Xjenza Online - Journal of The Malta Chamber of Scientists
www.xjenza.org
DOI: 10.7423/XJENZA.2014.2.05

Research Article

Monitoring Distributed Systems with Distributed PolyLarva

I. Cassar, C. Colombo, A. Francalanza
University of Malta, Department of Computer Science

Abstract. PolyLarva is a language-agnostic runtime verification tool, which converts a PolyLarvaScript into a monitor for a given system. While an implementation for PolyLarva exists, the language and its compilation have not been formalised. We therefore present a formal implementation-independent model which describes the behaviour of PolyLarvaScript, comprising of the μLarvaScript grammar and of a set of operational semantics. This allows us to prove important properties, such as determinism, and also enables us to reason about ways of re-designing the tool in a more scalable way. We also present a collection of denotational mappings for μLarvaScript converting the constructs of our grammar into constructs of a formal actor-based model, thus providing an Actor semantics for μLarvaScript. We are also able to prove certain correctness properties of the denotational translation such as that the denoted Actors behave in a way which corresponds to the behaviour described by our implementation-independent model. We finally present DistPolyLarva, a prototype implementation of the distributed PolyLarva tool, which implements the new actor-based semantics over a language that can natively handle distribution and concurrency called Erlang.

1 Introduction

Runtime Verification (RV) is a dynamic Bauer et al. (2006), Colombo (2008) verification technique which invokes monitoring procedures at runtime so as to verify that the current execution, of the system being verified, is correct with respect to a given specification. It is therefore important that RV tools should be verified for correctness themselves, thus making users more confident in trusting and relying on such tools for verification. As RV tools weave additional monitoring code into the system being verified, an inevitable runtime overhead is imposed upon the system. Moreover, monitoring demands may quickly increase especially when monitoring distributed systems, as these systems are able to scale up rapidly. Such a drastic increase in monitoring load would impose a negative effect on the monitoring efficiency, thus also affecting the performance of the monitored system. For this reason, various ways are being explored by which this overhead can be minimized Colombo et al. (2012), Francalanza and Seychell (2013). Concurrency and parallelisation provide a way of decreasing these overheads by exploiting tightly-coupled, multi-core architectures. When dealing with high monitoring demands, distributed monitoring may also be a more scalable and feasible alternative for increasing monitoring efficiency as distribution also enables the exploitation of loosely-coupled processing units.

PolyLarva Mizzi (2012), Colombo et al. (2012) is a language agnostic RV compiling tool, which when given an RV specification written in polyLS (short for polyLarvaScript), creates the additional monitoring computation for a given system. polyLS language provides an event-driven monitoring framework by which one can identify and specify a number of monitoring requests, that each monitor can handle, in terms of *Events*. For each monitor, one can also specify a set of monitoring checks and handling procedures in terms of *Conditions* and *Actions*. These three components are then associated with one another in the monitor's list of *rules*.

Example 1.1.

BR1 = ReqFunds(Usr,Sum) / !IsUsrValid(Usr) → WarnUsr();
BR2 = ReqFunds(Usr,Sum) / !EnoughFunds(Sum) → WarnUsr();
BR3 = ReqFunds(Usr,Sum) → TransferFunds(Usr,Sum);

Example 1.1 shows a sample pseudo-script defining three rules all of which are related to the same *ReqFunds* event. Whenever the monitor receives an event *e* from the system, it starts by matching it with the event pattern of the first rule in the sequence, i.e., BR1. If *e* is for example of the form ReqFunds("usr1",9000), it would match the rule's pattern ReqFunds(Usr,Sum) and as a result replace every occurrence of variables Usr by "usr1" and Sum by 9000. Subsequently, when *e* matches the event pattern of BR1, the associated condition !IsUsrValid(Usr) would change into !IsUsrValid("usr1") and evaluate to either true or false. If true, the rule's action WarnUsr() would also execute. Note that once an event matches a rule, it is consumed and it cannot match any further monitoring rules. Otherwise if *e* does not match the event pattern of BR1, or does not satisfy the associated condition, rule BR1 would be ignored and the event is matched with the pattern of BR2.

1.1 Problem Definition

There are several problems with the original PolyLarva Mizzi (2012):

1. PolyLarva was developed using a compiler-driven[1] Colombo et al. (2012) approach, hence no formal language

[1]The aim was to develop an actual compiler implementation.

Correspondence to: I. Cassar (ian.cassar.10@um.edu.mt)

semantics exist for polyLS. This is not ideal as one would require a thorough understanding of how the PolyLarva compiler is implemented, in order to understand the behaviour of the language constructs. This also makes it hard to understand how the PolyLarva compiler interprets and converts the polyLS constructs into monitoring constructs and even harder to improve it.

2. Since no formal model exists for PolyLarva , there also does not exist any type of formal proof which substantiates the validity and the correctness of the PolyLarva compiler. This makes it hard for users to trust that our RV tool would correctly verify their system, as specified in their compiled script.
3. Due to the shared-state, multi-threaded design of the synthesised monitor, PolyLarva does not provide a foundation by which the compiled monitor could be easily scaled up in order to make use of distributed architectures. A distributed design would introduce more areas that can be explored in order to exploit the advantages of distributed architectures so as to be capable of handling higher monitoring demands.

2 The High-level Model

The main focus of this model is that of providing a formal, implementation-independent description of the runtime behaviour of polyLS. In fact, this model formally describes the behaviour of the most essential constructs of PolyLarva 's polyLS. It consists of the μLarvaScript grammar, derived from the original polyLS language, and from a series of *operational semantics* which provide a formal implementation-independent description of the runtime behaviour of the constructs in our grammar.

The μLarvaScript Grammar presented in Table 3.1 is made from *abstract syntax*, meaning, that the language is treated as if it has already been *parsed* and hence assumed to be syntactically correct. It assumes denumerable sets of values $v \in Val$, variables $x \in Var$, and identifiers $i \in Id = Val \cup Var$, within its other constructs. The *state* of a monitor uses variables to store values collected from system events for further analysis. The grammar also assumes the inclusion of predicate functions, which are used in conditions so as to perform checks on the monitor's state. The entire μLarvaScript grammar is defined below.

Table 3.1 - The μLarvaScript Grammar.

