cancel` message containing (an address of) a process that receives the third argument.
- `Alarm` (of type α) must be a legal message (tuple beginning by an atom).
- The process calling `timeout` receives the alarm (it appears in `timeout` effect).
- The result of this function is the name of a process understanding `cancel` messages containing an address that receives the alarm message.
- A call to `cancel` must includes an argument that receives a cancellation message containing the address of the current process and returns this cancellation message.

Those types are complex but very informative about the behavior of these functions. For example, the system can ensure that the *pid* returned by a call to `timeout` does not receive messages other than cancellation. It can also ensure that the process calling this function is able to receive the alarm message.

Functional Typing

Pattern matching cannot be treated in the usual ML way: $(\alpha_1 \to \beta_1) \sqcup (\alpha_2 \to \beta_2)$ cannot be equal to $(\alpha_1 \sqcap \alpha_2) \to (\beta_1 \sqcup \beta_2)$. In fact, the type system must include pattern matching, to do this [2] introduced the notion of conditional type $t_1?t_2$. This type means t_1 (if t_2 is different from $\bot$) or $\bot$. For example, if $e : t_e$, `case` e `of true -> 1; false -> foo` is of type $(int?(t_e \sqcap \texttt{true})) \sqcup (\texttt{foo}?(t_e \sqcap \texttt{false}))$. Our system does not use this conditional type which enjoys good algebraic properties but is not really readable and leads to the loss of the pattern matching structure. Instead, we use a conditional constraint $c_1 \Rightarrow c_2$ meaning that if c_1 is verified then the system must also ensure c_2. This constraint, generated to approximate pattern matching, allows to keep a high level of precision on the link between matched values and results. Typing previous choice lead to the following set of constraints: $C = \{t_e \sqsubseteq \texttt{true} \Rightarrow int \sqsubseteq t_r,\ t_e \sqsubseteq \texttt{false} \Rightarrow \texttt{foo} \sqsubseteq t_r,\ t_e \sqsubseteq \texttt{true} \sqcup \texttt{false}\}$ where t_r is the result type. Either the system knows the structure of t_e and C can be simplified, or it is decomposed in two sub-systems (because the matching is composed of two branches):

- One, in which, t_e is subtype of `true` and therefore $C = \{t_e \sqsubseteq \texttt{true},\ int \sqsubseteq t_r\}$
- Otherwise (due to third constraint), t_e is a subtype of `false` and $C = \{t_e \sqsubseteq \texttt{false},\ \texttt{foo} \sqsubseteq t_r\}$

As, in general, we do not know precisely the matched value, all those decomposed sub-systems must have a solution. This means that a n branch pattern matching fires the resolution of n sub-systems. However, the practice have shown that this is not a real problem. Indeed, when applying a pattern matching to a value, we often know more or less its structure and many of the sub-systems are trivial.

The typing judgments have the following shape:

$$\text{Environment} \vdash \text{Expression} : \text{Type}, \ \text{ConstraintSet}$$

As, many typing rules are classic, we limit our explainations to sends, choices, receives and calls:

- Typing $e_1!e_2$ returns the second sub-expression type and the constraint set containing all constraints produced by the typing of e_1 and e_2, plus a constraint specifying that e_1 must evaluate to a process that receives the value of e_2:

$$\frac{\mathcal{E} \vdash e_1 : t_1,\ C_1 \qquad \mathcal{E} \vdash e_2 : t_2,\ C_2}{\mathcal{E} \vdash e_1!\,e_2 : t_2,\ C_1 \cup C_2 \cup \{t_1 \sqsubseteq @\widehat{t_2}\}}$$

