Transforming Erlang finite state machines

Dániel Lukács, Melinda Tóth, István Bozó
dlukacs@caesar.elte.hu, tothmelinda@caesar.elte.hu,
bozoistvan@caesar.elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

Model driven development approaches help to alleviate the abstraction gap between high-level design and actual implementation, to aid design, development and maintenance of industrial scale software systems. To provide automatic, easily usable tools for stakeholders, model driven development essentially relies on efficient and expressive translations between the program source code and the model.

We present a declarative, rule-based approach to deterministically transform Erlang program sources that satisfy a certain syntactical constraint, into valid UML models of state machines. The transformation relies only on static analysis techniques, and the produced model conforms to the state machine metamodel defined in OMG UML 2.0.

1 Introduction

In industrial settings there is an always present necessity to use large software applications with over millions of lines of source code. During design, development and maintenance of software, managing the complexity emerging from these large volumes becomes a critical issue in order to realise a successful product lifecycle. The most important resource needed to manage complexity is relevant, up-to-date information, commonly manifested as the design, development and user documentations. As the software evolves during its development and usage, documentations also have to be actualised to mirror the collective knowledge that was incorporated in the system during its changes. To implement this resource intensive process with the highest efficiency, organisations concerned with software development employ various automatic software analysis tools and machine processable documents to support the developers in creating documentations and keeping them up to date. These tools can also be extremely useful in cases of undocumented legacy systems, where maintainers have to collect information about the inner workings of the system from scratch. One of the more successful methodology to develop, analyse, store and process this kind of information is Model Driven Development (MDD) [20], which utilises models specified by both human and machine readable documents in order to represent the various aspects of these systems. MDD methodology uses sophisticated software solutions to create these models, and to aid in querying valuable information during the analysis of these models and the system itself.

In this paper, we concern ourself with state machine models, that can be used to model event-driven systems. Specifically, we introduce here a method to generate formal UML [7] state machine models from executable Erlang state machine source code. Formal UML models can be readily transformed into human readable diagrams, they are suited to calculate various model metrics on them, and there exist extensions of UML that can be used to specify executable models [20]. First, this transformation makes use of the RefactorErl static analysis

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

framework [16, 11, 24] to analyse the application source code, then it will transform and synthesise the program representation resulting from the analysis, into an UML state machine model. We specified the transformation by defining an algorithm that utilises backtracking and graph pattern matching of certain sets of transformation rules. To generalise the algorithm, we encapsulated all the RefactorErl specific logic into the transformation rules, thus separating the general operation principles of the method from the implementation specific details. This approach also makes it easier to extend the capability of the algorithm by adding more rules. To test our design in practice we also created a reference implementation, and, as presented in Section 6, used this to successfully transform several Erlang state machines selected from the source code of large, popular, open source Erlang applications.

The rest of this paper is structured as follows. At first, Section 2 describes UML and Erlang state machines. Sections 3 and 4 introduce the methodology and the transformation rules to generate UML state machines. In Section 5 we explain the transformation starting from an example Erlang source code. Section 6 presents the evaluation of our work on several open source projects. Finally, Sections 7 and 8 present related work and conclude the paper.

2 Background

One way to represent the execution history of a computer program is to take snapshots of the state of the program memory. State machines can be used to abstract away this low level representation. With state machines these memory snapshots are taken upon the occurrence of certain events, and instead of storing the content of the memory, we just store descriptive labels, called states. Therefore, a state describes a segment of the program behaviour, while a state transition describes a change in such behaviours, usually triggered by an event [21].

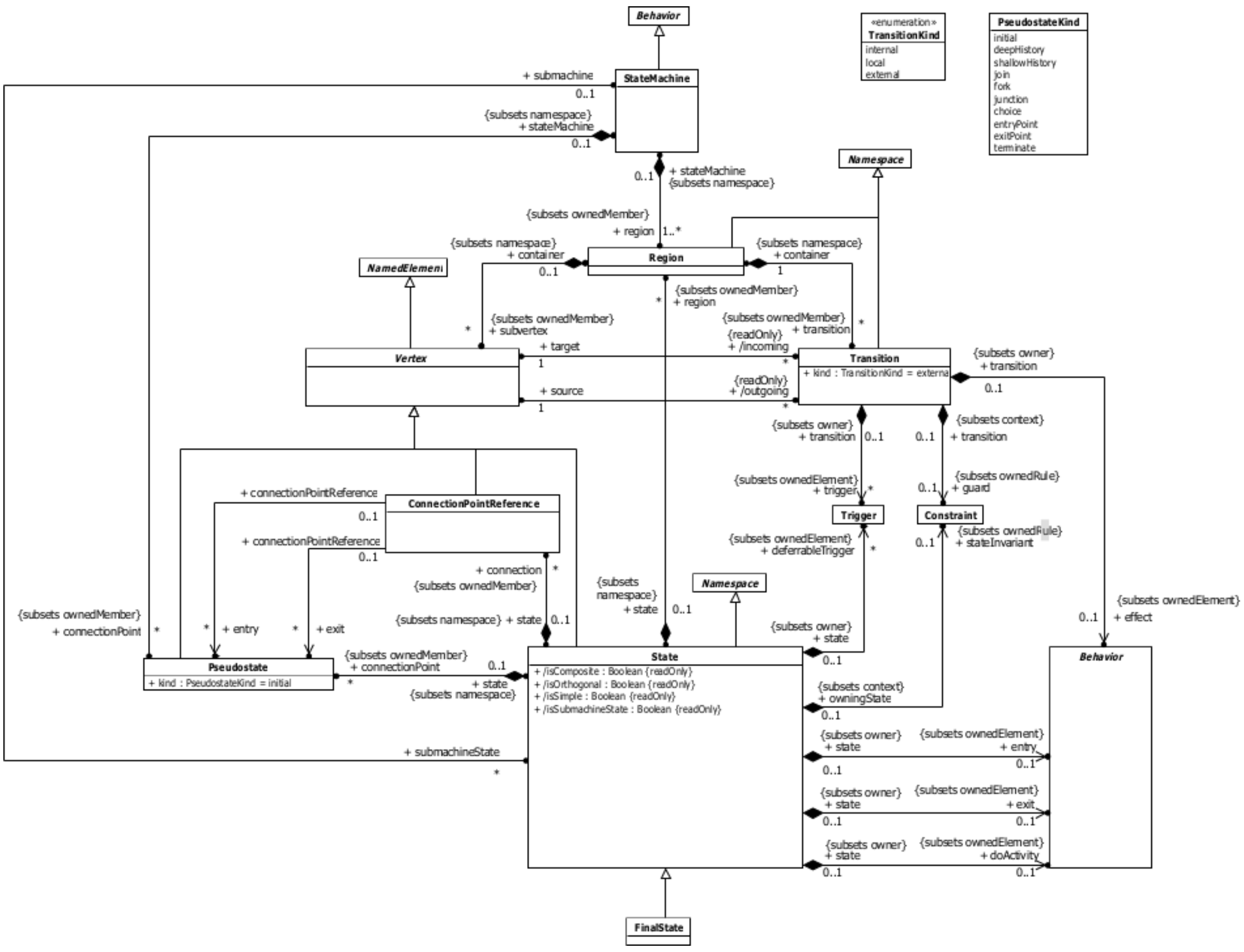
2.1 Our target metamodel: UML state machines

Among many others, Unified Modeling Language (UML) [7] is one of the more accepted standards to represent state machines. UML itself is a family of various languages (*metamodels*) suitable to represent various aspects of large software systems. The UML state machine language can be used to formally describe event-driven systems, i.e. systems that wait for certain events, and upon the occurrence of these events, they change their behaviour and wait for a possibly different set of events. The state machine metamodel of UML is a more general representation of computation than the classical models of finite state machines, since it provides several extensions to the classical model, like embedded state machines, assignable variables, branching states with guards, etc.

