Elarva Compiler Manual

Rudolph Gatt

May 2011

Contents

1. Introduction **1**

1.1 Elarva 1

1.2 Elarva Design 1

1.2.1 Specification 1

1.2.2 Instrumentation 2

1.2.3 Monitoring 2

1.2.4 Handling Violations 3

1.2.5 Mitigating the impact of verification 4

1.3 Document Outline 4

2. Language Specification **6**

2.1 Introduction 6

2.1.1 Events 6

2.1.2 States 7

2.1.3 Transitions 8

2.1.4 Properties 9

2.1.5 User Registration Example 10

2.1.6 Variables 10

2.1.7 Context 11

2.1.8 Channels 13

3. Elarva Tool **17**

3.1 Introduction 17

3.2 Required Components 17

3.3 Elarva Compiler 18

3.4 Monitoring Code 18

3.4.1 Monitor 19

3.4.2 Supervisor 19

3.4.3 Tracer Process 19

3.4.4 Set of FSMs 21

1. Introduction

1.1 Elarva

The Larva monitoring tool has been successfully applied to a number of industrial Java systems, providing extra assurance of behaviour correctness. Given the increased interest in concurrent programming, we propose Elarva, an adaptation of Larva for monitoring programs written in Erlang, an established industry-strength concurrent language. Object-oriented Larva constructs have been translated to process-oriented setting, and the synchronous Larva monitoring semantics was altered to an asynchronous interpretation.

1.2 Elarva Design

This section illustrates how each runtime verification phase is tackled in Elarva.

1.2.1 Specification

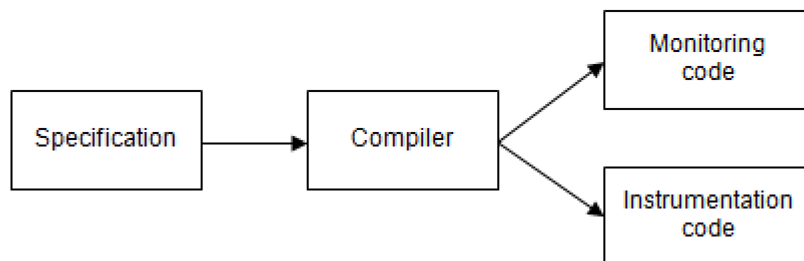

Figure 1.1: *Specification phase.*

As in Larva the chosen specification is DATEs, a logic based on symbolic automata, which offers a high degree of expressivity. Being designed for runtime verification it requires much less computational resources than having a logic to be decidable over all execution paths. Figure 1.1 depicts the specification

being fed to the Elarva compiler which transforms it into monitoring code and instrumentation code.

1.2.2 Instrumentation

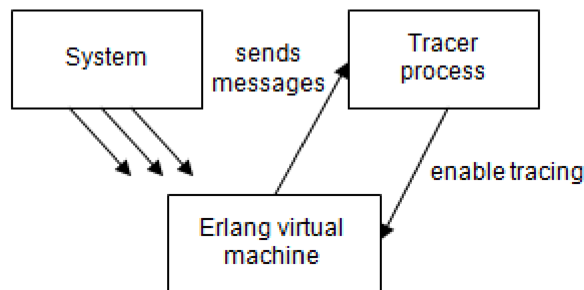

Figure 1.2: *Instrumentation phase.*

In Larva, the instrumentation is tackled via aspect-oriented programming which uses joinpoints (i.e. identifiable points in the code) defined by a pointcut (i.e. rule to which a number of joinpoints should match) to capture events. This is achieved because Java, the implementation language used in Larva, supports aspect-oriented programming through AspectJ. Message passing concurrency languages do not support aspect-oriented programming therefore we had to use another appraoch. We had two options, either use hand instrumentation i.e. inserting specific code in the target system for each occurrence of a particular event, or using the tracing mechanism of Erlang. We opted for the Erlang tracing mechanism because it gives us an elegant way how to capture events without modfying the target system while hand instrumentation would result in the cluttering of the target system. Figure 1.2 shows us the instrumentation phase where the tracer process enables the Erlang tracing mechanism and begins to receive trace messages by the Erlang virtual machine.

1.2.3 Monitoring

The Erlang tracing mechanism does not suspend the system while a trace event is triggered therefore the target system will continue its execution while the Erlang tracing mechanism generates trace events and sends them to the tracer process. This requires us to adopt an asynchronous monitoring mode where the target system and the monitoring code are both running in parallel but there exists a delay between the occurence of events and the delivery of trace messages.

As we can see from figure 1.3, when the Erlang tracing mechanism is activated the Erlang virtual machine sends trace messages to a tracer process. This tracer process is the central and most important part of our monitoring code. The Erlang

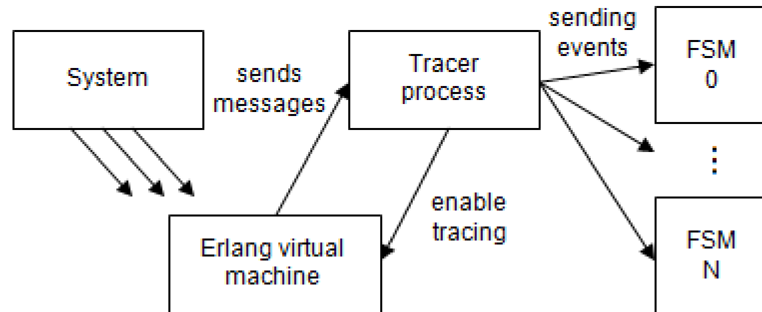

Figure 1.3: *Monitoring phase.*

virtual machine can only send trace messages to a single process thus at any one time there is only one tracer process. As a consequenece, the tracer process is responsbile for receiving trace messages and generating meaningful events which are forwarded to the appropiate FSMs. If the erlang virtual machine could send trace messages to more than one process we could eliminate the tracer process and let each FSM receive its pertaining events.

