Monitoring Distributed Systems with Distributed POLYLARVA

Ian Cassar, Adrian Francalanza, Christian Colombo  
Faculty of ICT, CS Department  
University of Malta  
Emails: \{ian.cassar.10, adrian.francalanza, christian.colombo\}@um.edu.mt

Abstract—POLYLARVA is a language-agnostic RV tool, which converts a POLYLARVA script 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 calculus 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 calculus into constructs of a formal actor-based model [7], 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.

I. INTRODUCTION

Runtime Verification (RV) [3] is a dynamic [3], [5] 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 [6], [7]. 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.

A. PolyLarva

POLYLARVA [11], [6] is a language agnostic RV compiling tool, which when given an RV specification written in polyLS (short for poly-LarvaScript), 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.

\[\begin{align*}
BR1 &\leftarrow \text{ReqFunds}(\text{Usr}, \text{Sum})/\text{IsUsrValid}(\text{Usr}) \rightarrow \text{WarnUsr}(); \\
BR2 &\leftarrow \text{ReqFunds}(\text{Usr}, \text{Sum})/\text{EnoughFunds}(\text{Sum}) \\
& \rightarrow \text{WarnUsr}(); \\
BR3 &\leftarrow \text{ReqFunds}(\text{Usr}, \text{Sum}) \rightarrow \text{TransferFunds}(\text{Usr}, \text{Sum}); \\
\end{align*}\]

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 \text{e} from the system, it starts by matching it with the event pattern of the first rule in the sequence, i.e., BR1. If \text{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, else the event is matched to the event pattern of Rule BR2. When \text{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, otherwise the rule is ignored and the event would be matched with the pattern of BR2.

B. Problem Definition

There are several problems with the original POLYLARVA [11]:

(i) POLYLARVA was developed using a compiler-driven\footnote{The aim was to develop an actual compiler implementation.} [6] approach, hence no formal language 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.

(ii) 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.
Due to the shared-state, multi-threaded design of the synthesised monitor, POLYLARVA does not provide a foundation on 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.

II. 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 calculus, 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 calculus.

A. The µLarvaScript Calculus

The following µLarvaScript calculus 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. It 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 calculus is defined below.

| Table 3.1. \( M \in \text{Mons} := (s,d) \mid M_0||M_1 \) |
| \( d \in \text{RulesList} := r; d \in \) |
| \( n \in \text{Rule} := ((q,c)\rightarrow a) \) |
| \( s \in \text{State} : \text{Var}^* := \{ x_0 , x_1 , \ldots \} \) |
| \( e \in \text{Event} := n(v_0 \ldots v_k) \) |
| \( t \in \text{EventStream} := e; t \in \) |
| \( q \in \text{Query} := n(\sigma_0 \ldots \sigma_k) \) |
| \( b \in \text{Boolean} := \text{true} \mid \text{false} \) |
| \( c \in \text{Conditions} := b \mid \{c\} \mid c_1 \&\& c_2 \mid p(v_0 \in Val, \ldots , v_k \in Val) \) |
| \( a \in \text{Actions} : (\text{State} \rightarrow \text{State}) := \text{stop} \mid \text{fail} \mid \text{noOp} \mid a_1,a_2 \mid \text{update}(S,F) \mid \text{load}(M) \) |

A monitoring system consists of a collection of concurrent monitors, \( M_0||M_1 \), where each individual monitor, \( (s,d) \), possesses its own current local state “\( s \)” and its own rule list “\( d \)”. Monitors are able to process sequences of events “\( e \)” which are forwarded to the system by the state. The state of a monitor, “\( s \)”, comprises a set of local variables, \( \{x_0,\ldots,x_n\} \), while a rule list, “\( d \)”, consists of a sequence of rules. Each individual rule, of the form “\( ((q,c)\rightarrow a) \)”, 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, “\( c \)”, 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 following example script shows the same rules defined in Example 1.1, written in µLarvaScript syntax:

Example 3.1.

\[
\{\text{usr1, funds}, \{\text{ReqFunds} (\text{Usr}, \text{Sum}), !\text{IsUsrValid} (\text{Usr}) \Rightarrow \text{WarnUsr} ()\}; \{\text{ReqFunds} (\text{Usr}, \text{Sum}), !\text{EnoughFunds} (\text{Sum}) \Rightarrow \text{WarnUsr} ()\}; \{\text{ReqFunds} (\text{Usr}, \text{true}) \Rightarrow \text{TransferFunds} (\text{Usr, Sum})\};
\]

B. 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 (\( \rightarrow \)), and into the low-level monitoring rules, denoted by the low-level relation (\( \Rightarrow \)) 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. They also describe how an event is consumed and removed from the event stream if there exists a single monitor which is capable of consuming that event.