Figure 1 depicts the UML state machine language with a metamodel diagram. The root container object is always an instance of the `StateMachine` class. The root object contains a `Region` object, which in turn contains `Vertex` and `Transition` objects, that can be used to denote states and state transitions respectively. `Pseudostate` vertices can be used to denote initial states and choice states, `FinalState` vertices to denote stop states, and `State` vertices to denote ordinary states. The transitions can be assigned events (`Trigger`), guards (`Constraint`) and actions (`Behavior`).

2.2 State machines in the Erlang language

Erlang [12, 14] is a general purpose, functional, dynamically typed, open source programming language, mostly used to develop multithreaded, real time, fault tolerant applications, like telecommunication systems, web servers, or distributed databases. The language provides various abstractions to support these applications. For example, the event handler (`gen_event`), thread monitor (`supervisor`), server (`gen_server`), and state machine (`gen_fsm`) behaviours, provided by the built-in OTP library [17, 14]. Behaviours have similar roles to abstract classes in the object oriented paradigm: to implement a behaviour, we have to implement certain functions, called *callback functions*, specified by the behaviour semantics. The complex, behaviour specific background logic connecting these callbacks together is provided by Erlang. This way, we only have to implement the logic specific to our application, but not the logic specific to the behaviour semantics. For example, the requirements of the `gen_fsm` behaviour are to implement a callback function, named `init`, and any number of transition functions. The function `init` will designate, at a minimum, the initial state of the state machine. The transition functions will designate, at a minimum, the next state the state machine will be in when it receives a specific event, while in a specific state. All the logic necessary to handle multiple threads, messages, events, etc. will be handled by Erlang in accordance to the `gen_fsm` semantics [14].



Behavior
StateMachine
«enumeration» TransitionKind
internal
local
external
PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
Namespace
NamedElement
Region
Vertex
Transition
+ kind : TransitionKind = external
ConnectionPointReference
Trigger
Constraint
Namespace
Pseudostate
+ kind : PseudostateKind = initial
State
+ /isComposite : Boolean {readOnly}
+ /isOrthogonal : Boolean {readOnly}
+ /isSimple : Boolean {readOnly}
+ /isSubmachineState : Boolean {readOnly}
Behavior
FinalState
+ submachine
{subsets namespace} + stateMachine
+ stateMachine {subsets namespace}
{subsets ownedMember} + region
{subsets namespace} + container
{subsets ownedMember} + subvertex
+ target
+ source
{readOnly} + /incoming
{readOnly} + /outgoing
{subsets ownedMember} + transition
{subsets owner} + transition
{subsets context} + transition
{subsets ownedElement} + trigger
{subsets ownedRule} + guard
{subsets ownedElement} + deferrableTrigger
{subsets ownedRule} + stateInvariant
+ connectionPointReference
+ connection {subsets ownedMember}
{subsets namespace} + state
{subsets ownedMember} + connectionPoint
+ entry
+ exit
{subsets ownedMember} + connectionPoint
+ submachineState
{subsets owner} + state
{subsets context} + owningState
{subsets ownedElement} + entry
{subsets ownedElement} + exit
{subsets ownedElement} + doActivity
{subsets ownedElement} + effect

Figure 1: The UML state machine metamodel [7]

In our research, we use the RefactorErl static analysis framework [11, 24] to analyse Erlang source code. The RefactorErl tool first analyses the source code, and then stores the discovered lexical, syntactic and semantic information in a database. This information can be accessed through various user interfaces, and the framework provides several feature to run refactorings on the source code, to perform further analyses – like data flow, and dynamic function call analysis –, to execute various queries, to calculate certain metrics, and many other features. To transform Erlang state machines to UML state machines, we based our definition of the transformation on the data structure RefactorErl uses to represent the lexical, syntactic and semantic information it gathers. This data structure will be described in more detail in Section 3.

3 Methodology

In this section we first describe the main entities and their related notations involved in the transformation of Erlang state machines (i.e. Erlang modules implementing the `gen_fsm` behaviour), and then we give the algorithm that realises this transformation.

3.1 Internal program representation of RefactorErl

The RefactorErl analysis framework stores all information it gathers about Erlang programs via static analysis in a special data structure, called semantic program graph (*SPG*) [11]. In this paper, we show how the *SPG* can be transformed into a state machine, which corresponds to the original state machine described by the original Erlang source code. As the *SPG* is based on a syntax tree and extended with various semantic elements, it represents the lexical, syntactic and semantic structure of one or more Erlang applications. Syntactic and

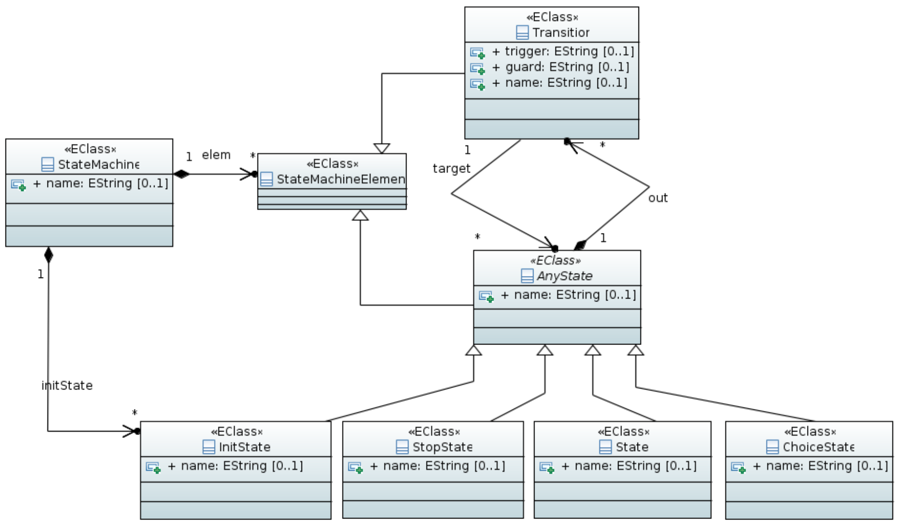


Figure 2: A simpler metamodel for describing abstract state machines

semantic elements of a program are mapped to nodes in this graph, while their relationships are mapped to edges between the corresponding nodes. In the following sections the set of all nodes in a specific *SPG* instance will be denoted as V_{SPG}, while set of all edges will be denoted as E_{SPG}.