$M \in \mathit{Mons} ::= \langle \mathit{State,RulesList} \rangle \mid \langle \mathit{State,RulesList} \rangle \| \mathit{Mons}$
$d \in \mathit{RulesList} ::= \mathit{Rule};\ \mathit{RulesList} \mid \varepsilon$
$r \in \mathit{Rule} ::= ((q, c) \mapsto a)$
$n \in \mathit{EventName} \supseteq \{\mathit{mthdInvoked, exThrown, mthdRet, internal}\}$
$s \in \mathit{State} : \mathit{Var}^* ::= \{ x_0 , x_1 , \ldots \}$
$e \in \mathit{Event} ::= \mathit{EventName}(v_0 \in \mathit{Val} \ldots v_k \in \mathit{Val})$
$q \in \mathit{Query} ::= \mathit{EventName}(i_0 \ldots i_k)$
$b \in \mathit{Boolean} ::= \mathit{true} \mid \mathit{false}$
$c \in \mathit{Condition} ::= \mathit{Boolean} \mid !(\mathit{Condition}) \mid$
$\quad \mathit{Condition}\ \&\&\ \mathit{Condition} \mid p(v_0 \in \mathit{Val}, \ldots ,v_k \in \mathit{Val})$
$a \in \mathit{Actions} : (\mathit{State} \to \mathit{State}) ::= \mathit{stop} \mid \mathit{fail} \mid \mathit{noOp} \mid a_1,a_2 \mid$
$\quad \mathit{update(State,Function)} \mid \mathit{load(Mons)}$

A monitoring system consists of a collection of concurrent monitors, $M_0 \| M_1$, where each individual monitor, $\langle s, d \rangle$, possesses its own current *local state* "s" and its own *rule list* "d". Monitors are able to process sequences of events "t" which are forwarded to the monitor by the system. The state of a monitor, "s", comprises a set of local variables, $\{x_0, ..., x_n\}$, while a rule list, "d" consists of a sequence of *rules*. Each individual rule, of the form $((q, c) \mapsto a)$, binds an event query "q", and a condition "c", with an action "a". Although an event query, "q", has a very similar structure to an event, "e", the latter describes an actual event which originates from the system being monitored.

Conversely, the former is used to describe a *pattern* which states that the host monitor is able to handle system events which match the pattern denoted by the query. A condition "c", can be a boolean formula or a predicate which performs checks on the monitor's current state and on the values passed as its arguments, so as to yield a boolean result. Similarly, an action "a" is a deterministic function which processes a sequence of operations which can possibly modify the monitor's current state. The monitor supports the following actions: (*i*) stop – halts the execution of the current monitor; (*ii*) fail – indicates that the monitored system has violated the property; (*iii*) nop – the monitor applies a rule but does not carry out an action; (*iv*) update(S,F) – the monitor takes the current monitor state "S" and a custom action function "F" and applies "$F(S)$", such that F is able to take state S as input and return an updated monitor state; (*v*) load(M) – a monitor is able to dynamically load another monitor M.

The following example script shows the same rules defined in Example 1.1, written in μLarvaScript syntax. As a shorthand, we refer to an action $update(state, Act(args))$ as $Act(args)$.

Example 3.1.

$\langle\{usr1, funds\},$
$((ReqFunds(Usr, Sum), \mathit{!IsUsrValid(Usr)}) \mapsto \mathit{WarnUsr()});$
$((ReqFunds(Usr, Sum), \mathit{!EnoughFunds(Sum)}) \mapsto \mathit{WarnUsr()});$
$((ReqFunds(Usr, Sum), \mathit{true}) \mapsto \mathit{TransferFunds(Usr,Sum)};\rangle$

2.1 Operational Semantics

The operational semantics for polyLS consists of a group of reduction rules. These rules, defined below, are segmented into high level monitoring rules, denoted by the high-level relation ($\longmapsto$), and into the low-level monitoring rules, denoted by the low-level relation ($\to$) relation. These rules serve to indicate how a collection of monitors would behave when they receive a system event. In fact, they describe how an event is *ignored* when no monitor in the collection is able to handle the event.

μLarvaScript High-Level Monitoring rules

$$\textsc{rHlMon1}\ \frac{t \triangleright M \to t' \triangleright M'}{t \triangleright M \longmapsto t' \triangleright M'}$$

$$\textsc{rHlMon2}\ \frac{e; t \triangleright M \not\to}{e; t \triangleright M \longmapsto t \triangleright M}$$

μLarvaScript Low-Level Monitoring rules

$$\textsc{rParMon}\ \frac{t \triangleright M_0 \to t' \triangleright M_0'}{t \triangleright M_0 \parallel M_1 \to t' \triangleright M_0' \parallel M_1}$$

$$\textsc{rMonEvtHandling}\ \frac{e, s, d \Downarrow s'}{e; t \triangleright \langle s, d\rangle \to t \triangleright \langle s', d\rangle}$$

μLarvaScript Event Consumption rules

$$\textsc{rConsAx}\ \frac{\text{matches(q,e)}= \sigma \qquad s, c\sigma \Downarrow^c \text{true}}{e, s, ((q, c) \mapsto a); d \ \Downarrow\ a\sigma(s)}$$

$$\textsc{rConsInd1}\ \frac{\text{matches(q,e)}\neq \sigma \qquad e, s, d \ \Downarrow\ s'}{e, s, ((q, c) \mapsto a); d \ \Downarrow\ s'}$$

$$\textsc{rConsInd2}\ \frac{\text{matches(q,e)}= \sigma \qquad s, c\sigma \Downarrow^c \text{false} \qquad e, s, d \ \Downarrow\ s'}{e, s, ((q, c) \mapsto a); d \ \Downarrow\ s'}$$

μLarvaScript Condition Evaluation

$$\textsc{rTru}\ \frac{}{s,\text{true }\sigma \Downarrow^c \text{true}} \qquad \textsc{rFls}\ \frac{}{s,\text{false }\sigma \Downarrow^c \text{false}}$$

$$\textsc{rPred1}\ \frac{p(v_0, \dots, v_n)(s)}{s, p(x_0, \dots, x_n)\sigma \Downarrow^c \text{true}}$$

$$\textsc{rPred2}\ \frac{\neg p(v_0, \dots, v_n)(s)}{s, p(x_0, \dots, x_n)\sigma \Downarrow^c \text{false}}$$

$$\textsc{rNot}\ \frac{s, c\sigma \Downarrow^c b}{(s, !\, c\sigma) \Downarrow^c b_1} \quad \text{where } b_1 = \neg b$$

$$\textsc{rAnd}\ \frac{s, c_1\sigma \Downarrow^c b_1 \qquad s, c_2\sigma \Downarrow^c b_2}{s, c_1\sigma \;\&\&\; c_2\sigma \ \Downarrow^c b_3}$$
where $b_3 = b_1 \wedge b_2$