- Typing a choice consists in typing the tested value and all patterns and associated expressions of the filter. A reaction expression must be typed after adding to the current environment the environment resulting from typing of the corresponding pattern:

$$\frac{\mathcal{E} \vdash e : t_e,\, C_e \qquad \mathcal{E} \vdash p_i : t_i^p,\, \mathcal{E}_i \qquad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\, C_i}{\mathcal{E} \vdash \texttt{case}\ e\ \texttt{of}\ p_1 \to e_1;\ \ldots : t,\ C_e \cup \bigcup_i C_i \cup C}$$

where the resulting constraints cumulate all already calculated constraints and those due to the choice (C). C specifies that the tested value must be taken into account by one of the patterns and add all already explained conditional constraints (one for each branch):

$$C = \{t_e \sqsubseteq \bigsqcup_i t_i^p\} \cup \bigcup_i (\{t_e \sqsubseteq t_i^p \Rightarrow t_i \sqsubseteq t\})$$

This means that the result type t will be the *union* of the type of each pattern that may match the tested value.

- Typing the message handling may result in any possible branch type (hence the union) and adds all pattern types to the current self type:

$$\frac{\begin{array}{c}\mathcal{E} \vdash p_i : t_i^p,\, \mathcal{E}_i \\ \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\, C_i \qquad C_i' = \{\mathcal{E}(\texttt{self}) \sqsubseteq @\overline{t_i^p}\}\end{array}}{\mathcal{E} \vdash \texttt{receive}\ p_1 \to e_1;\ \ldots : \bigsqcup_i t_i,\ \bigcup_i (C_i \cup C_i')}$$

- Typing an application is much more complex. First, one must type the function expression and each argument expression.

$$\frac{\mathcal{E} \vdash e : t_e,\, C_e \qquad \mathcal{E} \vdash e_i : t_i,\, C_i}{\mathcal{E} \vdash e(e_1, \ldots, e_n) : t,\ C_e \cup \bigcup_i C_i \cup C}$$

where C is composed of $t_e \sqsubseteq dom(T_{\mathcal{F}})$, $\mathcal{E}(\texttt{self}) \sqsubseteq @I$, $\mathbf{Fun}(T_{\mathcal{F}}, t_e, n) \sqsubseteq (t_1 \times \ldots \times t_n) \xrightarrow{I} t$ meaning that:

  - The function must be defined.
  - Its effect I is added to the current process effect.
  - All possible functions are subtype of a function type accepting the n actual arguments t_i, having an effect I and resulting in t (it is the result of the application). To get the set of possible functions, we use a function **Fun** which applied to $(T_{\mathcal{F}}, t_e, n)$ returns the union of all function types associated to an atom (and the arity n) of t_e in $T_{\mathcal{F}}$. Like the transformation from tuple type to message type, this function is lazy and waits to know the value of t_e to perform its action.

For each possible functions of type $\alpha \xrightarrow{I'} \beta$, the last constraint ensures that all applications are legals because by substyping it leads to $\{t_1 \times \ldots \times t_n \sqsubseteq \alpha,\ I' \sqsubseteq I,\ \beta \sqsubseteq t\}$. Furthermore, all effects (resp. results) are cumulated in the global effect I (resp. result t).

The function typing environment $T_{\mathcal{F}}$ results from the typing of all functions in $\mathcal{F}$. A mapping $(s, n) \mapsto f$ in $\mathcal{F}$ adds a mapping $(s, n) \mapsto t_f$ if the typing of f by the rule below results in t_f. And, We suppose that all constraints it may produce are added to the global constraint set before resolution.

$$\frac{\mathcal{E} \vdash p_i : t_i,\, \mathcal{E}_i \qquad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i',\, C_i}{\mathcal{E} \vdash p_1 \to e_1;\ \ldots : \bigsqcup_i (t_i \to t_i'),\ \bigcup_i C_i}$$

Going back to our example, the application of `F` leads to:

$$\left\{\begin{array}{l}\texttt{state3} \sqsubseteq T,\ unit \sqsubseteq \alpha,\ \ T \sqsubseteq \{\texttt{state1}, \texttt{state2}, \texttt{state3}\}, \\ T_P \sqsubseteq @I,\ \ \mathbf{Fun}(T_{\mathcal{F}}, T, 0) \sqsubseteq unit \xrightarrow{I} t\end{array}\right\}$$