A small Erlang program and a segment of its *SPG* can be observed in Figure 3 and Figure 4 in Section 5. Apart from the special node called *root*, all nodes in the *SPG* correspond to lexical, syntactic, or semantic elements in the Erlang source code. The root node is the only node without incoming edges, and serves as the common ancestor for all nodes of the *SPG*. Edges of the *SPG* are ordered.

RefactorErl supplies various tools for discovering and analysing the *SPG*, such as a user friendly query language, and several useful library functions that can be used in more complex programmed queries [26]. RefactorErl also performs special predefined semantic analyses on the *SPG*. One of these is the zeroth and first order dataflow analysis that discovers how data can flow between the syntactical elements of an Erlang program and marks these dataflow relations as edges between the corresponding *SPG* nodes [25, 11]. Another one is dynamic function call analysis that discovers the functions called by dynamic function calls, and represents this relationship with an edge in the *SPG* between the corresponding function node and the node of the dynamic caller expression [15].

3.2 A simpler metamodel for describing abstract state machines

In this section we will describe a simple state machine metamodel, depicted by Figure 2, with which we represent the target state machines of the transformation. We showed in [19] that this metamodel (and its instances) can be mapped onto the UML state machine metamodel (and its instances). This intermediate state machine language explicitly highlights the elements we utilise from UML. Later in this section we will introduce a notation for these elements. For implementation purposes, the intermediate state machine can be omitted altogether, by substituting the UML state machine element descriptions for the corresponding elements in our notation.

The target state machines will basically consist of states (`AnyState`) and transition (`Transition`) elements between those states. There are four kinds of states: ordinary states (`State`), the initial state (`InitState`), stop states (`StopState`), and choice states (`ChoiceState`). Every transition may have exactly one source and one target state. States may have arbitrary number of incoming and outgoing edges, including zero. Transitions may have trigger and guard attributes, depending on whether their source state is an ordinary state or a choice state, respectively. In this paper, triggers and guards will be represented as simple strings constructed from events and guard expressions in the Erlang source code. In order to make the target state machines executable, further research could extend this approach to include a more sophisticated representation for trigger and guard elements. In the following sections the set of all states in a specific state machine instance will be denoted as V_{FSMG}, while the set of all transitions will be denoted as E_{FSMG}.

Let $Init$ be the following set of gen_fsm callback functions:

$$Init \stackrel{\text{def}}{=} \{\texttt{init/1}, \texttt{handle\_event/3}, \texttt{handle\_sync\_event/4}, \texttt{handle\_info/3}, \texttt{code\_change/4}\}$$

As per the specification of the gen_fsm behaviour, the functions in $Init$, when triggered, can put the state machine in any arbitrary state, independently of the actual state. If we only consider the behaviour of an Erlang state machine in terms of states and transitions, but do not consider the memory changes (side effects) accumulated during the execution, then it can be said that these functions effectively restart the state machine. Therefore, it makes sense to model these as initial states, i.e. states from which the state machine execution can be started.

To describe the transformation rules we will use a textual notation to denote a mapping from nodes in the *SPG* and states in the state machine, and also its inverse, a mapping from states to nodes. Since we previously distinguished four types of states, we will use a different function for every type: each of these maps a node to a state with the associated state-type. All these functions can be defined to be invertible.

- $state \in V_{SPG} \to V_{FSMG}$ maps nodes to ordinary states. The nodes mapped to ordinary states will precisely be the semantic nodes representing transition functions, i.e. the user defined callback functions of the gen_-fsm behaviour, bearing the name of a state. We chose these nodes, since they have a 1:1 correspondence with the states of the state machine implemented by the analysed Erlang module.

- $choice \in V_{SPG} \to V_{FSMG}$ maps nodes to choice states. The nodes mapped to choice states are either representing functions with multiple clauses, or they are representing branching expressions. For brevity, in this paper we only touch upon the latter – and simpler – case: branching expressions. We presented handling of the former case in [18].

- $stop \in V_{SPG} \to V_{FSMG}$ maps nodes to stop states. Following the specification of the gen_fsm behaviour, the nodes mapped to stop states will precisely be those representing tuple expressions, that are return points of a transition function, and have the `stop` atom as their first element.

- $init \in V_{SPG} \to V_{FSMG}$ maps nodes to initial states. Following the specification of the gen_fsm behaviour and our previous remark, the nodes mapped to initial states will precisely be those representing the functions in $Init$.

After applying the transformation, every state in the target state machine will correspond to a node in the *SPG*, and every transition in the target state machine will correspond to an edge sequence in the *SPG*. In Section 4 where we describe the transformation rules we will denote this state-node correspondence relation with the function $node : V_{FSMG} \to V_{SPG}$. The *node* function is defined as the inverse of the node-state mapping described earlier. Thus, if we regard the earlier functions as relations, i.e. sets of ordered pairs, then

$$node \stackrel{\text{def}}{=} (state \cup choice \cup stop \cup init)^{-1}$$

Since the range of these functions are pairwise disjoint, and all four functions were invertible, *node* always exists.

In the definition of the transformation rules, we also use a textual notation to denote the trigger and guard labels of the transitions. As mentioned earlier, we represent these as simple human or machine-readable strings. The set of all strings will be denoted by $\mathbb{S}$.

- $trigger \in V_{SPG} \to \mathbb{S}$ maps nodes representing the first parameter pattern of a transition function to a string. For example, the string may consist entirely of the Erlang term denoting this parameter.

- $guard_{branch}$, $guard_{if}$, $guard_{try}$, $guard_{catch}$,$guard_{after}$ and $guard_{clause} \in V^n_{SPG} \to \mathbb{S}$ functions map nodes representing the pattern and guard expressions of the clauses of the various branching expressions can be found in the Erlang programming language. Since it may be necessary to use more patterns and guard expression to uniquely identify a state machine guard, we assume these functions can handle more parameters. The exact value of n depends only on the *guard* function it describes.

3.3 An algorithm for transforming program graphs to state machines

In this section we present an algorithm, that – with the help of a predefined set of transformation rules – transforms the semantic program graph of any Erlang state machine adhering to the specification of the `gen_fsm` behaviour, to a state machine model described in UML or the simple state machine language we introduced in Section 3. Since most of the application specific logic (the logic related to the RefactorErl semantic program graph) of the transformation are encoded in the transformation rules, the algorithm itself is relatively simple. Basically it is an extended depth first search, that selects the neighbouring nodes to discover, based on predefined rules.

This approach has several advantages. To extend the transformation for currently unhandled cases, we do not have to modify the procedural algorithm, we only have to add more rules to the transformation sets. For example, UML offers several state machine features (e.g. embedded state machines, variables, effects) that could be utilised by the transformation, after the rule set were to be appropriately extended to handle these cases. Also, the rule sets encapsulate the RefactorErl specific logic, which means we could use the same algorithm with other static analysis frameworks too. We just have to swap the RefactorErl specific rule sets to the rule sets specific to the other static analysis framework. It is worth noting though that devising a rule set, in general, is not a trivial task. Finally, this approach opens up the possibility to execute rules in parallel, to achieve better runtime performance.

Informally, our transformation algorithm, together with the rules presented in Section 4, will perform the following tasks:

1. Starting with the transition functions in $Init$, the algorithm analyses the return point of each transition function it visits.