The high-level monitoring rules ($\longmapsto$) state that a high-level reduction is only possible if $t \ \rhd M$ is able to reduce into $t' \ \rhd M'$ through a series of low-level reductions ($\rightarrow$). However, if a low-level reduction is unable to reduce $e; t \ \rhd M$ into some other form, then it means that event "e" will be *ignored*, thus reducing $e; t \ \rhd M$ into $t \ \rhd M$ where "t" is the tail of "$e; t$" and "M" remained unmodified by the reduction.

rParMon is a low-level inductive rule which determines whether $t \ \rhd M_0 || M_1$, consisting of a sequence of events "t" and monitor collection "$M_0||M_1$", is capable of reducing into $t' \ \rhd M_0' || M_1$, where "t' " is a modified stream of events while "$M_0'||M_1$" represents a modified monitor collection. It states that such a reduction is only allowed if *there exists* some sub-monitor collection "M_0", which when given the same event stream, "t", reduces it into event stream "t' " and "M_0'", i.e., a modified version of collection "M_0". rMonEvtHandling is an axiom which specifies that a monitor, of the form "$\langle s, d\rangle$" which is provided with a sequence of events "$e; t$", changes its state to "s'". It also specifies that this reduction is allowed if the event "e", together with the current monitor's state "s" and rule list "d", are able to evaluate into the next state "s'" by using the *Event Consumption rules*.

The Event Consumption rules ($\Downarrow$) describe how an individual monitor, consisting of state "s" and rule list "d", reacts and behaves in order to handle the received event "e". In fact they indicate that a *successive* state "s'" is derived once the event has been handled by the monitor and removed from the event stream. Hence, the above rules, describe the operational behaviour by which a μLarvaScript monitor consumes a system event. Particularly, these rules define that a modified state "s'" is only produced when the received system event "e" *matches* a query "q" of one of the monitor's rules, which causes condition "c" to evaluate to true, thus invoking an action "a" which modifies state "s" into some "s'". Note that σ is produced when a query q matches [2] an event e so to provide a mapping between the variables in q and the system values received in e. This mapping is then used by conditions and actions which require information about the system. Furthermore, To evaluate a condition "c", the event consumption rules use the *Condition Evaluation* rules ($\Downarrow^c$) to determine whether the event satisfies or violates the associated condition.

[2](Cassar, 2013) provides the formal definition for $matches(q, e) = \sigma$.

2.2 The Single Receiver Property

One of the most prominent properties observed in PolyLarva was that no matter how many monitors are specified, only a maximum of *one* monitor ends up receiving and handling an event. For this reason we assume that a *sound monitoring specification* is one which coincides with the Single Receiver Property defined by Definition 3.1. This property is quite essential, especially in a distributed context, so to ensure that two or more monitors are never allowed to handle the same event simultaneously, meaning that *at most only one* monitor is allowed to execute an action whenever a specific event occurs. We therefore base our arguments and evaluation proofs upon this important property, meaning that any guarantees offered by our models, only apply for sound specifications.

Definition 3.1. The Single Receiver Property.

$$t \rhd M_0||M_1 \rightarrow t' \rhd M' \quad \textit{implies}$$
$$t \rhd M_0 \rightarrow t' \rhd M_0' \ \textit{and} \ t \ \rhd M_1 \not\rightarrow$$

3 The Distributed-State Model and its Translation

This model aims to provide a formal description of the behaviour of the μLarvaScript constructs in a way which is closely related to an actual, distributed-state implementation. In fact, this distributed-state model consists in a formal translation from μLarvaScript constructs to constructs of a formal Actor model for Erlang (presented in Sections 3.2 and 3.3) adapted from Francalanza and Seychell (2013). In this way, the meaning of the μLarvaScript constructs is given in terms of a highly scalable Haller and Sommers (2012), distributed state model, which produces a monitoring system capable of handling larger monitoring demands with the same or better performance. This claim is supported by Gustafson's Law Gustafson (1988).

3.1 Concurrency, the Actor Model & Erlang

The Actor Model Gul A. et al. (2001) is a highly scalable paradigm Haller and Sommers (2012) which offers a level of abstraction by which both data and procedures can be encapsulated into a single construct.

Actors differ from objects since actors are also concurrent units of execution, each of which executes independently and asynchronously. This fusion of data abstraction and concurrency relieves the developer from having to recur to the explicit concept of a thread in order to make use of concurrency. Moreover, since Actors communicate through Message Passing Gul A. et al. (2001), the developer does not need to develop explicit synchronization mechanisms to prohibit dangerous concurrent access to the data, shared amongst the communicating threads.

Additionally, message passing between these actors is performed asynchronously Gul A. et al. (2001), which means, that an Actor is able to send a message without having to wait for the receiver's response. Conversely, the receiver does not need to be listening for incoming messages in order to receive them since the messages are deposited in the Actor's mailbox.

In order for an actor to retrieve the received data, it must issue a receive command to recover a message from its mailbox. An important factor is that message passing in the Actor model normally assumes fairness, that is, any message sent by an actor to another existing actor, is *guaranteed* to eventually be deposited inside the target actor's mailbox. In addition to this merger between data, functions and concurrency, an actor is also assigned a unique and persistent identifier, which is essential to identify the target destination actor of the message being sent. A case in point is Erlang Vermeersch (2009), Armstrong (2007), a programming language which natively implements this model.

Although forms of concurrency are employed in the monitors synthesised by PolyLarva , this is done through multi-threading and shared state communication Mizzi (2012) using explicit locking mechanisms. As these concurrent monitors do not use a distributed state[3], they can only be executed concurrently on the same machine. This implies that unlike a distributed multi-processing design, a multi-threaded monitor side cannot exploit the full processing capabilities of loosely coupled distributed architectures, making it less scalable A and P (2010).

3.2 Actor Calculus for Erlang

The following calculus, adapted from Francalanza and Seychell (2013), denotes a formalized abstract syntax for modeling the behaviour of Erlang programs. The calculus was further restricted so as to only describe the core Erlang constructs which are most relevant to our intents and purposes.