The first constraint combined with the fifth leads to:

$$unit \xrightarrow{\{\texttt{kill}\,:\,(\perp,\, unit)\}} \texttt{true} \sqsubseteq unit \xrightarrow{I} t$$

This imply that $T_P \sqsubseteq @I \sqsubseteq @\{\texttt{kill} : (\perp, unit)\}$ and $\texttt{true} \sqsubseteq t$. The first constraint simulates (in the type system) the reception of *unit* message: $(\perp, unit) \sqsubseteq (unit, \alpha)$ equivalent to $\{\perp \sqsubseteq unit,\ \alpha \sqsubseteq unit\}$. Adding this to the initial constraint set leads to a solvable constraint set (where $\alpha = unit$). This allows the system to guarantee the correctness.

6. SCALING TO ERLANG TYPING

The simplified system presented here does not correspond to the real prototype implementation. To scale to this system, we have to:

- extend the types by lists, characters, floating point numbers and all other basic types (corresponding to ERLANG basic values). This extension and the definition of built-in function is straightforward but need to add a lot of rules.
- change scoping rule policy. Our system needs to have an input and an output environment for each expression. This is also boring routine.
- add guards to the pattern matching (again routine extension). Notice that in the prototype, it is one of the constructions that contains a lot of type informations.
- take care of dynamic patterns. Indeed, in ERLANG, a variable in a pattern is a definition only if the variable is not already defined. This small modification of the semantics and more precisely of the semantics of patterns needs important changes in the type system summarized just below.

One of the biggest problem that we faced when typing ERLANG is dynamic pattern matching. Indeed, in the patterns, a variable is not always a binding occurrence, that is, if the variable is already bound, its value replaces the variable before pattern matching is realized. For example, consider:

```
g(X) -> case 1 of X -> ok; _ -> no end.
```

The term `{g(1),g(2)}` reduces to:

```
{case 1 of 1 -> ... , case 1 of 2 -> ...}
```

and then to `{ok,no}`. Usual typing of this function gives $\alpha \to t$ with the constraints:

$$\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t\}$$

Therefore, the application has type $(\texttt{ok} \sqcup \texttt{no}) \times (\texttt{ok} \sqcup \texttt{no})$ because the two applications gives $1 \sqcup 2 \sqsubseteq \alpha$ meaning that both branches may be used. The problem comes from the fact, that the usual function typing impose to all possible real argument types to be **simultaneously** compatibles with all their potential use in the body of the function. For this, when typing the body of the function, the system collects constraints of the form $\alpha \sqsubseteq t$ where α is the type of an argument. And each call to the function produces constraints of the form $t' \sqsubseteq \alpha$ which enable by transitivity to ensure that $t' \sqsubseteq t$. But, in the body of a function, if a pattern includes an argument, the system generates a constraint $t \sqsubseteq \alpha$ incomparable with $t' \sqsubseteq \alpha$. This means that we cannot guarantee that the argument respect one of the constraints required by the function.

The type obtained for `{g(1),g(2)}` is not very precise (using usual strategy) but above all, if the joker branch is not in the choice, the program cause an error that cannot be detected by the type system. To solve this problem, the system is going to type each application of a function using a fresh instance of its type. With this strategy no harmful flow (of information) may happen between two application sites as before. Indeed, the intuition behind this problem is that when a function use one of its arguments in a pattern, each application produces a new (and different) version of the body (of the function). Therefore, the constraints it imposes are not the same and the return type are different too.

The typing of a function leads to a type $\alpha \to \beta$ and a constraint set C. Its calling on an argument of type t will use type $t \to \beta'$ (where β' is fresh) and add $[t/\alpha,\ \beta'/\beta]C$ to the global constraint set. Therefore, typing:

```
g(X) -> case 1 of X -> ok end.
```

gives $\alpha \to t$ with $\{1 \sqsubseteq \alpha,\ \texttt{ok} \sqsubseteq t\}$. Therefore, the type of `{g(1),g(2)}` is $t_1 \times t_2$ with $\{1 \sqsubseteq 1,\ \texttt{ok} \sqsubseteq t_1,\ \boxed{1 \sqsubseteq 2},\ \texttt{ok} \sqsubseteq t_2\}$ where the boxed constraint is false. The error is now detected!