2. If during the analysis, it discovers an branching expression or a function with multiple clauses, denoted as b, the corresponding $choice(b)$ choice state will be added to the state machine. Then the algorithm continues by analysing the return points of each clause of the b branching expression or function.

3. If during the analysis a t tuple is discovered, it will further analysed t to decide whether it indicates a stop state, or an ordinary state. In the former case, the corresponding $stop(t)$ stop state will be added to the state machine, and the algorithm backtracks. In the latter case, it will analyse the appropriate element of t (see the `gen_fsm` specification [14]), to find out the f name of the target state of the currently analysed transition function. If found, the algorithm will continue to analyse the transition function with name f.

4. For any other node type, the algorithm will proceed in way specific to this node type. In most cases it will utilise the dataflow analysis provided by the RefactorErl tool [25, 11]. Since the dataflow analysis may abstract away information necessary to analyse `gen_fsm` modules, we require node type specific analysis in some cases.

5. When the algorithm adds an s state to the state machine, it will also have to add an appropriate transition between s and the old state o, which corresponds to the transition function named o, that was analysed as s was discovered.

More specifically, our algorithm will perform these generic tasks in three separate stages: an analysis stage, a transformation stage, and a synthesis stage. In the *analysis stage*, our goal is to discover precisely those nodes and edges in the SPG that will be mapped to state machine elements in later stages. The result is a filtered SPG, called the analysed SPG, consisting only these nodes and edges. The edges are relabelled with semantic information about their role in the future state machine. The *transformation stage* eliminates nodes and edges from this analysed SPG, so as to obtain a reduced SPG that can be mapped to a state machine instance with relative ease. The *synthesis stage* will map the reduced SPG to a state machine instance, producing the final result of the transformation.

The general algorithm with the three stages is denoted on Figure 1. All three stages will start a depth first algorithm from the function nodes in the $Init$ set or, in the case of the synthesis stage, $init(Init)$, the set of the initial states corresponding to the nodes in $Init$. For certain edges we will not continue the analysis, i.e. will not extend the target nodes of these edges. The types of these edges are listed in the $Exclude_1$ set.

Later in this paper we will define separate rule sets for each of these stages. Rule sets R_1 and R_2 – corresponding to the analysis and synthesis stage respectively – are side-effect free. By matching the left hand side of the rules to a graph, their right hand side can be used to construct another graph. Rules in rule set R_3 –

Algorithm 1 $DiscoverFSMG(SPG, R_1, R_2, R_3)$

```
1: $Init \leftarrow \{init/1, handle\_event/3, handle\_info/3,$
       $handle\_sync\_event/4, code\_change/4\}$
2: $Exclude_1 \leftarrow \{\overset{\textbf{trigger}}{\rightsquigarrow}, \overset{\textbf{cond*}}{\rightsquigarrow}, \overset{\textbf{fsmguard}}{\rightsquigarrow}\}$
3: $RelationGraph \leftarrow discover(Init, R_1, Exclude_1, SPG)$
4: $TRelationGraph \leftarrow transform(Init, R_2, RelationGraph)$
5: $FSMG \leftarrow discover(init(Init), R_3, \varnothing, TRelationGraph)$
6: **return** $FSMG$
```

corresponding to the transformation stage – are not side-effect free. By matching their left hand side in a graph, their right hand side can be used to modify this same graph.

The analysis stage and the synthesis stage will make use of the same backtracking algorithm, that tries to pattern match every rule to an environment of the current node, and determines the next neighboring nodes to visit using the matching rules. The transformation stage makes use of a slightly modified backtracking algorithm. This one will try to apply a rule to a node as many times as possible, before moving on the next node. After it visited every node, it will proceed to repeat this procedure with the next rule. The stage ends when the last rule was applied as many times as possible to every node in the graph.

4 Specification of the transformation rules

As mentioned earlier, the transformation is realised by two backtracking algorithms, that are performing pattern matching on their respective input graphs with the left hand side (LHS) of certain set of rules, and then apply the right hand side (RHS) of the matching rules to construct an output graph, or – in case of the transformation stage – to modify the input graph. This section describes the algorithms referenced by Figure 1 and outlines the rule sets utilised by the analysis, transformation and synthesis stages. The keep the discussion concise, we only selected a few rules to present here, and these can only be used to transform small, simple state machines, like the one demonstrated in Section 5. We presented more elaborate rule sets capable of transforming large, complex state machines, in the appendices of [18]. Our reference implementation is also based on these larger rule sets, and, as described in Section 6, it was successfully tested on state machines, selected from the sources of large, open source, widely used Erlang applications.

First, we introduce a notation that will be used to denote these rules. The LHS of the rules will consist of edge patterns and logical expressions, featuring relations between nodes of the input graph. In these rules, x always denotes the node that the pattern matching must start on, while other variable names are arbitrary. For example, if a rule has the pattern $x \xrightarrow{\textbf{def}} y \in SPG_E$ as its LHS, then the rule matches on some x_0 node of the input graph if and only if there is an edge, between the nodes x_0 and some arbitrary y_0, with the label `def`. We use a shorthand notation for connecting relations: we write $x \to y \to z \subset SPG_E$ instead of $x \to y \in SPG_E \ \wedge \ y \to z \in SPG_E$. Edges in the semantic program graph are ordered: we denote restrictions on the sequence number of an edge by denoting the number after the edge label. E.g. a rule with $x \xrightarrow{\textbf{clause/2}} y \in SPG_E$ in its LHS matches on x_0 only if there is an edge with the label *clause* and the sequence number 2 between x_0 and some other node. Unnumbered edges in the patterns may match edges of the input graph with arbitrary sequence numbers. Nodes in the semantic program graph can also possess various properties: we denote restrictions on a property of a node with an equality expression between braces after the node. E.g. $x[type = tuple] \xrightarrow{\textbf{elem/1}} y \in SPG_E$ matches on some node x_0 only if the pattern $x \xrightarrow{\textbf{elem/1}} y \in SPG_E$ matches and the *type* property of x_0 is of the value *tuple*. Pattern nodes without property restrictions may match nodes with arbitrary values on their properties.

4.1 Analysis stage

In the analysis stage we traverse part of the semantic program graph of RefactorErl to identify all the nodes used by the following stages, and to attach transformation-specific semantic information to the edges by relabelling them. This traversal is done by the backtracking algorithm described in Figure 2, utilising the rules collected in Table 1.

The backtracking algorithm in Figure 2 starts by visiting and expanding the nodes in the *Init* set. To expand a node x_0, it iterates over all the rules in Table 1, and tries to pattern match these rules on x_0. For any matching rules, it adds the RHS of the matching rule to the node in the output graph, corresponding to x_0, and puts the

Algorithm 2 *discover*$(Init, Rules, Exclude, G)$

```
1: $V \leftarrow Init$
2: $E \leftarrow \varnothing$
3: $Visited \leftarrow \varnothing$
4: $S \leftarrow stack(Init)$
5: while $S \neq \varnothing$ do
6:     $v \leftarrow S.pop()$
7:     if $v \notin Visited$ then
8:         $Visited.add(v)$
9:         $NEdges \leftarrow \varnothing$
10:        for $r \in Rules$ do
11:            $NEdges.add(r.match(G, v))$
12:        end for
13:        for $e \in NEdges$ do
14:            $E.add(e)$
15:            $V.add(endpoint(e))$
16:            if $e.edgetype() \notin Exclude$ then
17:                $S.add(endpoint(e))$
18:            end if
19:        end for
20:    end if
21: end while
22: return $(V, E)$
```

endpoint, denoted in the rules by y, in the stack to expand later. If the type of the edge in the RHS of the matching rule is featured in *Exclude*, then the y endpoint will not be expanded. As both the input graph and the rule sets are finite, the backtracking algorithm is guaranteed to terminate, and operates in polynomial time.