1.2.4 Handling Violations

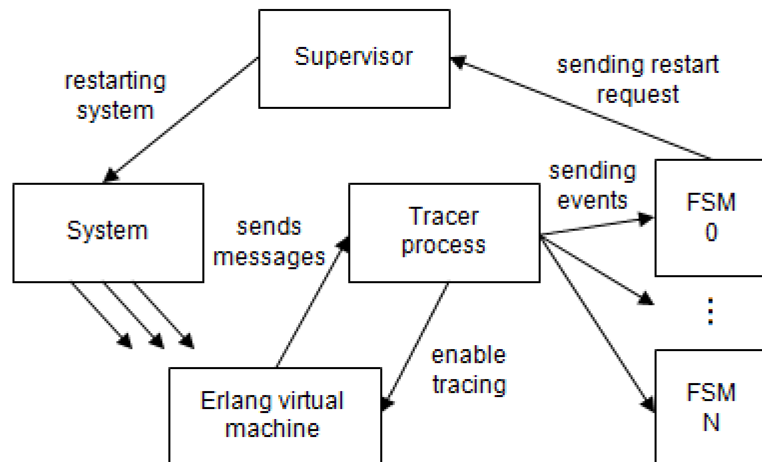

Figure 1.4: *Handling Violations phase.*

Since we adopted an asynchronous monitoring mode, we are restricted with respect to the actions that can be taken upon a violation detection because upon a violation detection the target system will continue execution with the possibility of doing more damage (for example exposing sensitive data etc.). Even though

we cannot take an immediate action upon a violation detection, figure 1.4 shows us how we can take action via the Supervisor behaviour which can restart the target system to limit the damage. If the target system is designed with built-in recovery strategies (i.e. routines that can rearrange the system's state), we can trigger these routines (for example by a function call or by a message) as an action upon violation detection.

1.2.5 Mitigating the impact of verification

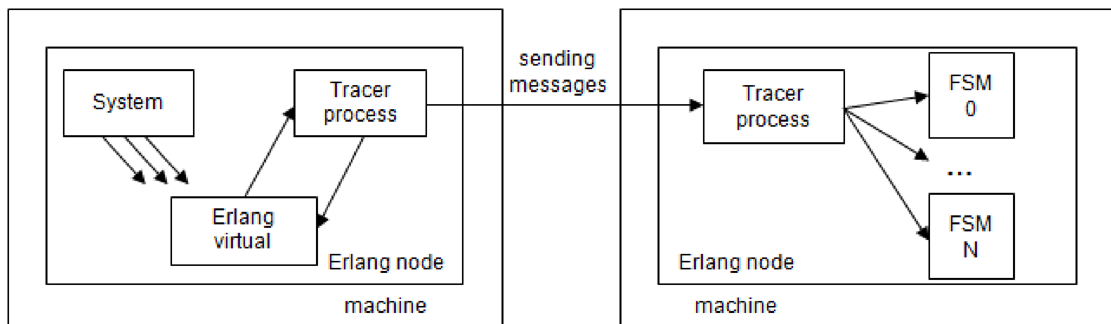

Figure 1.5: *Mitigating the impact of verification phase.*

The monitoring code induces an amount of overhead on the target system. We have to minimise this overhead as much as possible. One way to minimise the induced overhead is to adopt a distributed monitoring approach where the target system runs on Erlang node while the monitoring code resides on a different Erlang node and on a separate machine with dedicated hardware. Figure 1.5 shows us two erlang nodes running on two separate machines. The first machine hosts the target system and a tracer process which receives trace messages from the Erlang virtual machine. The tracer process on the first machine forwards all trace messages to the other tracer process situated on the second machine which generated meaningful events and forwards them to the appropiate FSMs. It would be ideal if the Erlang virtual machine on the first Erlang node sends the trace messages directly to the tracer process on the second machine however this setting cannot be achieved because the tracer process must be a local process.

1.3 Document Outline

This manual is organised in two main parts:

- *Language Specification* - This part explains the language specification and syntax of each construct used in Elarva.

- *Elarva Tool* - This part outlines the required components to use the Elarva tool and also explains each component generated by the Elarva tool.

2. Language Specification

2.1 Introduction

The following sections give us an outline of the language specification and syntax used in Elarva. An example of a user registration system, which allows users to log in and use any provided services, is introduced bit by bit in each construct to aid the reader to fully-understand the language specification used in Elarva and how each construct can be used to monitor a particular target system.

2.1.1 Events

The alphabet of a DATE's automaton are events where each event represents an occurence of something meaningful in the target system. An event in an object-oriented langauge might be a method call or an exception throw while an event in a concurrent language might be a function call or the spawning of a process. We have to be able to distinguish between events therefore each event must be given a unique name. The events must be specified in a block with the following format:

```
EVENTS{
...
}EVENTS
```

The event types that Elarva can handle are `SEND, RECEIVE, CALL, LINK, UNLINK, REGISTER, UNREGISTER, SPAWN` and `CHANNEL.` We must also specify the relevant arguments in each event. In a `SEND` or `RECEIVE` event type, the specified arguments must reflect the format of the message being sent or received respectively. In the `CALL, LINK, UNLINK,` and `SPAWN` event types, the specified arguments must match the MFA tuple `{Module, Function, Arity}` where in the `CALL` event type the tuple specifies which function is being called and in the other events types the tuple specifies the initial function call of the process either begin linked/unlinked or the process being spawned. In the `REGISTER` and `UNREGISTER` event types, the specified arguments must match the name being registered or unregistered respectively. So far an event declaration should be specified as follows:

```
eventName = eventType <arguments>
```