The drawback of this strategy is that the number of type variables and constraints grow more rapidly. To solve this problem, in practice, the system apply this strategy only to a subset of functions. More precisely, this strategy is applied to the arguments of functions using one of their arguments in a pattern. As this situation is not the most usual, the cost to pay (for this strategy) is not too expensive (in general).

7. DISCUSSION

In this paper, we have proposed a formalization of the ERLANG semantics using a two level reduction system. A first level concentrates on concurrent aspects of the language using a formalism inspired by the π-calculus, the configurations. And a second expressing the functional semantics (and its potential concurrent effects) using a more classic setting. Finally, we have introduced a type system for ERLANG insisting in the original parts of our works: message typing and the fact that the system try to stay close to the language. The versions presented in this article represent only insight of the complex system developed and the prototype of static analyzer realized.

Formal semantics of Erlang

This work though not complete can be a good beginning to reach a good formalization of the semantics of ERLANG. A complete formalization of the whole language would require a lot of work because one would have to:

- **add the node (site) notion**. For this, configurations must be extended by a set of node names and by a construction $\langle n \mid w \rangle_n$ meaning that w is executed on node n. A configuration describing a two nodes could then be $\nu n_1, n_2.(\langle n_1 \mid w_1 \rangle \parallel \langle n_2 \mid w_2 \rangle)$.

- **implement dynamic code replacement**. Each site must include the environment of defined functions and the values of those functions could change: $\langle n \mid \mathcal{E} \mid w \rangle$.

- **allow sending message between sites**. The target of the message may be local keeping the same syntax or remote on node n and the transit message could be $a@n \triangleleft m$.

- **integrate the time notion**. In ERLANG, the message handling operation has a clause `after` that allows to stop the execution of this instruction after a specified delay. One solution could be to add a notion of counter to each node.

- **add a notion of symbolic names and a dictionary**. A service can be abstracted by associating it with a name. This declared name represent a process (that can change). Each node needs to maintain dictionary: $\langle n \mid \mathcal{E}_f \mid \mathcal{E}_n \mid w \rangle$.

- **add signals**. ERLANG use signals to propagate exceptions among processes. For example, we could add a flag to the message making it possible for the receiver to distinguish a signal from a message.

Some recent work on distributed process calculi like $D\pi$ (see [21]) or the join calculus (see [14]) can also help in such a project of formalization of the semantics of ERLANG. Notice that those points are not all the problems that needed to be solved, we refer the interested reader to the chapter 10, 11 and 12 of [3]. Those three chapters does not include a formal semantics but their informal systematic description of ERLANG semantics enable to view all possibilities.

Complete Erlang Typing

To become a complete and widely usable tool our system needs some extensions.

First, the ERLANG messages does not contain label so the type of process must be retailored. The works on XMλ (a typed functional language used to manipulate XML documents) of [23] can be a good basis. Indeed, to type correctly the choices of XML, they build a typed λ-calculus including a notion of record without label. For example, $(1) + (\texttt{"test"}) + (\lambda x.\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 0)$ is typed by $\{int;\ string;\ bool \to int\}$. This adaptation does not seem to be straightforward because the type system of XMλ use equality constraints and is based upon a notion of constraint implication. Therefore, its integration with the subtyping needed for ERLANG needs studies about subtyping constraint implication and to our knowledge, none of the work made in this area have really achieved that goal yet.

In the context of telecommunication systems, exceptions are very important to reach a certain level of quality for programs. Indeed, the reliability of such applications needs a precise treatment of every possible exceptions. A type system helping the programmer in this task would be a real aid. It could estimate the set of potential exception caused by every expressions of the program and ensure that they are treated. An extension of [18] may be a good start point toward such a static analyzer.