Apart from identifying the neighbouring nodes to be visited by the backtracking algorithm, the rules in Table 1 describe how certain edge sequences encountered in the semantic program graph will be labelled in the analysed semantic program graph. These labels indicate the semantic roles of the endpoints of their respective edges, and these roles will determine how each node will be treated in the subsequent stages. Labels $\overset{\mathbf{so}}{\rightsquigarrow}$ and $\overset{\mathbf{s}}{\rightsquigarrow}$ denote a transition function node at their source, $\overset{\mathbf{nameof}}{\rightsquigarrow}$ connects an atom with a function with the same name as the value of that atom, and the edges with $\overset{\mathbf{trigger}}{\rightsquigarrow}$ and $\overset{\mathbf{cond}}{\rightsquigarrow}$ labels point to nodes that will be used to construct trigger and guard labels in the final state machine. The $\overset{\mathbf{fsmo}}{\rightsquigarrow}$ label is a special label that needs to substituted to other labels as specified by Table 2. This substitution only serves to make our definition more compact: it can be eliminated in design time by adding new rules by combining the conditions of the rules in Table 1 and Table 2. Because of this, the substitution step does not need to appear in the algorithm either. As for the labels appearing in Table 2, $\overset{\mathbf{c}}{\rightsquigarrow}$ denotes a branching expression at its source, $\overset{\mathbf{ao}}{\rightsquigarrow}$ and $\overset{\mathbf{a}}{\rightsquigarrow}$ denote ordinary functions, $\overset{\mathbf{e}}{\rightsquigarrow}$ points to a tuple that will be mapped to a stop state, and $\overset{\mathbf{t}}{\rightsquigarrow}$ points to a tuple which contains the name of a state in the state machine. The label $\overset{\mathbf{0}}{\rightsquigarrow}$ does not convey any specific semantic meaning, it is only used to specify the next nodes to be visited by the backtracking algorithm.

In later stages, transition functions will be mapped to states, and each function clause will correspond to a state transition leading out of that state, with a trigger label constructed from the first parameter (the event) of the clause. Since the `gen_fsm` specification allows for multiple clauses to have the same event, this mapping could results in non-deterministic state machines. To avoid this, we introduce a choice state for every group of clauses with the same event. This way there will be only one transition for each event between the original state and the choice state, and we will use guard labels on the transitions leading out from the choice states to distinguish between clauses in the same group. Ordinary functions with multiple clauses, similar to branching expressions, will be mapped to choice states, while those with single clauses do not have to be denoted in the resulting state machine.

In Table 2, the predicate $multiclause(x)$ is true iff the node x is a node in the semantic program graph, representing a function with multiple clauses. Let $\sim$ denote a relation between function clauses, and let $c_1 \sim c_2$ be true for c_1, c_2 of a function iff their first parameter patterns are identical. As $\sim$ is an equivalence relation, it partitions the clauses of a function into equivalence classes. In Table 1 the predicate $multiclausegroup(x)$ is true iff the node x represents a transition function with multiple clauses and there is at least one class of $\sim$ with at least two elements, i.e. the function has at least two clauses with identical first parameter patterns. If x is an atom, $find(x)$ denotes the set of all functions with the same name as the value of x.

Table 1: Rules for the analysis stage

	LHS	RHS
§1	$x \xrightarrow{\mathbf{def}} y \in SPG_E \quad \wedge \quad multiclausegroup(y)$	$x \overset{\mathbf{s}}{\rightsquigarrow} y$
§2	$x \xrightarrow{\mathbf{def}} y \in SPG_E \quad \wedge \quad \neg\ multiclausegroup(y)$	$x \overset{\mathbf{so}}{\rightsquigarrow} y$
§3	$x \xrightarrow{\mathbf{def}} y \xrightarrow{\mathbf{fclause/i}} cl \xrightarrow{\mathbf{pattern/1}} patt \subset SPG_E$	$x \overset{\mathbf{trigger/i}}{\rightsquigarrow} patt$
§4	$x_{[type=atom]} \in SPG_V \quad \wedge \quad y \in find(x)$	$x \overset{\mathbf{nameof}}{\rightsquigarrow} y$
§5	$x_{[type=func\|fun\_expr]} \xrightarrow{\mathbf{fclause/i}} clause \xrightarrow{\mathbf{cret}} y \subset SPG_E$	$x \overset{\mathbf{fsm_0/i}}{\rightsquigarrow} y$
§6	$type(x) \in \{if\_expr\|case\_expr\|try\_expr\|receive\_expr\} \quad \wedge$ $x \xrightarrow{\mathbf{(exprcl\|catchcl\|aftercl)/i}} cl \xrightarrow{\mathbf{cret}} y \in SPG_E$	
§7	$type(x) \in \{case\_expr\|try\_expr\|receive\_expr\} \quad \wedge$ $x \xrightarrow{\mathbf{exprcl/i}} cl \xrightarrow{\mathbf{pattern/1}} patt \subset SPG_E$	$x \overset{\mathbf{cond_{branch}/i}}{\rightsquigarrow} patt$
§8	$x_{[type=case\_expr\|try\_expr]} \xrightarrow{\mathbf{exprcl}} cl_1 \in SPG_E \ \wedge$ $x \xrightarrow{\mathbf{headcl/i}} cl_2 \xrightarrow{\mathbf{cret}} y \subset SPG_E$	$x \overset{\mathbf{cond_{head}/i}}{\rightsquigarrow} y$
§9	$x_{[type=tuple]} \xrightarrow{\mathbf{elem/1}} z \ \in SPG_E \quad \wedge$ $z \overset{\mathbf{flow}}{\leftsquigarrow} atom_{[value='next\_state'\|'ok']} \ \in SPG_E \quad \wedge$ $x \xrightarrow{\mathbf{elem/2}} y \ \in SPG_E$	$x \overset{\mathbf{fsm_0}}{\rightsquigarrow} y$
§10	$x_{[type=tuple]} \xrightarrow{\mathbf{elem/1}} z \ \in SPG_E \quad \wedge$ $z \overset{\mathbf{flow}}{\leftsquigarrow} atom_{[value='reply']} \ \in SPG_E \quad \wedge$ $x \xrightarrow{\mathbf{elem/3}} y \ \in SPG_E$	