Calculus for Actor Systems:

$$\begin{array}{rcl}
A, B, C \in \textsc{Actr} & ::= & i[e \triangleleft q] \mid A \parallel B \mid (i)A \\
q, r \in \textsc{MBox} & ::= & \epsilon \mid v : q \\
e, d \in \textsc{Exp} & ::= & v \mid \mathsf{self} \mid e!d \mid \mathsf{rcv}\, g\, \mathsf{end} \mid e(d) \mid \mathsf{spw}\, e \mid \\
 & & x = e, d \mid \mathsf{case}\, e\, \mathsf{of}\, g\, \mathsf{end} \mid \ldots \\
v, u \in \textsc{Val} & ::= & x \mid i \mid a \mid \mu y.\lambda x.e \mid \{v, \ldots, v\} \mid l \mid \mathsf{exit} \mid \ldots \\
l, k \in \textsc{Lst} & ::= & \mathsf{nil} \mid v : l \\
p, o \in \textsc{Pat} & ::= & x \mid i \mid a \mid \{p, \ldots, p\} \mid \mathsf{nil} \mid p : x \mid \ldots \\
g, f \in \textsc{PLst} & ::= & \epsilon \mid p \rightarrow e; g \mid p\ \mathsf{when}\ e \rightarrow e; g
\end{array}$$

Evaluation Contexts

$$C \ ::= \ [-] \mid C!e \mid v!C \mid C(e) \mid v(C) \mid x = C, e \mid \ldots$$

This calculus uses denumerable sets of variables $x, y, z \in$Var, atoms $a, b \in$Atom, and process identifiers $i, j, k \in$Pid amongst other constructs, so as to describe the execution of an Erlang program in terms of a "*system of actors*" Francalanza and Seychell (2013). A system of actors is composed of a collection of actors executing in parallel, $A \| B$, where each individual actor, $i[e \triangleleft q]$, is uniquely identified by its process identifier i.

Moreover, "e" represents an *expression* which the actor will execute concurrently, with respect to its local mailbox "q" Francalanza and Seychell (2013). An actor's mailbox, is denoted as a list of values[4] "$v : q$", where "v" represents the head of the queue while "q" denotes its tail. Additionally, actor expressions Francalanza and Seychell (2013) usually consist of a sequence of expressions "$x = e, d$", which is expected to reduce down to a value. Moreover, expressions may consist of: (*i*) *sending* messages to other actors through "$e\ !\, d$" (where expression e should reduce to a Pid; (*ii*) referencing to the actor's own process identifier by using self; (*iii*) applying functions to other expressions with "$e(d)$"; (*iv*) branching using the case statement; and (*v*) pattern matching when reading a value from the mailbox through the recv g end construct, where "$g \in$PLst" represents a *guarded / protected list*. Additionally, expressions Francalanza and Seychell (2013) may also define evaluation contexts expressed as "C". An expression defined within a context will be the first to execute entirely. Moreover, values may consist of variables, *recursive functions*[5] $\mu y.\lambda.x.e$, tuples $\{v_1, ..., v_n\}$, lists and other constructs.

3.3 Erlang Reduction Semantics for Actor Systems

The operational semantics in figures 1, 2 and 3 Francalanza and Seychell (2013), provide a formal description of the behaviour of the actor calculus. Moreover, the semantics assume that the actor systems are "*well-formed*" Francalanza and Seychell (2013),ie, every actor is identified by a *unique* process identifier.

$$\textsc{Com}\ \frac{}{j[C[i!v] \triangleleft q] \parallel i[e \triangleleft q] \ \rightarrow\ j[C[v] \triangleleft q] \parallel i[e \triangleleft q{:}v]}$$

$$\textsc{Rd}1\ \frac{\mathrm{mtch}(g, v) = e}{i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft (v : q)] \ \rightarrow\ i[C[e] \triangleleft q]}$$

$$\textsc{Rd}2\ \frac{\mathrm{mtch}(g, v) = \bot \qquad i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft q]^m \ \rightarrow\ i[C[e] \triangleleft r]^m}{i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft (v : q)]^m \ \rightarrow\ i[C[e] \triangleleft (v : r)]^m}$$

Figure 1: Reduction Semantics for Actor Systems - Part 1.

The Com rule, in Figure 1 describes a message passing mechanism by which an actor "$j[C[i!v] \triangleleft q]$" can send a message containing a value "v" and append it at the end of the mailbox of another actor. The recipient actor will only retrieve and be notified about the message, residing in its mailbox, when it issues a recv command. In fact, rules Rd1 and Rd2 can then be used retrieve a message from the actor's mailbox. Rd1 states that a value is retrieved from the mailbox if it matches at least one pattern of some protected list $g \in$PLst, associated with the recv command, thus returning the expression associated with the first matching guarded rule, "$p \rightarrow e$" or "p when $\rightarrow e$". Moreover, rule Rd2 is an inductive rule which allows for an actor to perform a selective receive, meaning that an actor is not restricted to only retrieve the topmost message in the queue, but is allowed to keep on searching in its mailbox, or if necessary keep on waiting for new messages, until it finds a message which matches at least one pattern in the guarded list, associated with the receive function.

The Cs1 rule, in Figure 2, states that a value "v" will only be

[3] "Distributed state" means that each monitor has its own local state and communicate through message passing.

[4] The colon ":" in $v : q$, represents the list constructor operator, ie, value v is added to list q.

[5] The "y" in $\mu y.\lambda.x.e$ denotes a self-referencing variable which is required to perform recursive calls for the function "$\lambda.x.e$".

$$\textsc{Cs1}\ \frac{\mathrm{mtch}(g, v) = e}{i[C[\mathsf{case}\ v\ \mathsf{of}\ g\ \mathsf{end}]]^m \ \rightarrow\ i[C[e]]^m}$$

$$\textsc{Slf}\ \frac{}{i[C[\mathsf{self}]] \ \rightarrow\ i[C[i]]}$$

$$\textsc{App}\ \frac{}{i[C[\mu y.\lambda x.e\,(v)]] \ \rightarrow\ i[C[e\{^{\mu y.\lambda x.e}/_{y}\}\{^{v}/_{x}\}]]}$$

$$\textsc{Spw}\ \frac{}{i[C[\mathsf{spw}\,e] \triangleleft q] \ \rightarrow\ (j)(i[C[j] \triangleleft q] \parallel j[e \triangleleft \epsilon])}$$

Figure 2: Reduction Semantics for Actor Systems - Part 2.

accepted if it matches a pattern in the associated guarded list "g". For example, consider the following code:

$$\mathsf{case}\ Bin\ \mathsf{of}\ 1 \rightarrow ok;\ 0 \rightarrow ok;\ \_ \rightarrow nok\ \mathsf{end}.$$

This code states that if variable "*Bin*" reduces to 1 or to 0, during execution, then it is accepted and the "*ok*" atom is returned. Otherwise, if it reduces to some other form, "*nok*" is returned, since in Erlang, the "_" pattern refers to a *catch-all* pattern which matches anything. Moreover, the Spw rule is used to describe how new concurrent actor instances can be dynamically created, while the Slf rule dictates that the self statement reduces into the calling Actor's Pid. Moreover,App rule states that when some value "v" is passed as an argument of a recursive function "$\mu y.\lambda.x.e$", then all occurrences of the self-referencing variable "y" in expression "e", will be replaced by the entire recursive function. Moreover, all occurrences of argument "x", in function "$\lambda.x.e$", will be replaced by the passed value "v".