We can also target specific processes by specifying the MFA tuple before the arguments. The MFA tuple matches with the initial function call made by the process. Therefore the event declaration changes as follows:

```
eventName = eventType <{Module, Function, Arity> <arguments>
```

It is often convenient to be able to use wildcards to capture events that might not be identical but have some similarities or where some information is only available at runtime. This can be achieved by using an underscore ‘ _ ’ which would act as a placeholder. Wildcards can be used inside arguments or instead of the MFA tuple specifying the initial call of the process. The following are examples of events using wildcards:

```
receiveAllMessagesEvent = RECEIVE <{module, test, 0> <_>
callEvent = CALL <_> <{module, test, 0}>
```

The following events block illustrates the events that a user can make in the user registration system:

```
EVENTS{
  login = SEND <_> <{login, User, Pass}>
  logout = SEND <_> <{logout, User, Pass}>
  request = SEND <_> <{request, User}>
}EVENTS
```

The BNF notation of the Event construct is given as follows:

```
Name        ::= [a-zA-Z0-9]
Messages    ::= Erlang messages
Module      ::= [a-zA-Z0-9]
Function    ::= [a-zA-Z0-9]
Arity       ::= [0-9]
EventName   ::= identifier
InitialCall ::= ‘{’ Module ‘,’ Function ‘,’ Arity ‘}’ | ‘_’
Arguments   ::= Name | Messages | InitialCall | ‘_’
EventType   ::= ‘SEND’ | ‘RECEIVE’ | ‘CALL’ | ‘LINK’ | ‘UNLINK’ | ‘REGISTER’ |
                ‘UNREGISTER’ | ‘SPAWN’ | ‘CHANNEL’
Event       ::= EventName ‘=’ EventType ‘<’ InitialCall ‘>’ ‘<’ Arguments ‘>’
Events      ::= Event Events | Event
EventsBlock ::= ‘EVENTS{’ Events ‘}EVENTS’
```

2.1.2 States

A state dictates the current status of the DATE’s automaton. States are divided into a starting state, normal states and bad states. Each type of state must be enclosed by a block. The following states block illustrates all the possible user process states during its lifecycle:

```
STATES{
  BAD{ violation }BAD
  NORMAL{ loggedIn }NORMAL
  STARTING{ loggedOut }STARTING
}STATES
```

The BNF notation of the State construct is given as follows:

```
StateName  ::= identifier
StateDecl  ::= StateName
StateList  ::= e | StateDecl StateList
Bad        ::= ‘BAD{’ StateList ‘}Bad’
Normal     ::= ‘NORMAL{’ StateList ‘}NORMAL’
Starting   ::= ‘STARTING{’ StateDecl ‘}STARTING’
StateBlock ::= ‘STATES{’ Bad Normal Starting ‘}STATES’
```

2.1.3 Transitions

A transition causes a state change in the DATE's automaton and has the following format:

```
source -> destination [event \ condition \ action]
```

The source refers to the current state while the destination refers to which state the DATE's automaton moves. Each transition is trigerred by an event therefore we must specify an event name. We can also specify a condition and an action where a transition is taken only if the condition is satisfied and the action is executed upon taking the transition. The condition must be a boolean expression and can access information from the event's argument. The action can have four different formats which are:

```
{code, erlang code}
{restart}
{variable, get(...)/put(..., ...)}
{channel, [...]}
```

The first format allows the user to specify any Erlang code that he wants to be executed. The second format is used to restart the target system via the Supervisor behaviour. The third and fourth formats allows the user to use variables and channels respectively which will be explained in subsequent sections. Transitions must be enclosed in the following block:

```
TRANSITIONS{
...
}TRANSITIONS
```

The following transisitions block illustrates the transitions that a user process can take:

```
TRANSITIONS{
  loggedOut -> loggedIn [login \\]
  loggedIn -> loggedOut [logout \\]
  loggedIn -> loggedIn [request \\]
  loggedOut -> violation [request \\]
}TRANSITIONS
```

The last transition is the most important one because it dictates how a violation is reached when a `requestReset` event is triggered while the state is `loggedOut`. The BNF notation of the Transition construct is given as follows:

```
StateName       ::= identifier
EventName       ::= identifier
Condition       ::= BooleanExp
Code            ::= erlang code
Variable        ::= explained in subsequent section
ChannelList     ::= explained in subsequent section
Action          ::= ‘{’ ‘code’ ‘,’ Code ‘}’ |
                    ‘{’ ‘restart’ ‘}’ |
                    ‘{’ ‘variable’ ‘,’ Variable ‘}’ |
                    ‘{’ ‘channel’ ‘,’ ChannelList ‘}’
ActionList      ::= Action | Action ActionList
ActionBlock     ::= ‘[’ ActionList ‘]’
Transition      ::= StateName ‘->’ StateName ‘[’ EventName ‘\’ ‘\’ ‘]’ |
              StateName ‘->’ StateName ‘[’ EventName ‘\’ Condition ‘\’ ‘]’ |
              StateName ‘->’ StateName ‘[’ EventName ‘\’ ‘\’ Action ‘]’ |
              StateName ‘->’ StateName ‘[’ EventName ‘\’ Condition ‘\’ Action ‘]’
TransitionList  ::= Transition | Transition TransitionList
TransitionBlock ::= ‘TRANSITIONS{’ TransitionList ‘}TRANSITIONS’
```