Table 2: Substitution rules for $x \overset{\mathbf{fsm_0}}{\rightsquigarrow} y$

Condition (x, y)	$\overset{\mathbf{fsm_0}}{\rightsquigarrow}$
$type(y) = if\_expr \mid case\_expr \mid try\_expr \mid receive\_expr$	$\overset{\mathbf{c}}{\rightsquigarrow}$
$type(y) = func \mid fun\_expr \ \wedge \ \neg \ multiclause(y)$	$\overset{\mathbf{a_0}}{\rightsquigarrow}$
$type(y) = func \mid fun\_expr \ \wedge \ multiclause(y)$	$\overset{\mathbf{a}}{\rightsquigarrow}$
$type(y) = atom$	$\overset{\mathbf{e}}{\rightsquigarrow}$
$type(y) = tuple \ \wedge$ $y \xrightarrow{\mathbf{elem/1}} z \ \in SPG_E \ \wedge$ $z \overset{\mathbf{flow}}{\leftsquigarrow} atom_{[value='stop']} \ \in SPG_E$	$\overset{\mathbf{e}}{\rightsquigarrow}$
$type(y) = tuple \ \wedge$ $y \xrightarrow{\mathbf{elem/1}} z \ \in SPG_E \ \wedge$ $z \overset{\mathbf{flow}}{\leftsquigarrow} atom_{[value='next\_state' \mid 'reply' \mid 'ok']} \ \in SPG_E$	$\overset{\mathbf{t}}{\rightsquigarrow}$
otherwise	$\overset{\mathbf{0}}{\rightsquigarrow}$

4.2 Transformation stage

In the transformation state we reduce the analysed semantic program graph to acquire a graph that can be easily mapped to a state machine. For this stage we slightly modified the backtracking algorithm. This algorithm iterates over all the rules in a predetermined order, and in each iteration it visits the nodes of the analysed semantic program graph to apply the actual rule as many times as possible. Since the matching rules specified in Table 3 always eliminate some of the edges from their LHS, the algorithm is guaranteed to terminate. Since eliminated nodes will not be visited again, this algorithm also operates in polynomial time.

Algorithm 3 $transform(Init, Rules, G)$

```
Require: $Rules$ is ordered according to the transformation rule set
 1: for $r \in Rules$ do
 2:     $Visited \leftarrow Init$
 3:     $S \leftarrow stack(Init)$
 4:     while $S \neq \varnothing$ do
 5:         $v \leftarrow S.pop()$
 6:         if $v \notin Visited$ then
 7:             $Visited.add(v)$
 8:             do
 9:                 $success \leftarrow r.execute(G, v)$
10:             while $success$
11:             $S.add(G.children(v))$
12:         end if
13:     end while
14: end for
```

We used a procedural approach to describe the rules for this stage: the RHS of these rules contains simple statements that are inserting or removing edges to or from the input graph. If the LHS of a rule matches, the modifications specified in the RHS are performed on the input graph. As mentioned previously, we only included in this paper those rules that are necessary to demonstrate the transformation of a small example state machine, and a more elaborate extended rule set capable of transforming any Erlang state machines is presented in the appendices of [18]. While the rules presented here could have been expressed with a declarative approach, it would have been more difficult to express declaratively those in the extended transformation rule set. The rule §11 contracts $\overset{\mathbf{nameof}}{\rightsquigarrow}$ edges, while §12 contracts $\overset{\mathbf{s_0}}{\rightsquigarrow}$ edges. The rule §13 contracts $\overset{\mathbf{t}}{\rightsquigarrow}$ and $\overset{\mathbf{0}}{\rightsquigarrow}$ edges in a way that the new edge inherits the edge sequence number of the contracted edge. With the exception of the placeholder $\overset{\mathbf{0}}{\rightsquigarrow}$, all these edges can be used to eliminate unnecessary or unwanted segments from the analysed program graph. For example, branches that would be mapped to a choice state with only one outgoing transitions, or for example dead ends, i.e. paths that would be mapped to transitions without a target state. We also described these eliminations in [18].

Table 3: Rules for the transformation stage

	LHS	RHS
§11	$\exists i_1, ..., i_n\ (n \geq 1):$ $x \overset{\mathbf{e/i_1}}{\rightsquigarrow} y_{i_1} \overset{\mathbf{nameof}}{\rightsquigarrow} z\ \wedge$ $\vdots$ $x \overset{\mathbf{e/i_n}}{\rightsquigarrow} y_{i_n} \overset{\mathbf{nameof}}{\rightsquigarrow} z\ \wedge$ $\nexists j : j \notin \{i_1, ..., i_n\}\ \wedge\ x \overset{\mathbf{e/j}}{\rightsquigarrow} y_j \overset{\mathbf{nameof}}{\rightsquigarrow} z$	$\forall k \in \{i_1, ..., i_n\}\ (insert(x \overset{\mathbf{e/k}}{\rightsquigarrow} z))$; $\forall k \in \{i_1, ..., i_n\}$ ($remove(x \overset{\mathbf{e/k}}{\rightsquigarrow} y_k \overset{\mathbf{nameof}}{\rightsquigarrow} z)$)
§12	$\exists i_1, ..., i_n\ (n \geq 1):$ $S \in \{t, 0, a, c, e, s\}\ \wedge$ $x \overset{\mathbf{s0}}{\rightsquigarrow} y \overset{\mathbf{S/i_1}}{\rightsquigarrow} z_{i_1}\ \wedge$ $\vdots$ $x \overset{\mathbf{s0}}{\rightsquigarrow} y \overset{\mathbf{S/i_n}}{\rightsquigarrow} z_{i_n}\ \wedge$ $\nexists j : j \notin \{i_1, ..., i_n\}\ \wedge\ x \overset{\mathbf{s0}}{\rightsquigarrow} y \overset{\mathbf{S/j}}{\rightsquigarrow} z_j$	$\forall k \in \{i_1, ..., i_n\}$ ($insert(x \overset{\mathbf{S/k}}{\rightsquigarrow} z_k)$; $remove(y \overset{\mathbf{S/k}}{\rightsquigarrow} z_k)$) ; $remove(x \overset{\mathbf{s0}}{\rightsquigarrow} y)$
§13	$R \in \{t, 0\}\ \wedge$ $S \in \{a, c, e, s\}\ \wedge$ $x \overset{\mathbf{R/i}}{\rightsquigarrow} y \overset{\mathbf{S}}{\rightsquigarrow} z$	$insert(x \overset{\mathbf{S/i}}{\rightsquigarrow} z)$; $remove(x \overset{\mathbf{R/i}}{\rightsquigarrow} y \overset{\mathbf{S}}{\rightsquigarrow} z)$

4.3 Synthesis stage

In the last stage we map the reduced analysed semantic program graph to a state machine. The purpose of the previous transformation stage was to make the rules in the synthesis stage simpler. As shown in Figure 1, this stage reuses the backtracking algorithm in Figure 2 with the rules specified in Table 4. This time the algorithm expands states instead of program graph nodes. In the analysis stage, the LHS of the rules were pattern matched to the input semantic program graph, and the news edges identified by the RHS were added to the output analysed program graph. In the synthesis state, the input graph is the reduced analysed program graph and output graph is a state machine, thus the LHS pattern matching is performed on the reduced analysed graph, while the new edges in the RHS are transitions, which are added to the state machine. Similar to the $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ label of the first stage, the $\xrightarrow{\mathbf{lab}}$ labels in the RHS of the rules in Table 4 can be substituted according to Table 5. Again, this label substitution is only required to keep the description concise, and it can be accomplished during design time by adding new rules that combine the corresponding conditions and rules of the two tables.