µLarvaScript High-Level Monitoring rules:

\[
\frac{}{t \triangleright M \quad t \triangleright M' \quad \frac{}{r; t \triangleright M \quad r; t \triangleright M'} \quad \frac{}{c; t \triangleright M \quad c; t \triangleright M'} \quad \frac{}{d; t \triangleright M \quad d; t \triangleright M'} \quad \frac{}{t \triangleright M \quad t \triangleright M'} \quad \frac{}{e; t \triangleright M \quad e; t \triangleright M'}
\]

µLarvaScript Low-Level Monitoring rules:

\[
\frac{}{t \triangleright M \quad t \triangleright M' \quad \frac{}{t \triangleright M \quad t \triangleright M'} \quad \frac{}{c; t \triangleright M \quad c; t \triangleright M'} \quad \frac{}{d; t \triangleright M \quad d; t \triangleright M'}
\]

µLarvaScript Event Consumption rules:

\[
\frac{}{\frac{}{r; t \triangleright M \quad r; t \triangleright M'} \quad \frac{}{c; t \triangleright M \quad c; t \triangleright M'} \quad \frac{}{d; t \triangleright M \quad d; t \triangleright M'} \quad \frac{}{t \triangleright M \quad t \triangleright M'} \quad \frac{}{e; t \triangleright M \quad e; t \triangleright M'}
\]

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

µPARMON is a low-level inductive rule which determines whether \( t \triangleright M_0||M_1 \), consisting of a sequence of events “\( e \)” and monitor collection “\( M_0||M_1 \)”, is capable of reducing into “\( t' \triangleright M_0'||M_1 \)”, 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 “\(M_0\)”. R\(M\)\(EVENT\)\(HANDLE\)\(N\) is an axiom which specifies that a monitor, of the form “\((s, d)\)” 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 Evaluation rules.

These rules 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 \(\mu\)\(Larva\)\(Script\) 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 “\(e'\)” to evaluate to true, thus invoking an action “\(a\)” which modifies state “\(s''\)” into some “\(s''\)”.

C. The Single Receiver Property

One of the most prominent properties observed in \(P\)\(OLY\)\(LARVA\) 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. Moreover, we will 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.

\[ t \triangleright M_0 \triangleright M_1 \rightarrow t' \triangleright M'_1 \text{ implies } t \triangleright M_0 \rightarrow t' \triangleright M'_0 \text{ and } t \triangleright M_1 \not\rightarrow \]

III. THE DISTRIBUTED-STATE MODEL AND ITS TRANSLATION

This model aims to provide a formal description of the behaviour of the \(\mu\)\(Larva\)\(Script\) 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 \(\mu\)\(Larva\)\(Script\) constructs to constructs of a formal Actor model for Erlang adapted from [7] by Seychell et al. In this way, the meaning of the \(\mu\)\(Larva\)\(Script\) constructs is given in terms of a highly scalable [10], 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 [9].

A. Concurrency, the Actor Model & Erlang

The Actor Model [8] is a highly scalable paradigm [10] 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 [8], 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 [8], 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 [13], [2], a programming language which natively implements this model.

Although forms of concurrency are employed in the monitors synthesised by \(P\)\(OLY\)\(LARVA\), this is done through multi-threading and shared state communication [11] using explicit locking mechanisms. As these concurrent monitors do not use a distributed state\(^2\), 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 [1].