2.1.4 Properties

A property represents a system behaviour that we need to monitor and is made up of events, states and transitions which form a DATE's automaton. Each property must be given a unique name to be able to distinguish between one property and another when monitoring multiple properties. Hence the format of the property monitoring a user is given as follows:

```
PROPERTY user {
  EVENTS{ ... }EVENTS
  STATES{ ... }STATES
  TRANSITIONS{ ... }TRANSITIONS
}PROPERTY
```

The BNF notation of the Property construct is given as follows:

```
PropertyName  ::= identifier
Property      ::= ‘PROPERTY’ PropertyName ‘{’ EventsBlock StateBlock
                    TransitionBlock ‘}PROPERTY’
PropertyBlock ::= Property PropertyBlock | ϵ
```

2.1.5 User Registration Example

As from section 4.3 we have been introducing an example of how we can monitor a user registration system. We will now present the whole script together with the respective automaton in figure 2.1 and continue building upon this example to illustrate the *variable*, *context* and *channel* constructs in the upcoming sections.

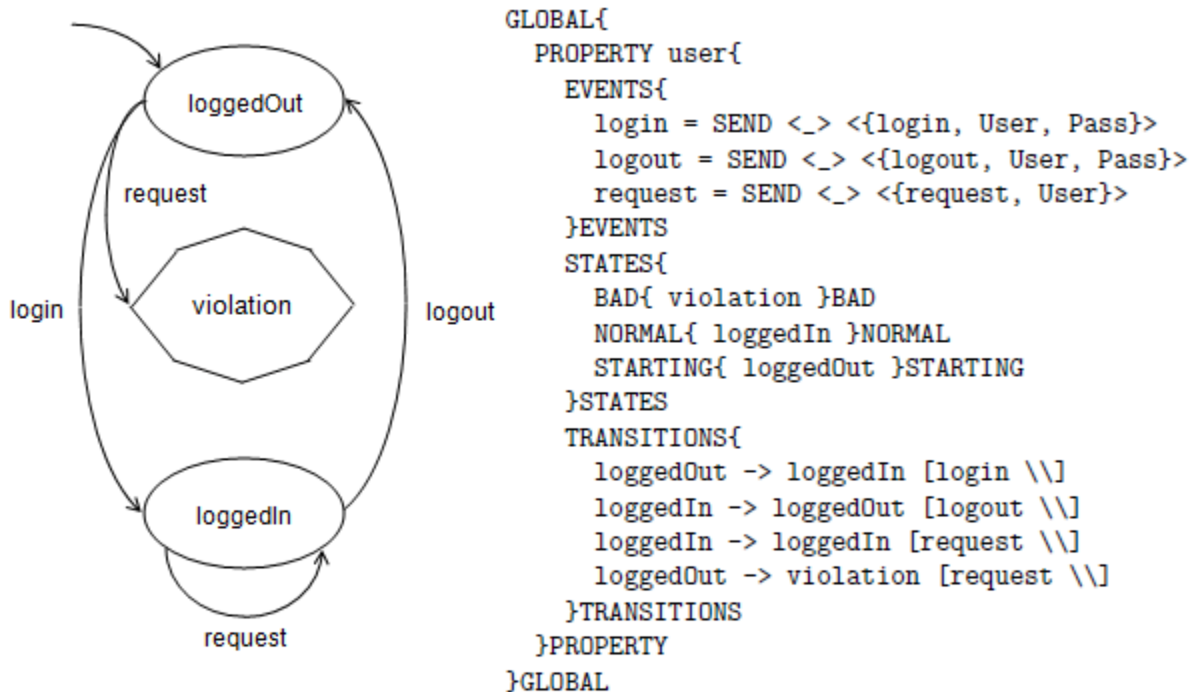

Figure 2.1: *Automaton and Elarva script for user registration example.*

2.1.6 Variables

There exists situations where the user might need to keep some information acquired from an event's arguments to be able to monitor a property. As a consequence, each property has an internal state which can hold variables. A user can create a variable by using the following format in a transition's action as follows:

```
source -> destination [event \ \ {variable, put(variableName, variableValue)}]
```

A stored variable can be used both in the condition and action and can be retrieved as follows:

```
... get(variableName)
```

We continue to extend the user registration example by adding a *variable* action to the first transition as follows:

```
TRANSITIONS{
  loggedOut -> loggedIn [login \\{variable, put(user, User)}]
  ...
}TRANSITIONS
```

When the transisition is taken, the argument `User` is retrieved from the message sent by the user and saved under the identifier `user.` The BNF notation of the Variable construct is given as follows:

```
VariableName  ::= identifier
VariableValue ::= erlang term
Variable      ::= 'get(' VariableName ')' |
                  'put(' VariableName ',' VariableValue ')'
```

2.1.7 Context

Properties have to be enclosed in one container called the global context as follows:

```
GLOBAL{
  PROPERTY propertyName { ... }PROPERTY
  ...
}GLOBAL
```

Properties declared under the global context act on the whole target system. However we can specify properties for each process through the use of the foreach context where processes are distinguishable by the initial function called on spawning. Hence a foreach context should be specified as follows:

```
FOREACH <{Module, Function, Arguments> {
  PROPERTY propertyName { ... }PROPERTY
  ...
}FOREACH
```

The arguments support wildcards through the use of an underscore, '_', which acts as a placeholder. Figure 2.2 illustrates the differences between properties declared under the global context and properties declared under a foreach context. Another important difference is that properties declared under the global context are uniquely identifiable by their property name while properties declared under a foreach context are uniquely identifiable by their property name and the process's Pid that they are monitoring.