Table 4: Rules for the synthesis stage

	LHS	RHS
§14	$node(v) \overset{\mathbf{e}}{\rightsquigarrow} y_{[type \neq tuple]}$	$v \xrightarrow{\mathbf{lab}} state(y)$
§15	$node(v) \overset{\mathbf{e}}{\rightsquigarrow} y_{[type = tuple]}$	$v \xrightarrow{\mathbf{lab}} stop(y)$
§16	$node(v) \overset{\mathbf{c}}{\rightsquigarrow} branch$	$v \xrightarrow{\mathbf{lab}} choice(branch)$

Table 5: Substitution rules for $v \xrightarrow{\textbf{lab}} u$

Condition (v, u)	$\xrightarrow{\textbf{lab}}$
$node(v) \overset{\textbf{fsm}_\textbf{0}\textbf{/i}}{\rightsquigarrow} node(u) \quad \wedge$ $node(v) \overset{\textbf{trigger/i}}{\rightsquigarrow} patt$	$trigger(patt)$
$node(v) \overset{\textbf{fsm}_\textbf{0}\textbf{/i}}{\rightsquigarrow} node(u) \ \wedge$ $node(v)_{[case\_expr \mid try\_expr]} \overset{\textbf{cond}_\textbf{branch}\textbf{/i}}{\rightsquigarrow} patt1 \ \wedge$ $node(v) \overset{\textbf{cond}_\textbf{head}}{\rightsquigarrow} hd$	$guard_{branch}(hd, patt1)$

5 Demonstration of the state machine transformation

In this section we demonstrate the transformation method previously introduced, by presenting the intermediate results of the transformation of a small Erlang state machine. Although we intentionally selected a simple state machine for this example, an implementation of the transformation was also successfully tested on large state machines selected from open source Erlang applications. The code of the example state machine shown in Figure 3 describes the language of identifiers: words starting with letters and proceeding with letters, numbers or underscores. We omitted from the source code some of the callbacks required by the `gen_fsm` specification, and also those functions that can be used to interact with the state machines to initialise the state machine, and to send events to it. In this example events could be letters read from some input word. The state machine accepts identifiers ending with the line ending character (`$\n`), and rejects every other words. The initial state of this state machine is `pos1`, where it transitions to a rejecting state upon receiving a non-letter character, and transitions to `posOther` state upon receiving a letter. In `posOther`, it transitions to an accepting state upon receiving a line ending character, stays in `posOther` upon receiving alphanumeric characters or underscores, and transitions to a rejecting state upon receiving any other event.

```
-module(id_validator).
-behaviour(gen_fsm).

%% The state machine that accepts identifiers, described
%%   by the regular expression   ^[A-Za-z][A-Za-z0-9_]*$

[...]

init(_) ->
  {ok, pos1, []}.

pos1($\n, _, _) -> {stop, normal, {[], reject}, []};
pos1(X, _, _) ->
  case alpha(X) of
    true -> {reply, {[X], step}, posOther, []};
    false -> {stop, normal, {[X], reject}, []}
  end.

posOther($\n, _, _) -> {stop, normal, {[], accept}, []};
posOther(X, _, _) ->
  case (alphanumeric(X) or X == $_) of
    true -> {reply, {[X], step}, posOther, []};
    false -> {stop, normal, {[X], reject}, []}
  end.
[...]
```

Figure 3: The state machine that recognizes the language of identifiers

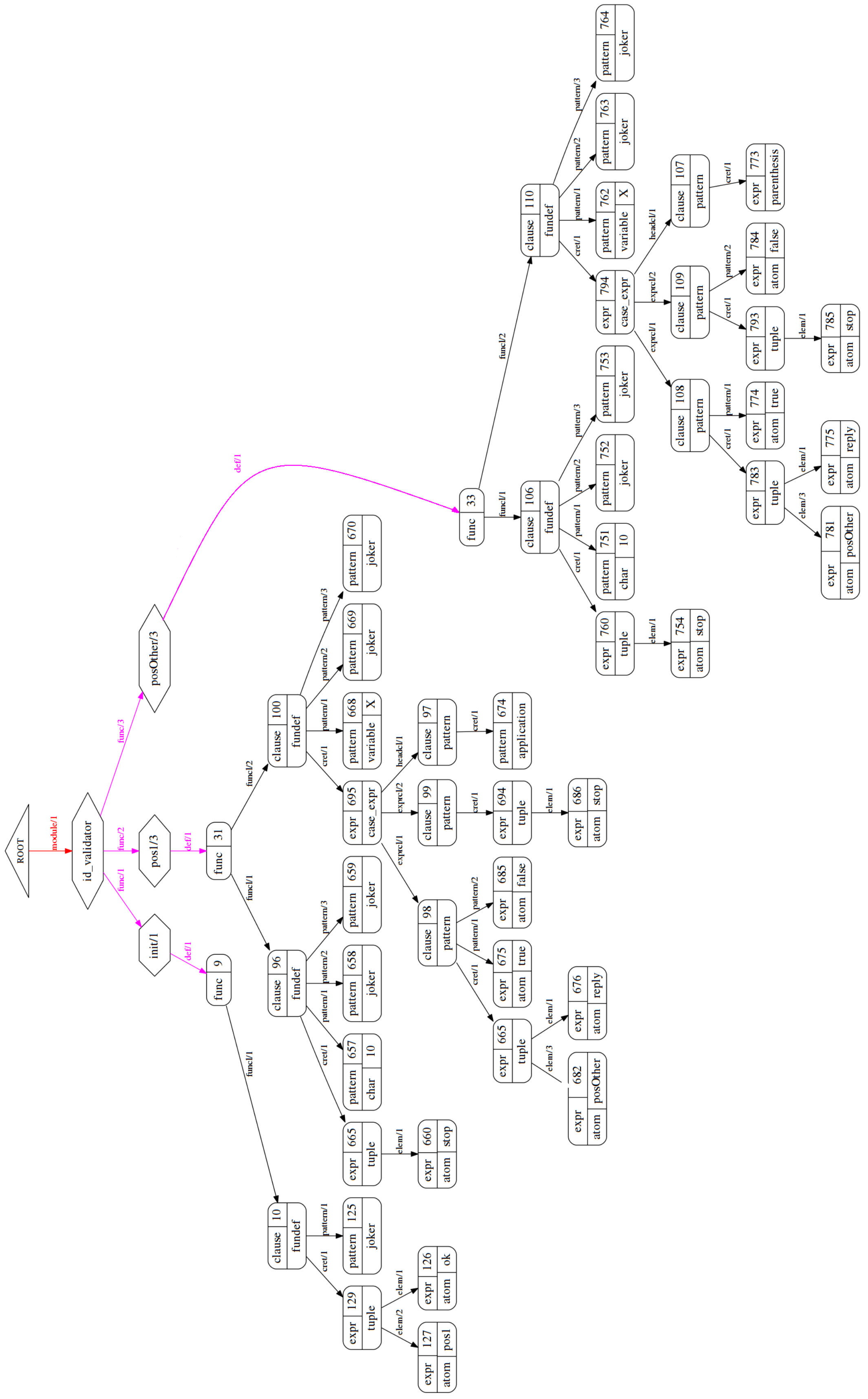


Figure 4: A segment of the RefactorErl semantic program graph

Figure 4 depicts parts of the semantic program graph created by the RefactorErl framework by analysing the program in Figure 3. While the full program graph is a lot larger, the graph shown on this figure is the precisely the subgraph the backtracking algorithm of the first state will traverse and analyse. On top of this graph is the root node, below the root is the semantic node representing the `id_validator` module, and below the module node are the semantic nodes representing the functions in this module. Each function has a form node, and under the form nodes are the nodes corresponding to the clauses of the function. Finally, the subtrees determined by each clause represent the expressions in the clause bodies. The node with the value "10" represents the line ending character.