Finally, the most difficult point with ERLANG is that the approximation made by this ideal type system should have to be compatible with *hot code swapping*. Indeed, in ERLANG, a module is used by hundreds or thousands of nodes that cannot be stopped or restarted. An evolution of such a module use dynamic code replacement and therefore, the old version and the new one have to be executed simultaneously and must cooperate safely (at least for a temporary period). Such a task is totally out of reach at the moment, but a first step to its resolution could start from [22].

8. REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. The MIT Press, Cambridge, MA, USA, 1986.

[2] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proc. of POPL*, pages 163–173, Portland, USA, Jan. 1994. ACM Press.

[3] J. Barklund and R. Virding. ERLANG *4.7.3 Reference Manual*, February 1999. downloadable from `www.erlang.org`.

[4] G. Boudol. The π-calculus in direct style. In *Proc. of POPL*, pages 228–241. ACM, Jan. 1997.

[5] R. Carlsson. An introduction to core erlang. Erlang Workshop. Principles, Logics, and Implementations of High-level Programming Languages. Florence, 2001.

[6] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. *Core Erlang 1.0.2, language specification*, Oct. 2001.

[7] J.-L. Colaço, M. Pantel, F. Dagnat, and P. Sallé. Static safety analysis for non-uniform service availability in actors. In *Proc. of FMOODS*, pages 371–386, Florence, Italy, Feb. 1999. Kluwer.

[8] J.-L. Colaço, M. Pantel, and P. Sallé. CAP: An actor dedicated process calculus. In *Proc. of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[9] J.-L. Colaço, M. Pantel, and P. Sallé. A set-constraint based analysis of actors. In *Proc. of FMOODS*, Canterbury, UK, July 1997. Chapman & Hall.

[10] F. Dagnat. A framework for typing actors and concurrent objects. Ongoing report, available from `perso-info.enst-bretagne.fr/~fdagnat`, 2002.

[11] F. Dagnat, M. Pantel, M. Colin, and P. Sallé. Typing concurrent objects and actors. *L'Objet – Méthodes formelles pour les objets*, Volume 6(1/2000):pages 83–106, May 2000.

[12] M. Dam and L. Fredlund. On the verification of open distributed systems. In *Proc. of the ACM Symposium on Applied Computing*, volume 28, pages 532–540. ACM, June 1998.

[13] M. Fahndrich. *BANE: A library for Scalable Constraint-Based Program Analysis.* PhD thesis, University of California at Berkley, 1999.

[14] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proc. of CONCUR, Pisa, Italy*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[15] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *Proceedings of ICFP '99*, 34(9):261–272, Sept. 1999.

[16] A. Lindgren. A prototype of a soft type system for erlang. Master's thesis, Computing Science Departement, Uppsala University, 1996.

[17] S. Marlow and P. Wadler. A practical subtyping system for ERLANG. In *Proc. of International Conference on Functionnal Programming*, June 1997.

[18] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[19] F. Pottier. Simplifying subtyping constraints: a theory. *Information & Computation*, 170(2):153–183, Nov. 2001.

[20] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998.

[21] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. of ICALP '98. LNCS 1443*, pages 695–706. Spinger-Verlag, July 1998.

[22] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. of POPL*, pages 236–247, London, UK, Jan. 2001.

[23] M. Shields and E. Meijer. Type-indexed rows. In *Proc. of POPL*, pages 261 – 275, London, UK, Jan. 2001.

[24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[25] X. Thirioux, M. Pantel, and M. Colin. Multi-set abstraction of non-uniform behavior concurrent objects. Work in progress, Nov. 2002.

[26] V. T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. of OTAS, Kanazawa, Japan*, volume 742 of *LNCS*, pages 460–474, New York, USA, 1993. Springer-Verlag.