Moreover, the Ass rule, in Figure 3 below, describes that in an expression sequence "$x = e, d$", when the first expression e is reduced into a value "v", the value obtained can be used by the second expression d. It also states that the obtained value "v" will bind to variable "x", meaning that this variable will store the result obtained after reducing the *entire* expression sequence. The remaining rules are quite self explanatory.

$$\textsc{Ext}\ \frac{}{i[C[x = \mathsf{exit}, e]] \ \rightarrow\ i[C[\mathsf{exit}]]}$$

$$\textsc{Ass}\ \frac{v \neq \mathsf{exit}}{i[C[x = v, e]] \ \rightarrow\ i[C[e\{^{v}/_{x}\}]]}$$

$$\textsc{Par}\ \frac{A \ \rightarrow\ A'}{A \parallel B \ \rightarrow\ A' \parallel B}$$

Figure 3: Reduction Semantics for Actor Systems - Part 3.

3.4 Alternative Semantics for μLarvaScript

The denotations in Figure 4.1 convert μLarvaScript constructs into constructs of the formal Actor model for Erlang Francalanza and Seychell (2013), thus giving Actor semantics to μLarvaScript. Also one must distinguish between the constructs which are declared *within* the denotations and those declared without any denotation. The constructs declared in a denotation are μLarvaScript constructs, for example, *abc* in $\llbracket abc \rrbracket^m$ refer to a μLarvaScript construct, while if *abc* is not declared in a denotation, then it is a construct of the Erlang model Francalanza and Seychell (2013).

$\llbracket t \ \triangleright M \rrbracket^m$ presents the root denotational function which takes an event stream t and a μLarvaScript monitor specification "M". It then invokes another denotational function $\llbracket t \rrbracket^m_{es}$, which creates a coordinating Actor that executes in parallel with the monitoring actors returned by fst($\llbracket M \rrbracket^m_{par}$). Moreover, in order for the denotation $\llbracket t \rrbracket^m_{es}$ to keep on reducing, it requires a list of process identifiers[6] (Pids) returned by snd($\llbracket M \rrbracket^m_{par}$).

The translation $\llbracket t \rrbracket^m_{es}$ converts an event stream into a *coordinating actor*, when given a list of Pids. This special Actor is required to interface with the monitored system and to make sure that the synthesized monitor is behaving in accordance with the Single Receiver Property. In fact, $\llbracket t \rrbracket^m_{es}$ creates an actor with $\llbracket t \rrbracket^m_{mb}$ as its mailbox, meaning that the system events will be delivered to the coordinator's mailbox. The coordinator consists of a recursive function which takes a list of Pids and listens for messages in its mailbox via a recv command. Whenever the coordinator receives the message $\{new, Pid\}$, it signifies that one of the concurrent monitors has issued a $\llbracket load(M) \rrbracket^m_a$ action, so as to dynamically create a new concurrent monitor. Hence, the coordinator adds the Pid of the new monitor to its Pid-list and issues a recursive call, to restart listening for other messages.

Fig 4.1 The formal translation.

$$\llbracket t \ \triangleright M \rrbracket^m \stackrel{\text{def}}{=} \llbracket t \rrbracket^m_{es}(\mathsf{snd}(\ \llbracket M \rrbracket^m_{par}\ )) \parallel \mathsf{fst}(\ \llbracket M \rrbracket^m_{par}\ )$$

$$\begin{array}{ll}
\llbracket t \rrbracket^m_{es}(\mathrm{PidList}) \stackrel{\text{def}}{=} & coord\ [(\mu\ y_{rec}\ \cdot\ \lambda\ X_{lst} \cdot (\\
& \quad \mathrm{recv}\ \{\mathrm{evt,E}\}: \rightarrow\\
& \qquad \mathrm{bcast}(\{\ \mathrm{E,self()}\ \},\mathrm{X}_{lst}),\\
& \qquad \mathrm{case\ await(len(X}_{lst})\text{-}1)\ \ \mathsf{of}\\
& \qquad\quad 0 \rightarrow y_{rec}(\mathrm{X}_{lst});\\
& \qquad\quad 1 \rightarrow y_{rec}(\mathrm{X}_{lst});\\
& \qquad\quad _ \rightarrow \mathrm{error}\\
& \qquad \mathrm{end}\\
& \quad \{\mathrm{new,Pid}\} \rightarrow\\
& \qquad y_{rec}(\mathrm{X}_{lst}\mathrm{:Pid});\\
& \quad \mathrm{end.)(PidList)} \quad \triangleleft \quad \llbracket t \rrbracket^m_{mb}\ ]
\end{array}$$

$$\begin{aligned}
\llbracket M_0 \| M_1 \rrbracket^m_{par} &\stackrel{\text{def}}{=} \{\mathsf{fst}(\ \llbracket M_0 \rrbracket^m_{par}\ ) \parallel \mathsf{fst}(\ \llbracket M_1 \rrbracket^m_{par}\ )\ ,\\
&\qquad \mathsf{snd}(\ \llbracket M_0 \rrbracket^m_{par}\ ) : \mathsf{snd}(\ \llbracket M_1 \rrbracket^m_{par}\ )\}\\
\llbracket \langle s, d \rangle \rrbracket^m_{par} &\stackrel{\text{def}}{=} \{i[(\mu y_{rec} \cdot \lambda X_{state} \cdot X_{new} = \mathsf{recv}(\llbracket d \rrbracket^m_d\\
&\qquad (X_{state}))\mathsf{end}, y_{rec}(X_{new}).)(\llbracket s \rrbracket^m_s))) \triangleleft \varepsilon]\ ,\ i\}
\end{aligned}$$

$$\llbracket \varepsilon \rrbracket^m_d \stackrel{\text{def}}{=} \lambda X_{state}\ \cdot\ \{\ \mathsf{Coord}\ ,\ \_\_\ \} \rightarrow\ \mathsf{Coord}\ !\ \mathsf{nok}, (X_{state});$$