B. Alternative Semantics for \(\mu\)\(Larva\)\(Script\)

The denotations in Figure 4.1 convert \(\mu\)\(Larva\)\(Script\) constructs into constructs of the formal Actor model for Erlang [7], thus giving Actor semantics to \(\mu\)\(Larva\)\(Script\). 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 \(\mu\)\(Larva\)\(Script\) constructs, for example, \(\text{abc}\) in \([\text{abc}]^m\) refer to a \(\mu\)\(Larva\)\(Script\) construct, while if \(\text{abc}\) is not declared in a denotation, then it is a construct of the Erlang model [7].

\([t \triangleright M]^m\) presents the root denotational function which takes an event stream \(t\) and a \(\mu\)\(Larva\)\(Script\) monitor specification “\(M\)”. It then invokes another denotational function \([t]^m_{\text{par}}\), which creates a coordinating Actor that executes in parallel with the monitoring actors returned by \(\text{fst}(\[M]^m_{\text{par}}\). Moreover, in order for the denotation \([t]^m_{\text{par}}\) to keep on reducing, it requires a list of process identifiers\(^3\) (\(\Pi\)\(D\)\(S\)\)\(s\)) returned by \(\text{snd}(\[M]^m_{\text{par}}\).

The translation \([t]^m_{\text{par}}\) converts an event stream into a coordinating actor, when given a list of \(\Pi\)\(D\)\(S\)\(s\). This special Actor is required to interface with the monitored system and to make sure that the synthesized monitor is behaving in accordance with

\(^2\)“Distributed state” means that each monitor has its own local state and communicate through message passing.

\(^3\)A \(\Pi\)\(D\)\(S\)\( uniquely identifies an Actor.
the Single Receiver Property. In fact, \([t]\_m^\par\) creates an actor with \([t]\_m^\par\) as its mailbox, meaning that the system events will be delivered to the coordinator’s mailbox. Moreover, the coordinator consists of a recursive function which takes a list of PIDs and listens for messages in its mailbox via a \texttt{recv} command. Whenever the coordinator receives the message \{\texttt{new}, \texttt{Pid}\}, it signifies that one of the concurrent monitors has issued a \([\texttt{load}(M)]_m^\par\) action, so as to dynamically create a new concurrent monitor. For this reason, the coordinator adds the PID of the new monitor to its PID-list and issues a recursive call, to restart listening for other messages. Conversely, when the coordinator reads a system event message, \{\texttt{evt}, E\}, it broadcasts the message \(\epsilon_m \equiv \{\texttt{self}, E\}\) to all monitors executing concurrently, by using the “\texttt{broadcast}” function. The coordinator then awaits feedback from the monitors by calling “\texttt{await(count)}”, where “\texttt{count}” is initially set to be the length of the coordinator’s PID-list. Moreover, the “\texttt{await}” function makes use of a selective receive so as to only retrieve feedback messages, of the form “\texttt{ok}” or “\texttt{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.

\[\text{[\texttt{\_m}]_{\text{par}}}\] is a function that converts a \(\mu\)LarvaScript monitor into a meta-level tuple containing a list of monitoring actors together with another list with their PIDs. The meta-functions \(\texttt{fst}\) and \(\texttt{snd}\) are then invoked at compile-time so as to extract the two separate lists from the denoted meta-tuple. Each actor denoted by \([\text{[s, d]}_m^\par\] of \(\text{[\texttt{\_m}]_{\text{par}}}\) is always associated with a unique PID, “\(t\)”, and is initialized with an empty mailbox “\(e\)” so as to wait for event messages of the form \(\texttt{Coord \_\_}\rightarrow \texttt{Coord!}\ \texttt{ok}, (X;state)\). An empty \(\mu\)LarvaScript rule list, is converted by \([d]_m^\par\] 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 “\texttt{Coord! nok}” and leaves the monitor’s current state unmodified.

Each \(\mu\)LarvaScript rule, in a non-empty rule list, is translated through \([\text{[q, c]}_m^\par\rightarrow a]\) into an Erlang guarded command. Whenever the guarded rule’s tuple query, of the form \(\text{[Coord],[\text{[\_m}]_{\text{par}}}]\), matches patterns the structure of the received event in a way which causes condition \([\epsilon]_m^\par\) to return \texttt{true}, the rule sends an “\texttt{ok}” feedback message to the coordinator, which signifies that the event has been handled. It then executes the function denoted by \([a]_m^\par\) on the monitor’s current state, thus generating the next state.