[27] M. Williams and J. Armstrong. *Program Development Using Erlang - Programming Rules and Conventions.* ERICSSON, mar 1996. Doc. EPK/NP 95:035.

APPENDIX

Configurations reduction rules:

Congruence :
$$\frac{w_1 \equiv w_1' \qquad w_1' \longrightarrow w_2' \qquad w_2' \equiv w_2}{w_1 \longrightarrow w_2}$$

Parallel :
$$\frac{w_1 \longrightarrow w_2}{w \,||\, w_1 \longrightarrow w \,||\, w_2}$$

Restriction :
$$\frac{w_1 \longrightarrow w_2}{\nu a.w_1 \longrightarrow \nu a.w_2}$$

Accept :
$$\frac{\mathcal{P}(m,e)}{\langle a \,|\, \widetilde{m}\rangle \triangleright e \parallel a \triangleleft m \longrightarrow \langle a \,|\, m\ \widetilde{m}\rangle \triangleright e}$$

Reject :
$$\frac{not(\mathcal{P}(m,e))}{\langle a \,|\, \widetilde{m}\rangle \triangleright e \parallel a \triangleleft m \longrightarrow \mathbf{Err}}$$

Expression :
$$\frac{a \notin \mathcal{FN}(\alpha \triangleright e) \qquad a \vdash \alpha,\ e \xrightarrow{w}_e \alpha',\ e'}{\alpha \triangleright e \longrightarrow \nu a.(\alpha' \triangleright e' \parallel w)}$$

Evaluation context grammar:

$$\begin{array}{l}
\mathrm{C} ::= [] \mid (\mathrm{C}) \mid \{\mathrm{A}\} \mid \mathrm{C}, e \mid \mathrm{C}\ !\ e \mid e\ !\ \mathrm{C} \mid \mathrm{C}(e, \ldots, e) \mid e(\mathrm{A}) \mid \texttt{case}\ \mathrm{C}\ \texttt{of}\ f\ \texttt{end} \\
\mathrm{A} ::= [] \mid e, \mathrm{A} \mid \mathrm{A}, e
\end{array}$$

Matching semantics:

$$\begin{cases} v/[] \triangleq \mathbf{Err} \\ v/(p\ \texttt{when}\ g \rightarrow e) :: f \triangleq \begin{cases} v/f & \text{if } \mathbf{match}(p,v) = \mathbf{fail} \\ \sigma(e) & \text{if } \mathbf{match}(p,v) = \sigma \end{cases} \end{cases}$$

Functional reduction rules:

Variable Error :
$$a \vdash \alpha,\ \mathrm{C}[x] \longrightarrow_e \alpha,\ \mathbf{Err}$$

Sequence :
$$a \vdash \alpha,\ \mathrm{C}[v,\ e] \longrightarrow_e \alpha,\ \mathrm{C}[e]$$

Application Error :
$$\frac{(v,n) \notin dom(\mathcal{F})}{a \vdash \alpha,\ \mathrm{C}[v(v_1, \ldots, v_n)] \longrightarrow_e \alpha,\ \mathbf{Err}}$$

Application :
$$a \vdash \alpha,\ \mathrm{C}[v(v_1, \ldots, v_n)] \longrightarrow_e \alpha,\ \mathrm{C}[\{v_1, \ldots, v_n\}/\mathcal{F}(v,n)]$$

Case :
$$a \vdash \alpha,\ \mathrm{C}[\texttt{case}\ v\ \texttt{of}\ f\ \texttt{end}] \longrightarrow_e \alpha,\ \mathrm{C}[v/f]$$

Send Error :
$$\frac{v_1 \notin \mathbb{A}}{a \vdash \alpha,\ \mathrm{C}[v_1\ !\ v_2] \longrightarrow_e \alpha,\ \mathbf{Err}}$$

Send :
$$\frac{v_1 \in \mathbb{A}}{a \vdash \alpha,\ \mathrm{C}[v_1\ !\ v_2] \xrightarrow{v_1 \triangleleft v_2}_e \alpha,\ \mathrm{C}[v_2]}$$