$$\llbracket r_1\ ;\ d_1 \rrbracket^m_d \stackrel{\text{def}}{=} \lambda X_{state} \cdot \llbracket r_1 \rrbracket^m_r(X_{state})\ ;\ \llbracket d_1 \rrbracket^m_d(X_{state})$$

$$\begin{aligned}
\llbracket ((q, c) \mapsto a) \rrbracket^m_r &\stackrel{\text{def}}{=} \lambda X_{state}\ \cdot\ \{\ \mathsf{Coord}, \llbracket q \rrbracket^m_q\ \}\ \mathsf{when}\\
&\qquad (\llbracket c \rrbracket^m_c(X_{state})) \mapsto (\ \mathsf{Coord}\ !\ \mathsf{ok}, \llbracket a \rrbracket^m_a(X_{state}))
\end{aligned}$$

$$\llbracket \{x_0, x_1, \ldots, x_k\} \rrbracket^m_s \stackrel{\text{def}}{=} \llbracket x_0 \rrbracket^m_i : \llbracket x_1 \rrbracket^m_i : \ldots : \llbracket x_k \rrbracket^m_i$$

$$\llbracket \emptyset \rrbracket^m_s \stackrel{\text{def}}{=} \varepsilon$$

$$\llbracket n(v_0, \ldots, v_k) \rrbracket^m_e \stackrel{\text{def}}{=} \{'n'\ ,\ \{\llbracket v_0 \rrbracket^m_i : \llbracket v_1 \rrbracket^m_i : \ldots : \llbracket v_k \rrbracket^m_i\}\}$$

$$\llbracket n(i_0, \ldots, i_k) \rrbracket^m_q \stackrel{\text{def}}{=} \{'n',\ \{\llbracket i_0 \rrbracket^m_i : \llbracket i_1 \rrbracket^m_i : \ldots : \llbracket i_k \rrbracket^m_i\}\}$$

[6] A Pid uniquely identifies an Actor.

$$[\![true]\!]_c^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot \text{true}$$

$$[\![!\,(C)]\!]_c^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot \text{not } [\![C]\!]_c^m$$

$$[\![C_1 \&\& C_2]\!]_c^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot [\![C_1]\!]_c^m \text{ and } [\![C_2]\!]_c^m$$

$$[\![p(v_0,\ldots,v_k)]\!]_c^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot \lambda v_0,\ldots,v_k \cdot P(\{v_0,\ldots,v_k\}, X_{state})$$

$$[\![stop]\!]_a^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot \text{exit.}$$

$$[\![fail]\!]_a^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot \text{Coord ! error.}$$

$$[\![noOp]\!]_a^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot X_{state}$$

$$[\![update(S,F)]\!]_a^m \stackrel{\text{def}}{=} \lambda F \cdot \lambda S \cdot F(S)$$

$$[\![load(M)]\!]_a^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot (\text{Coord ! }\{new, \text{spw(fst(} [\![M]\!]_{par}^m \text{)) } \}\text{)}, X_{state}$$

$$[\![a_0, a_1]\!]_a^m \stackrel{\text{def}}{=} \lambda X_{state} \cdot [\![a_1]\!]_a^m([\![a_0]\!]_a^m(X_{state}))$$

Conversely, when the coordinator reads a system event message, $\{evt, E\}$, it broadcasts the message[7] $e_{msg} \equiv \{\text{self}, E\}$ to all monitors executing concurrently, by using the "*bcast*" function. The coordinator then awaits feedback from the monitors by calling "*await*(*count*)", where "*count*" is initially set to be the length of the coordinator's Pid-list. Moreover, the "*await*" function makes use of a selective receive so as to only retrieve feedback messages, of the form "*ok*" or "*nok*", from all the monitors in its Pid-list. This makes sure that only a maximum of *one* monitor has indeed handled the broadcasted event. In fact it issues an error if more than one monitor handles the event, thus signifying that the Single Receiver Property has been violated by the translated monitoring specification.

$[\![-]\!]_{par}^m$ is a function that converts a μLarvaScript monitor into a meta-level tuple containing a list of monitoring actors together with another list with their Pids. The meta-functions fst and snd are then invoked at compile-time so as to extract the two separate lists from the denoted meta-tuple. Each actor denoted by $[\![\langle s, d\rangle]\!]_{par}^m$ is *always* associated with a unique Pid, "*i*", and is initialized with an empty mailbox "ε" so as to wait for event messages of the form {CoordPid, e}, by issuing a "recv" command so as to listen for messages from the coordinator. This command is followed by $[\![d]\!]_d^m$ which converts a μLarvaScript rule list into an Erlang list of guarded rules. An empty μLarvaScript rule list, is converted by $[\![\varepsilon]\!]_d^m$ into a guarded rule which matches *any* broadcasted event message. This is required since when a message matches its pattern, the monitor sends a rejection feedback to the coordinator by using "*Coord*! *nok*" and leaves the monitor's current state unmodified.

Each μLarvaScript rule, in a non-empty rule list, is translated through $[\![((q,c) \mapsto a)]\!]_r^m$ into an Erlang guarded command. Whenever the guarded rule's tuple query, of the form {Coord,$[\![q]\!]_q^m$}, pattern matches the structure of the received event such that condition $[\![c]\!]_c^m$ returns *true*, the rule sends an "*ok*" feedback message to the coordinator, signifying that the event was handled. It then executes the function denoted by $[\![a]\!]_a^m$ on the monitor's current state, thus generating the next state.

The denotation $[\![-]\!]_s^m$, for the monitor's state, dictates that the monitor's state variables are converted into a list of Erlang variables. The translation $[\![-]\!]_e^m$, states that a μLarvaScript event is translated into an Erlang tuple containing the event name and a *tuple of values* created by the system, while the query denotation, $[\![-]\!]_q^m$, returns an Erlang tuple containing the event name and a *tuple of identifiers*, where each identifier can be either a value or a variable. The condition denotation $[\![-]\!]_c^m$, converts μLarvaScript conditions into Erlang functions which return a boolean value after performing a check on the monitor state passed as its argument. The action denotation $[\![-]\!]_a^m$, translates μLarvaScript actions into Erlang functions which take the monitor's current state and return an updated state accordingly.