So far the user registration example was able to monitor one user at a time since only one automaton was being created. Through the use of the *context* construct an automaton can be created for each user process spawned, identified by the MFA tuple passed in the foreach declaration. The following change in the script allows each user process spawned to be monitored by an individual automaton:

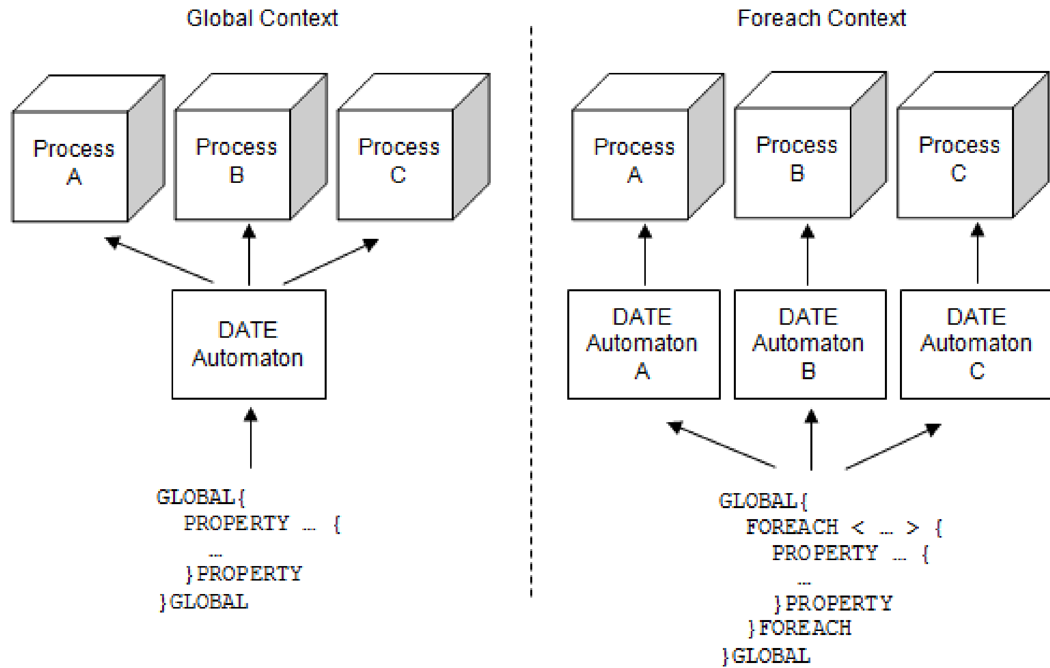

Figure 2.2: *Global Context vs Foreach Context.*

```
GLOBAL{
  FOREACH <{user, new, []> {
    PROPERTY user{
      EVENTS{
        login = SEND <{login, User, Pass}>
        logout = SEND <{logout, User, Pass}>
        request = SEND <{request, User}>
      }EVENTS
      ...
    }PROPERTY
  }FOREACH
}GLOBAL
```

The property `user` is now declared under a foreach context. Events under a foreach context cannot be specified with the MFA tuple before the arguments since we already know which initial function the process has executed on spawning. The BNF notation of the Context construct is given as follows:

```
Argument      ::= erlang term
Arguments     ::= [Argument]
Event         ::= EventName '=' EventType '<' Arguments '>'
Foreach       ::= 'FOREACH' '<' Module ',' Function ',' Arguments '{'
                   PropertyBlock '}FOREACH'
ForeachBlock  ::= Foreach ForeachBlock | ϵ
GlobalContext ::= 'GLOBAL' PropertyBlock ForeachBlock 'GLOBAL'
```

2.1.8 Channels

DATEs automata can communicate together by sending messages to each other denoted as the Channel construct. These messages are used to either trigger target system events or to trigger artificial events i.e. events which do not belong to the target system. Channels can be used in a transition's action as follows:

```
source -> destination [event \ \ {channel, [{channelType, property_name, event]}]
```

We can trigger a target system event by passing the event name and any other related arguments. If we want to trigger an artificial event we first have to declare the event in the events block as follows:

```
EVENTS{
  eventName = CHANNEL <{Module, Function, Arity> <arguments>
  ...
}EVENTS
```

This aids the user to distinguish which events are generated by the target system and those that are generated by the automata. A channel is divided into three types:

```
{global, property_name, event}
{parent, property_name, event}
{foreach, property_name, event}
```

As shown in figure 2.3, the *global* type is used to send an event to a property which is declared under the global context, which can be uniquely identified by its property name.

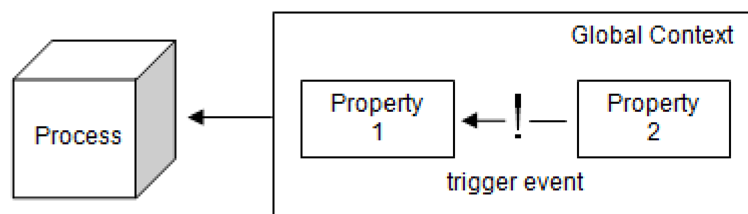

Figure 2.3: *Global channel type.*

The *parent* type can be used when we are using multiple properties under a foreach context. The property receving the channel message can be uniquely identified by its property name and by the process's PID being monitored, both known by the property sending the channel message. Figure 2.4 depicts a property using the *parent* channel type to communicate with another property declared in the same foreach context.