Spawn Error :
$$\frac{v' \text{ is not a tuple}}{a \vdash \alpha,\ \mathrm{C}[\texttt{spawn}(v, v')] \longrightarrow_e \alpha,\ \mathbf{Err}}$$

Spawn :
$$a \vdash \alpha,\ \mathrm{C}[\texttt{spawn}(v, v_1, \ldots, v_n)] \xrightarrow{\langle a \,|\, \varnothing\rangle \triangleright v(v_1, \ldots, v_n)}_e \alpha,\ \mathrm{C}[a]$$

Self Error :
$$a \vdash \star,\ \mathrm{C}[\texttt{self}()] \longrightarrow_e \star,\ \mathbf{Err}$$

Self :
$$a \vdash \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[\texttt{self}()] \longrightarrow_e \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[a']$$

Receive Error :
$$a \vdash \star,\ \mathrm{C}[\texttt{receive}\ f\ \texttt{end}] \longrightarrow_e \star,\ \mathbf{Err}$$

Receive :
$$\frac{\mathbf{matchmailbox}(f, \widetilde{m}) = \widetilde{m}',\ e}{a \vdash \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[\texttt{receive}\ f\ \texttt{end}] \longrightarrow_e \langle a' \,|\, \widetilde{m}'\rangle,\ \mathrm{C}[e]}$$

Mailbox semantics:

$$\frac{\exists j \quad (\forall i < j \ \ m_i/f_1 = \mathbf{Err}) \qquad m_j/f_1 = e}{\mathbf{matchmailbox}(f_1 :: \_, (m_i)_{i \in J}) = (m_i)_{i \in J \setminus \{j\}},\ e}$$

$$\frac{(\forall i \in J \ \ m_i/f_1 = \mathbf{Err})}{\mathbf{matchmailbox}(f_1 :: fl, (m_i)_{i \in J}) = \mathbf{matchmailbox}(fl, (m_i)_{i \in J})}$$

Type Conversion:

$$\begin{cases}
\widehat{s} \triangleq \{s : (unit,\ \top)\} \\
\widehat{s \times T_1 \times \ldots \times T_n} \triangleq \{s : (T_1 \times \ldots \times T_n,\ \top)\} \\
\widehat{\top} \triangleq \top_I \\
\widehat{\bigsqcup_i T_i} \triangleq \bigsqcup_i \widehat{T_i} \\
\widehat{\bigsqcap_i T_i} \triangleq \bigsqcap_i \widehat{T_i} \\
\widehat{\alpha} \triangleq \widehat{\alpha} \quad \text{if } \alpha \text{ is a type variable} \\
\widehat{T} \triangleq \mathbf{Err} \quad \text{otherwise}
\end{cases}
\qquad
\begin{cases}
\overline{s} \triangleq \{s : (\bot,\ unit)\} \\
\overline{s \times T_1 \times \ldots \times T_n} \triangleq \{s : (\bot,\ T_1 \times \ldots \times T_n)\} \\
\overline{\top} \triangleq \top_I \\
\overline{\bigsqcup_i T_i} \triangleq \bigsqcup_i \overline{T_i} \\
\overline{\bigsqcap_i T_i} \triangleq \bigsqcap_i \overline{T_i} \\
\overline{\alpha} \triangleq \overline{\alpha} \quad \text{if } \alpha \text{ is a type variable} \\
\overline{T} \triangleq \mathbf{Err} \quad \text{otherwise}
\end{cases}$$