Example 6.1. This example outlines how a monitor containing only the first rule used in Example 3.1, can be formally translated into Erlang code by applying the denotational functions provided.

$$[\![\langle\{usr1, funds\}, ((ReqFunds(Usr, Sum), !IsUsrValid(Usr)) \mapsto WarnUsr()); \rangle]\!]^m$$

$$\stackrel{def}{=} \textbf{\{ By applying the root denotation } [\![-]\!]^m \textbf{\}}$$

$$[\![t]\!]_{es}^m(\text{snd}(\ [\![\langle\{usr1, funds\}, ((ReqFunds(Usr, Sum), !IsUsrValid(Usr)) \mapsto WarnUsr()); \rangle]\!]_{par}^m\ ))\ \ \|\ \ \text{fst}(\ [\![\langle\{usr1, funds\}, ((ReqFunds(Usr, Sum), !IsUsrValid(Usr)) \mapsto WarnUsr()); \rangle]\!]_{par}^m\ )$$

$$\stackrel{def}{=} \textbf{\{ Applying } [\![-]\!]_{par}^m \textbf{, and extracting pidList "[i]" with the snd meta function and the actor expression with fst. \}}$$

$$[\![t]\!]_{es}^m([i])\ \|\ i[(\mu y_{rec} \cdot \lambda X_{state} \cdot X_{new} = \text{recv}(\ [\![((ReqFunds(Usr, Sum), !IsUsrValid(Usr)) \mapsto WarnUsr())]\!]_d^m (X_{state}))\text{end}, y_{rec}(X_{new}).)([\![\{usr1, funds\}]\!]_s^m))) \triangleleft \varepsilon] \quad \bullet\bullet\bullet$$

$$\stackrel{def}{=} \textbf{\{ After applying the necessary denotations \}}$$

$$[\![t]\!]_{es}^m([i])\ \|\ i[(\mu y_{rec} \cdot \lambda X_{state} \cdot X_{new} = \text{recv}(\lambda X_{state} \cdot \{\text{Coord}, \{\text{'ReqFunds'}, \text{Usr}, \text{Sum}\}\}\ \text{when}\ (!IsUsrValid(Usr))(X_{state}) \mapsto (Coord!\,ok, (WarnUsr()(X_{state})));\ \{\ \text{Coord}\ , \_\ \} \rightarrow\ \text{Coord}\ !\ \ \text{nok}, (X_{state}))\text{end}) \triangleleft \varepsilon]$$

$$\stackrel{def}{=} \textbf{\{ Applying} [\![t]\!]_{es}^m \textbf{to create the coordinator\}}$$

$$\text{coord}[(\mu y_{rec} \cdot \lambda X_{lst} \cdot (\text{recv}\ \{evt, E\} \rightarrow bcast(\{E, self()\}, X_{lst}), \text{case await}(len(X_{lst}) - 1)\ \text{of}\ 0 \rightarrow y_{rec}(X_{lst});\ 1 \rightarrow y_{rec}(X_{lst});\ \_ \rightarrow \text{error end};\ \{new, Pid\} \rightarrow y_{rec}(X_{lst} : Pid)\text{end.})([i])\ \triangleleft\ [\![t]\!]_{mb}^m]$$

$$\|\quad i[(\mu y_{rec} \cdot \lambda X_{state} \cdot X_{new} = \text{recv}(\lambda X_{state} \cdot \{\text{Coord}, \{\text{'ReqFunds'}, \text{Usr}, \text{Sum}\}\}\ \text{when}\ (!IsUsrValid(Usr))(X_{state}) \mapsto (Coord!\,ok, (WarnUsr()(X_{state})));\ \{\ \text{Coord}\ , \_\ \} \rightarrow\ \text{Coord}\ !\ \ \text{nok}, (X_{state}))\text{end}) \triangleleft \varepsilon]$$

4 The DistPolyLarva Prototype

DistPolyLarva ((Cassar, 2013)) is prototype implementation based on our new actor-based design. This prototype seeks to re-implement PolyLarva 's *monitor compiler* in a way which conforms to the denotational translations provided in our distributed-state model. This ensures that any guarantees offered by the formal models would also apply for our prototype compiler.

Also, DistPolyLarva parses a variant of polyLS, called Pseudo-polyLS, into a parse tree which, resembles the μLarvaScript

[7] Where self refers to the coordinator's Pid and E is the actual system event received.

abstract syntax, together with additional parsed constructs. Although our prototype compiler is able to recognize all polyLS keywords and synthesise additional monitoring features, which are not formalized in our models, it only guarantees correct behaviour for specifications which only use constructs from the formalized subset which forms μLarvaScript. The parsed constructs are then converted into Erlang actor expressions in a similar way as in our formal translation. Furthermore, this prototype was developed with the aim to demonstrate that our translation is implementable.

4.1 The Compilation Phases

DistPolyLarva passes a given Pseudo-polyLS specification from four subsequent stages so as to synthesise the required monitoring Erlang code.

Lexical and Parsing Phases: The Lexical phase uses a *regular grammar* which defines a number of patterns that a character sequence, in the given Pseudo-polyLS script, must match in order to be translated into an abstract token. The generated token sequence is passed to the Parsing phase which checks that the structure of the script being compiled, is correct with respect to the production rules defined by the *context free grammar* of our language defined in Table 3.1. If the entire token sequence obeys the rules of the grammar, it is converted into an unambiguous *parse tree* which conforms to the abstract syntax of μLarvaScript. DistPolyLarva's lexer was implemented using a lexer generator called LEEX while its parser was implemented using a parser generator called YECC Ericsson AB (2013).

Semantic Analysis and Code Generation Phase: This phase is essentially an Erlang implementation of our formal denotations in Figure 4.1. It starts by invoking the initial denotational function which inspects the initial node of the parse tree and invokes other denotational functions which inspect the semantics of its child nodes, from left to right. The compiler also checks that any event, condition and action referred by the rules of a specific monitor, is actually declared within the same monitor, so as to preserve scoping. The generated Erlang source modules (.erl) are then written in a directory specified by the user and are compiled into executable Beam files via the Erlang compiler.

5 Evaluation

The high level and distributed-state models were evaluated by proving certain theorems about the runtime behaviour they describe. The guarantees obtained from proving these theorems are also inherited by DistPolyLarva, as this was developed with a close relation to the formal denotational translation. Moreover, the prototype was further evaluated through a series of tests.

5.1 Evaluating the High-level Model

In order to evaluate the behaviour described by this model we proved a theorem which guarantees that any monitoring system, specified in μLarvaScript, will operate deterministically. This property is important since it ensures that whenever any collection of μLarvaScript monitors is in a particular collective state[8], and it receives a specific system event, it will *always* handle the event in the *same* manner, thus transitioning to the same successive collective state. This means that no matter how many times the monitoring system is executed, depending on the current state, it will always handle a specific event in the *same* way, and so transition to *same* consecutive state. Hence, this guarantees that a monitoring system will operate consistently.