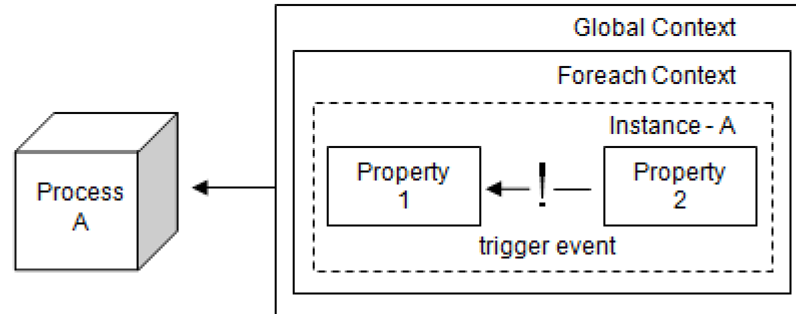

Figure 2.4: *Parent channel type.*

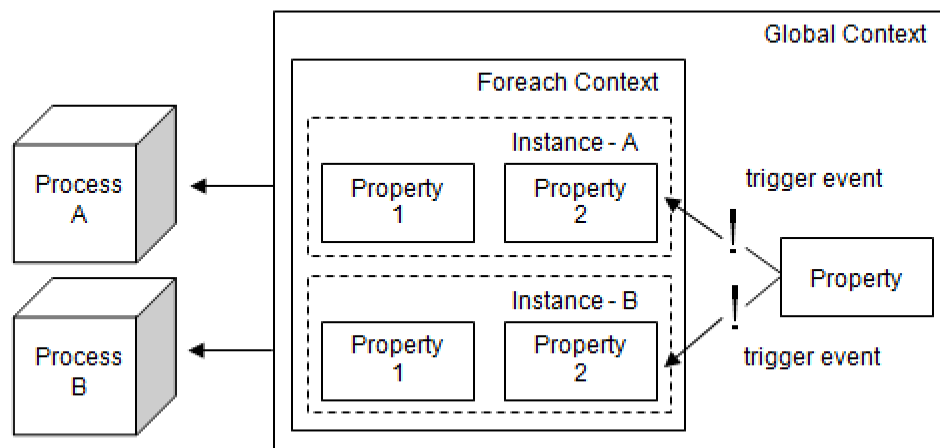

Figure 2.5: *Foreach channel type - Property sending channel event is declared under the Global context.*

The last channel type is the *foreach* which is used to allow a property to communicate with another property declared under a foreach context, having separate contexts. In this channel type we cannot uniquely identify a property because we need both the property name and the process's Pid and we only know the property name since the two properties have separate contexts. As a consequence, we have to send the channel message to all properties monitoring the same behaviour, identified by the property name, for each process instance. In figure 2.5, a property declared under the global context is using the channel construct to trigger an event in another property, declared under a foreach context. The channel event must be triggered on all foreach instances because the property declared under the global context does not have any knowledge of the process's PID. The same applies if the property sending the channel message is declared under a separate foreach context than the foreach context of the property receiving the channel message, as depicted in 2.6. The user should specify

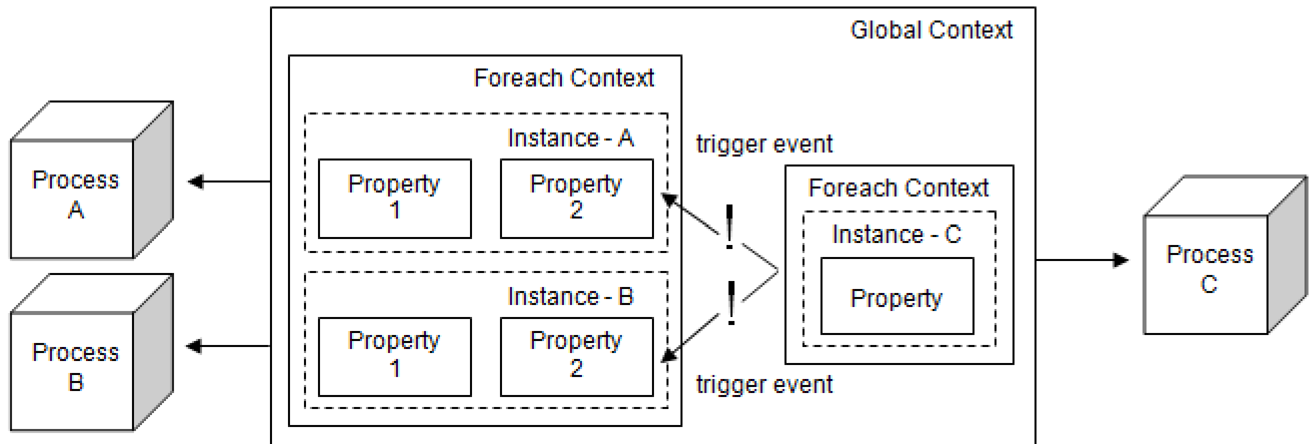

Figure 2.6: *Foreach channel type - Property sending channel event is declared under a Foreach context.*

a condition in the property receiving the channel message to be able to target an individual property.

We will extend further more the user registration example, to illustrate how the *channel* construct can be used, by allowing the system to upgrade itself while running. While the system is upgrading, users cannot request any service therefore if a user sends a request while the system is upgrading a violation is triggered. To be able to monitor this scenario we have to extend the property `user` by the following changes:

```
...
EVENTS{
  ...
  upgrade = CHANNEL <upgrade>
  resume = CHANNEL <resume>
}EVENTS
STATES{
  ...
  NORMAL{
    ...
    upgrading
  }NORMAL
}STATES
TRANSITIONS{
  ...
  loggedIn -> upgrading [upgrade \\]
  upgrading -> loggedIn [resume \\]
  upgrading -> violation [request \\]
}TRANSITIONS
...
```

We also have to include another property which will monitor the system and check whether the system is currently running or upgrading. The need for this property arises from the fact that property `user` is declared under a foreach context, which can only receive events generated by a user process, and therefore cannot be able to detect whether the system is currently running or upgrading. The property which acts on the system is given as follows:

```
...
PROPERTY system {
  EVENTS{
    upgrade = CALL <{system, start, 0}> <{system, upgrade, 0}
    resume = CALL <{system, start, 0}> <{system, resume, 0}
  }EVENTS
  STATES{
    BAD{ }BAD
    NORMAL{ upgrading }NORMAL
    STARTING{ running }STARTING
  }STATES
  TRANSITIONS{
    running -> upgrading [upgrade \\ {channel, [{foreach, user,
{'upgrade', upgrade}}]}]
    upgrading -> running [resume \\ {channel, [{foreach, user,
{'resume', resume}}]}]
  }TRANSIIONS
}PROPERTY
...
```

The property `system` will monitor the system and send channel events to all instances of the property `user` when the system begins upgrading and when the system is resumed.