Subtyping Deduction System:

$$\bot \sqsubseteq T \qquad T \sqsubseteq \top \qquad \{\} \sqsubseteq I \qquad I \sqsubseteq \top_I \qquad \frac{T \sqsubseteq T_1 \qquad T \sqsubseteq T_2}{T \sqsubseteq T_1 \sqcap T_2} \qquad \frac{T \sqsubseteq T_1}{T \sqsubseteq T_1 \sqcup T_2} \qquad \frac{T \sqsubseteq T_2}{T \sqsubseteq T_1 \sqcup T_2} \qquad \frac{i \in \mathbb{N}}{i \sqsubseteq int} \qquad \frac{s \in \mathbb{At}}{s \sqsubseteq atom}$$

$$T_1 \times \ldots \times T_n \sqsubseteq tuple \qquad \frac{\forall i \;\; T_i \sqsubseteq T_i'}{T_1 \times \ldots \times T_n \sqsubseteq T_1' \times \ldots \times T_n'} \qquad \frac{I' \sqsubseteq I}{@I \sqsubseteq @I'} \qquad \frac{T_1' \sqsubseteq T_1 \qquad I \sqsubseteq I' \qquad T_2 \sqsubseteq T_2'}{T_1 \xrightarrow{I} T_2 \sqsubseteq T_1' \xrightarrow{I'} T_2'}$$

$$\frac{T_1 \sqsubseteq T_1' \qquad T_2' \sqsubseteq T_2 \qquad I \sqsubseteq I'}{\{m : (T_1, T_2),\ I\} \sqsubseteq \{m : (T_1', T_2'),\ I'\}}$$

Typing Deduction System:

Var
$$\frac{V \in dom(\mathcal{E})}{\mathcal{E} \vdash V : \mathcal{E}(V),\ \{\}}$$

Constant
$$\mathcal{E} \vdash c : c,\ \{\}$$

Tuple
$$\frac{\mathcal{E} \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash \{e_1, \ldots, e_n\} : t_1 \times \ldots \times t_n,\ \bigcup_i C_i}$$

Paren
$$\frac{\mathcal{E} \vdash e : t,\ C}{\mathcal{E} \vdash (e) : t,\ C}$$

Sequence
$$\frac{\mathcal{E} \vdash e_1 : t_1,\ C_1 \qquad \mathcal{E} \vdash e_2 : t_2,\ C_2}{\mathcal{E} \vdash e_1,\ e_2 : t_2,\ C_1 \cup C_2}$$

Send
$$\frac{\mathcal{E} \vdash e_1 : t_1,\ C_1 \qquad \mathcal{E} \vdash e_2 : t_2,\ C_2}{\mathcal{E} \vdash e_1 !\, e_2 : t_2,\ C_1 \cup C_2 \cup \{t_1 \sqsubseteq @\widehat{t_2}\}}$$

Case
$$\frac{\mathcal{E} \vdash e : t_e,\ C_e \qquad \mathcal{E} \vdash p_i : t_i^p,\ \mathcal{E}_i \qquad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash \texttt{case}\ e\ \texttt{of}\ p_1 \to e_1;\ \ldots : t,\ C_e \cup \bigcup_i C_i \cup \{t_e \sqsubseteq \bigsqcup_i t_i^p\} \cup \bigcup_i (\{t_e \sqsubseteq t_i^p \Rightarrow t_i \sqsubseteq t\})}$$

Application
$$\frac{\mathcal{E} \vdash e : t_e,\ C_e \qquad \mathcal{E} \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash e(e_1, \ldots, e_n) : t,\ C_e \cup \bigcup_i C_i \cup \{t_e \sqsubseteq dom(T_{\mathcal{F}}),\ \mathcal{E}(\texttt{self}) \sqsubseteq @I,\ \mathbf{Fun}(T_{\mathcal{F}}, t_e, n) \sqsubseteq (t_1 \times \ldots \times t_n) \xrightarrow{I} t\}}$$

Receive
$$\frac{\mathcal{E} \vdash p_i : t_i^p,\ \mathcal{E}_i \qquad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash \texttt{receive}\ p_1 \to e_1;\ \ldots : \bigsqcup_i t_i,\ \bigcup_i (C_i \cup \{\mathcal{E}(\texttt{self}) \sqsubseteq @\overline{t_i^p}\})}$$