[8] By "collective state" we refer to the local states of all monitors in the specified monitor collection.

Theorem 6.1. μLarvaScript Determinism.
$t \rhd M \longmapsto t' \rhd M' \ \wedge\ t \rhd M \longmapsto t'' \rhd M''\ \textit{implies}\ \ t' = t'' \ \wedge\ M' \equiv M''$

Specifically, Theorem 6.1 states Cassar (2013) that if M reduces to both $t \rhd M'$ and $t \rhd M''$, by a using *high-level* reduction ($\longmapsto$), then it implies that $t \rhd M'$ and $t \rhd M''$ are *equal* to each other. The proof of this theorem was divided into separate lemmas, each of which were proved accordingly by using various inductive techniques.

5.2 Evaluating the Formal Translation

The evaluation of our denotational semantics consisted in proving Theorem 6.2, which shows that our formal translation is in some sense correct. We showed that the behaviour of *any* actor-based monitoring system, derived using our denotational conversion, *corresponds* to the behaviour described by the high-level model. These proofs not only help to increase the user's confidence but also state that any property proved on our high-level model, such as determinism in Theorem 6.1, would also transitively apply to our synthesised monitoring system. In our proofs we assume that all μLarvaScript specifications observe the Single Receiver Property. This implies that the denotational translation is only guaranteed to provide a correctly-behaving actor implementation when the specification script being translated observes the Single Receiver Property.

Theorem 6.2. Behaviour Correspondence.
Let M be a sound μLarvaScript specification and assume
$t \rhd M$ *behaves as* $[\![t \rhd M]\!]^m \cdot$

if $t \rhd M \longmapsto {}^*t' \rhd M'$ *and* $[\![t \rhd M]\!]^m \rightarrow^* [\![t' \rhd M']\!]^m$
then $t' \rhd M'$ behaves as $[\![t \rhd M']\!]^m$

Theorem 6.2 was further subdivided and proven using the following correspondence lemmas.

Lemma 6.1. Single-Step Correspondence.
$t \rhd M \longmapsto t' \rhd M'\ \textit{implies}\ [\![t \rhd M]\!]^m \rightarrow^* [\![t' \rhd M']\!]^m$

Lemma 6.2. Multi-Step Correspondence.
$t \rhd M \longmapsto {}^*t' \rhd M'\ \textit{implies}\ [\![t \rhd M]\!]^m \rightarrow^* [\![t' \rhd M']\!]^m$

Lemma 6.1 guarantees that for *one* high-level reduction, i.e., $t \rhd M \longmapsto t' \rhd M'$, there exists a corresponding translation, $[\![t \rhd M]\!]^m$, which reduces in 0 or more Erlang reduction steps into $[\![t' \rhd M']\!]^m$. The proof for Lemma 6.2 relies on Lemma 6.1 so as to guarantee that for *0 or more* high level reductions, we can find a denotational translation which reduces $[\![t \rhd M]\!]^m$ in 0 or more Erlang steps into $[\![t' \rhd M']\!]^m$.

6 Future Work

As part of our future work we propose to extend our μLarvaScript grammar so as to formalize other polyLS constructs such as timers. This extension requires modifications to our formal models, as well as, additional formal results. The new results would guarantee that the extended high-level model still operates deterministically and that its behaviour still corresponds to the behaviour of an extended version of our distributed-state model. The additional features in our DistPolyLarva compiler could then be properly implemented in a way which guarantees correct operation.

Moreover, as we were more concerned with the mathematical aspect of our designs and since our prototype implementation was only intended to demonstrate our actor-based concept, the DistPolyLarva compiler was rapidly developed. Hence we propose to provide a more thorough implementation based on our prototype and on our formal models. In fact we propose that the code of the prototype should be properly structured so as to be more maintainable in the future. Moreover, the synthesised monitoring code can be further optimized so as to reduce the tool's monitoring overhead as much as possible. Additionally, the finalized compiler should also provide better error reporting and error recovery mechanisms which would further aid users to debug their Pseudo-polyLS specification scripts. We also suggest that the proper implementation should also be tested for efficiency and compared with the original PolyLarva implementation.

7 Conclusion

We have sought to increase the understandability and reliability of PolyLarva with the aim of elevating the user's level of confidence in our RV tool. This was done by providing a high-level operational model which describes the runtime behaviour of the core constructs of polyLS. The evaluation for this model consisted in proving that the model describes a deterministic monitoring behaviour. We also created denotational semantics which convert μLarvaScript specifications into Erlang actor expressions. The evaluation of this model consisted in proving the correctness of the formal translation, which permits that any property proved for the high-level model would also apply for the denoted monitoring Actors. This also helps in increasing the user's level of confidence in our tool. This formal translation was then implemented as the DistPolyLarva prototype compiler which guarantees a correct translation for Pseudo-polyLS specifications which only include constructs that are formalized in μLarvaScript.

References

A, M. K. and P, K. (2010). Distributed computing approaches for scalability and high performance.

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

Bauer, A., Leucker, M. and Schallhart, C. (2006). *Runtime Verification for LTL and TLTL.*

Cassar, I. (2013). *Monitoring Distributed Systems with Distributed PolyLarva.* University of Malta.

Colombo, C. (2008). *Practical Runtime Monitoring with Impact Guarantees of Java Programs with Real-Time Constraints* (Master's thesis, University of Malta).

Colombo, C., Francalanza, A., Mizzi, R. and Pace, G. J. (2012). polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries. In *Softw. eng. form. methods - 10th int. conf. sefm 2012* (Vol. 7504, pp. 218–232). Lecture Notes in Computer Science. Springer.

Ericsson AB. (2013). Parse Tools Reference Manual.

Francalanza, A. and Seychell, A. (2013). *Synthesising Correct Concurrent Runtime Monitors in Erlang* (tech. rep. No. CS2013-01). University of Malta.

Gul A., A., Prasannaa, T. and Reza, Z. (2001). *Actors: A Model for Reasoning about Open Distributed Systems*. University of Illinois at Urbana USA.

Gustafson, J. L. (1988). Reevaluating Amdahl's Law. *Commun. ACM*. 31, 532–533.

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

Mizzi, R. (2012). *An Extensible and Configurable Runtime Verification Framework.* (Master's thesis, University of Malta).

Vermeersch, R. (2009, January). Concurrency in Erlang and Scala.