The BNF notation of the Channel construct is given as follows:

```
EventData   ::= '{' EventName ',' erlang term '}'
Channel     ::= '{' 'global' ',' PropertyName ',' EventData '}' |
                '{' 'parent' ',' PropertyName ',' EventData '}' |
                '{' 'foreach' ',' PropertyName ',' EventData '}'
ChannelList ::= [Channel]
```

3. Elarva Tool

3.1 Introduction

In this chapter we outline the required components, in section 3.2, needed to use Elarva for monitoring a target system. We continue by explaining the architecture of the Elarva compiler in section 3.3 and illustrate each entity that make up the monitoring code in section 3.4.

3.2 Required Components

In order to monitor a target system with Elarva the user needs the following components:

- Target system - the system to be monitored which must be implemeted as a gen_server.
- Specification - DATE properties written in Elarva script.
- Elarva compiler - an Erlang system which given an Elarva script generates the necessary monitoring code for monitoring the target system.

The following script gives us an example of an Elarva script which will be used to monitor a client-server system where users can login onto a server and use its services:

```
GLOBAL{
  PROPERTY user{
    EVENTS{
      login = CALL <{user, new, 1}> <{user, login, 2}>
      logout = CALL <{user, new, 1}> <{user, logout, 2}>
      request = SEND <{user, new, 1}> <{request, Service}>
    }EVENTS
    STATES{
      BAD{ violation }BAD
      NORMAL{ loggedIn }NORMAL
```

```
      STARTING{ loggedOut }STARTING
    }STATES
    TRANSITIONS{
      loggedOut -> loggedIn [login \\]
      loggedIn -> loggedOut [logout \\]
      loggedIn -> loggedIn [request \\]
      loggedOut -> violation [request \\]
    }TRANSITIONS
  }PROPERTY
}GLOBAL
```

We will use the above script to illustrate how the monitoring code is generated and implemented.

3.3 Elarva Compiler

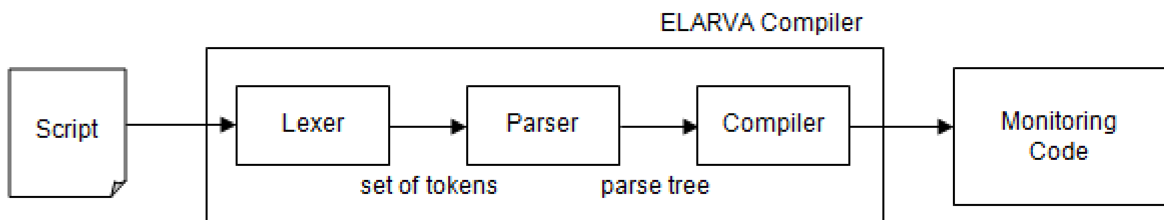

Figure 3.1: *Elarva Compiler.*

As we can see from figure 3.1, the Elarva compiler is divided into three parts: a lexer, a parser and the actual compiler. The user can use the compiler by executing the following function:

```
compiler:compile(system_name, script_path).
```

The script is first passed to the lexer, implemented by leex (a regular expression based lexical analyzer generator for Erlang), which converts the script to a set of tokens. The set of tokens is then passed to the parser, implemented by yeec (a parse-tree generator for Erlang), which generates a parse tree. Finally the parse tree is passed to the actual compiler which generates the necessary monitoring code.

3.4 Monitoring Code

Figure 3.2 illustrates us the entities that make up the monitoring code. In the following sections we explain each entity's responsabilities and any necessary implementation issues.

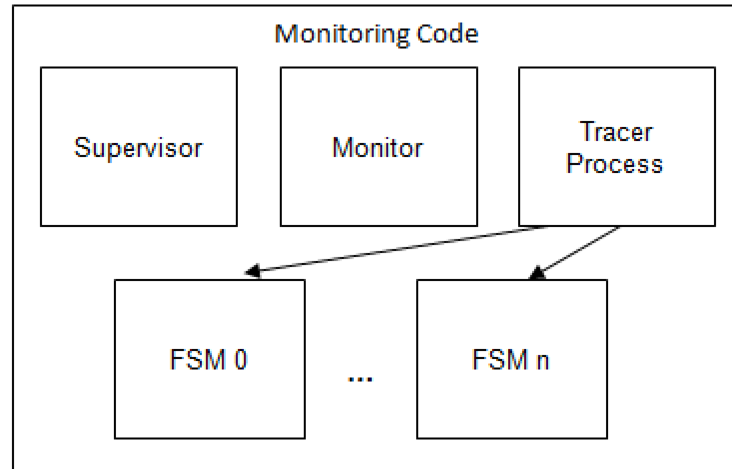

Figure 3.2: *Monitoring code structure.*

3.4.1 Monitor

The monitor is a module used to start the global properties and the supervisor, which in turn starts the system and the tracer process. The user has to call the function `monitor:start/0` to start the monitoring. The main reason why we opted to start the global properties in the monitor module is to allow each global property to be registered before the system is started since registering a process might generate a bottleneck if the system is already started.