The denotation \([\_m]\), for the monitor’s state, dictates that the monitor’s state variables are converted into a list of Erlang variables. The translation \([\_m]\), states that a \(\mu\)LarvaScript event is translated into an Erlang tuple containing the event name and a tuple of values created by the system, while the query denotation, \([\_m]\), 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 \([\_m]\), converts \(\mu\)LarvaScript conditions into Erlang functions

---

\\(4\)Where \texttt{self} refers to the coordinator’s Pid and \(E\) is the actual system event received.
which return a boolean value after performing a check on the
monitor state passed as its argument. The action denotation
\([-\cdot]^m\), 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.

\[
\mu \text{UsrValid}(\text{Usr}) \rightarrow \text{WarnUsr}()
\]

\[
\text{def } \text{By applying the root denotation } [-\cdot]^m
\]

\[
[[\{\text{Usr}1, \text{funds}\}, (\text{ReqFunds}(\text{Usr}, \text{Sum}),
\text{!UsrValid}(\text{Usr})) \rightarrow \text{WarnUsr}())]; [\text{par}])\]

\[
\text{def } \text{Applying } [-\cdot]^m , \text{and extracting pidList “[i]” with the
}
\text{snd meta function and the actor expression with fst }

\[
[[\{\text{Usr}1, \text{funds}\}, \text{(ReqFunds}(\text{Usr}, \text{Sum}),
\text{!UsrValid}(\text{Usr})); [\text{par}])]]

\[
\text{def } \text{Applying } [-\cdot]^m \text{to create the coordinator}

\[
\text{coord}([\mu \text{rec} \cdot \lambda \text{X}_{\text{state}} \cdot \text{X}_{\text{new}} \rightarrow \text{recv (}
(\text{ReqFunds}(\text{Usr}, \text{Sum}), \text{!UsrValid}(\text{Usr})) \rightarrow \text{WarnUsr}()); [\text{par}])]

\[
\text{coord} \rightarrow \text{Coord! ok, (WarnUsr()}; (\text{X}_{\text{state}}));
\text{Coord, } \rightarrow \text{Coord! nok, (X}_{\text{state}})

\[
\text{def } \text{After applying the necessary denotations }

\[
[[\{\text{Usr}1, \text{funds}\}, (\text{reqrec} \cdot \lambda \text{X}_{\text{state}} \cdot \text{X}_{\text{new}} \rightarrow \text{recv (}
(\text{reqrec} \cdot \lambda \text{X}_{\text{state}} \cdot \text{X}_{\text{new}} \rightarrow \text{recv (}
(\text{reqrec} \cdot \lambda \text{X}_{\text{state}} \cdot \text{X}_{\text{new}} \rightarrow \text{recv (}
(\text{reqrec} \cdot \lambda \text{X}_{\text{state}} \cdot \text{X}_{\text{new}} \rightarrow \text{recv (}
(\text{reqrec} \cdot \lambda \text{X}_{\text{state}} \cdot \text{X}_{\text{new}} \rightarrow \text{recv (}

\[
\text{IV. THE DISTPOLYLARVA Prototype}

DISTPOLYLARVA 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 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.

A. The Compilation Phases

DISTPOLYLARVA passes a given Pseudo-polyLS specifi-
cation 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
total 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
[12].

**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.

V. 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.

A. 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\(^5\), and it receives a specific system event, it will _always_

\(^5_{By “collective state” we refer to the local states of all monitors in the
specified monitor collection._
We showed that the behaviour of $J_\mu$ corresponds to the behaviour of an extended version of our formal models, as well as additional formal results. This extension requires modifications to the high-level reduction, i.e., $[t \triangleright M]_\mu ^m$, which reduces in 0 or more Erlang steps into $[t' \triangleright M']^m$. 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 $[t \triangleright M]_\mu ^m$ in 0 or more Erlang steps into $[t' \triangleright M']^m$.

VI. FUTURE WORK

As part of our future work we propose to extend our $\mu$LarvaScript calculus 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 DISTPOLYARVA 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 DISTPOLYARVA 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 finalised 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 POLYARVA implementation.

REFERENCES