3.4.2 Supervisor

The supervisor is responsible for starting the tracer process and the system. This approach allows the system and the tracer process to be restared if any one of them terminates abnormally. The supervisor can also be used to terminate and restart the system if a violation is detected by an automaton.

3.4.3 Tracer Process

The tracer process is started by the supervisor and is responsible for enabling the Erlang tracing mechanism and for receiving trace messages sent from the Erlang virtual machine. The tracer process is mainly divided into two parts. The first part initialises the Erlang tracing mechanism while the second part, the trace loop, takes care of receving and processing trace messages.

The first part enables the Erlang tracing mechanism by calling the `erlang:tr--ace/3` function. The first argument takes the Pid of the process that is going to be traced, the second argument takes a boolean value and the third argument takes a flag list. We pass the system's Pid, that has already been started, as the

first argument. The second argument represents whether we want to enable or disable tracing, therefore we pass the boolean value `true`. As last argument we pass a flag list that is build upon the events specified in the script. If we consider the script presented in the section 3.2, the Erlang tracing mechanism is enabled as follows:

```
erlang:trace{whereis(system), true, [set_on_spawn, call, send]),
erlang:trace_pattern({user, login, 2}, true, [local]),
erlang:trace_pattern({user, logout, 2}, true, [local]),
```

The events types in the script are `CALL` and `SEND`, therefore we specify the call and send flags in the flag list. The set_on_spawn flag is used to allow any child process to inherit the flags of its parent, including the set_on_spawn flag. In this way any process created by the target system can be traced by the Erlang virtual machine. We also make use of the 'receive' and procs trace flags when the user specifies an event with type `RECEIVE` and one of `LINK, UNLINK, REGISTER, UNREGISTER` and `SPAWN` respectively in the script. The `erlang:trace_pattern/3` function is used to determine the set of functions that are going to be monitored.

The second part of the tracer process is the `trace_loop/0` function which receives the trace messages sent from the Erlang virtual machine and generates meaningful events that are forwarded to the appropiate automata. The Erlang virtual machine can send trace messages only to one process therefore we had to adopt a centralized approach where the tracer process receives all trace messages and forwards them to the appropiate automata. The ideal scenario would be to have a decentralized approach where the Erlang virtual machine would send messages to each automaton and each automaton only receives the events pertaining to it. Figure 3.3 shows us the difference between a centralized approach and a decentralized approach.

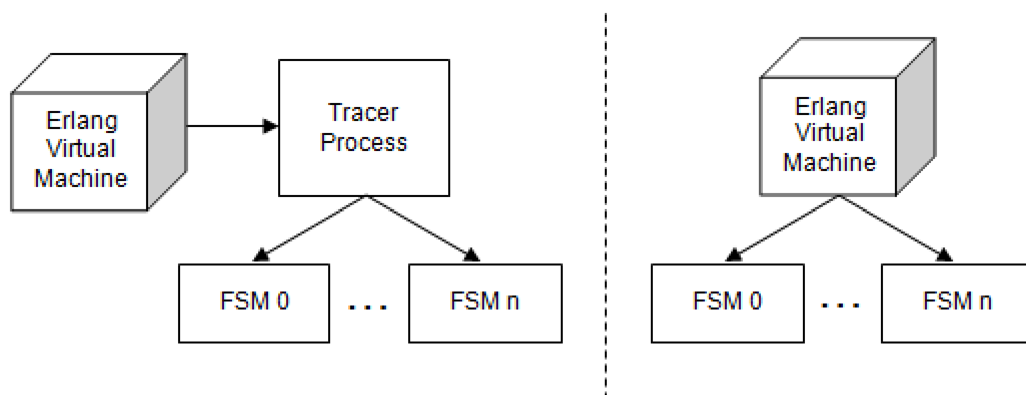

Figure 3.3: *Centralized approach vs. Decentralized approach.*

The following shows us part of the `trace_loop/0` function generated from the script used in the section 3.2:

```
trace_loop() ->
  receive Trace ->
    case Trace of
      {trace, Pid, call, {user, login, A}, _} ->
        gen_fsm:send_event(system_user_fsm, {'login', A}),
        trace_loop();
      {trace, Pid, send, {request, Service}, _} ->
        gen_fsm:send_event(system_user_fsm, {'request', {request, Service}}),
        trace_loop();
       ...
      _ ->
        trace_loop()
    end
  end.
```

For each event found in the script a pattern is written in the case block. When a trace message is received the function tries to match the trace message with a pattern. When a pattern matches with the recieved message an event is sent to the appropiate automaton.

3.4.4 Set of FSMs

Each property defined in the script has to be monitored by an automaton. We opted to implement automata with the gen_fsm behaviour to make good use of the OTP framework and take benefit of its advantages. Therefore for each property declared in the script the Elarva compiler generates a gen_fsm module. The following is a part of the gen_fsm, generated for the script used in section 3.2:

```
-module(system_user_fsm).
-behaviour(gen_fsm)
...
init(State)
  {ok, loggedOut, State}.

loggedOut(Event, State) ->
  ...
  case Event of
    {'login', A} ->
      {next_state, loggedIn, State};
  ...
```

The `init/1` function returns a tuple that is used to specify the start state of the FSM, `loggedOut` in our case, and to initialize any state data needed in the life cycle of the FSM. Furthermore a function is created for each state specified in the script. When an event is sent to the gen_fsm the function named after the current state is executed. Therefore when a `login` event is sent to this gen_fsm and the current state is `loggedOut` the state moves to the `loggedIn` state.