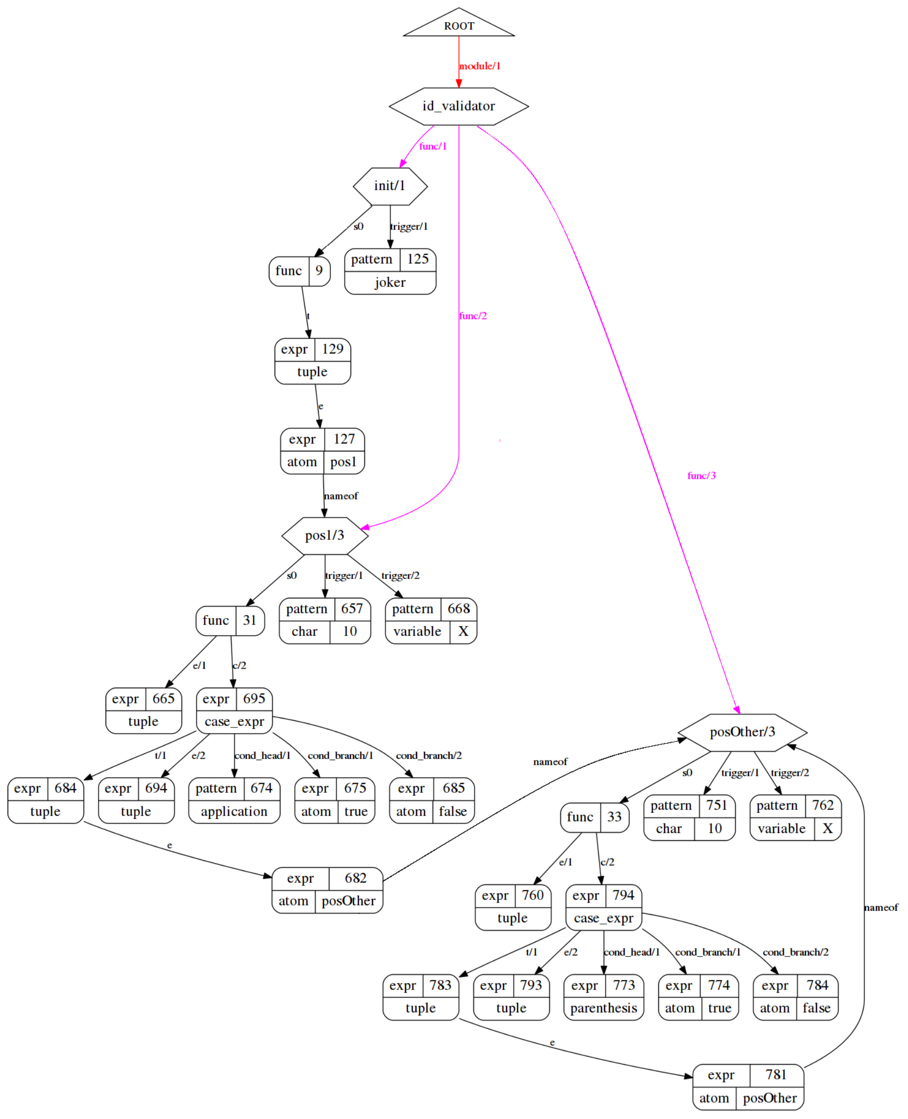


Figure 5: Analysed semantic program graph, resulting from the analysis stage

5.1 Demonstration of the analysis stage

The first stage of the transformation applies the algorithm in Figure 2 to the semantic program graph. The result of this analysis stage is shown in Figure 5. This stage identifies all the nodes used by the following stages and attaches transformation-specific semantic information to the edges by relabelling them.

The first node to analyse is the `init/1` function node, since `init/1` is element of the *Init* set. This function has only one clause, thus the first rules that match are §2 and §3. At this point the resulting graph consists of the edges $\overset{\mathtt{s_0}}{\rightsquigarrow}$ and $\overset{\mathtt{trigger/1}}{\rightsquigarrow}$, and their nodes. Since $\overset{\mathtt{trigger/1}}{\rightsquigarrow}$ is a member of $Exclude_1$, we will not expand the target node of this edge. The next node to expand, as specified by §2, is the form node. The rule §5 matches, thus we add an $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ edge to the result graph. We may choose to do the substitution of $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ right now, based on Table 2, swapping it to $\overset{\mathbf{t}}{\rightsquigarrow}$. The left hand side of §9 matches the tuple node, thus after substituting $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$, we add an $\overset{\mathbf{e}}{\rightsquigarrow}$ edge to the result. Finally §4 matches on the `atom` node, thus we add to results a $\overset{\mathbf{nameof}}{\rightsquigarrow}$ edge, pointing to the semantic node of the `pos1/3` function. At this point the analysis of the `init/1` function concludes, since we found the name of the first state, and the next function to analyse.

On the `pos1/3` function node we can match §2 and §3 again, since all the clauses of `pos1/3` have different events. Now, we can match §5 on two branches, one for each clause. The first tuple represents a stop state, thus we have to substitute $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ by $\overset{\mathbf{e}}{\rightsquigarrow}$. There are no rules matching on this branch, thus the algorithm backtracks to the other branch. In the other branch $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ is substituted to $\overset{\mathbf{c}}{\rightsquigarrow}$. We can match §6, §7, and §8 on the `case_expr` node and its clauses. Again, $\overset{\mathbf{cond_{head}}}{\rightsquigarrow}$, and $\overset{\mathbf{cond_{branch}}}{\rightsquigarrow}$ are in $Exlcude_1$, thus their target nodes – used in later stages to construct guards for the state machine – will not be expanded. In the left branch that matched §6, we substitute $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ by $\overset{\mathbf{t}}{\rightsquigarrow}$, and then continue the analysis of this branch by matching §10 on the tuple, and swap $\overset{\mathbf{fsm_0}}{\rightsquigarrow}$ with $\overset{\mathbf{e}}{\rightsquigarrow}$. Finally, we find the `posOther/3` function node by matching §4. The analysis of `posOther/3` is almost identical, except that the last matching of §4 will result in a $\overset{\mathbf{nameof}}{\rightsquigarrow}$ edge pointing back to the `posOther/3` itself. With this the analysis stage concludes.

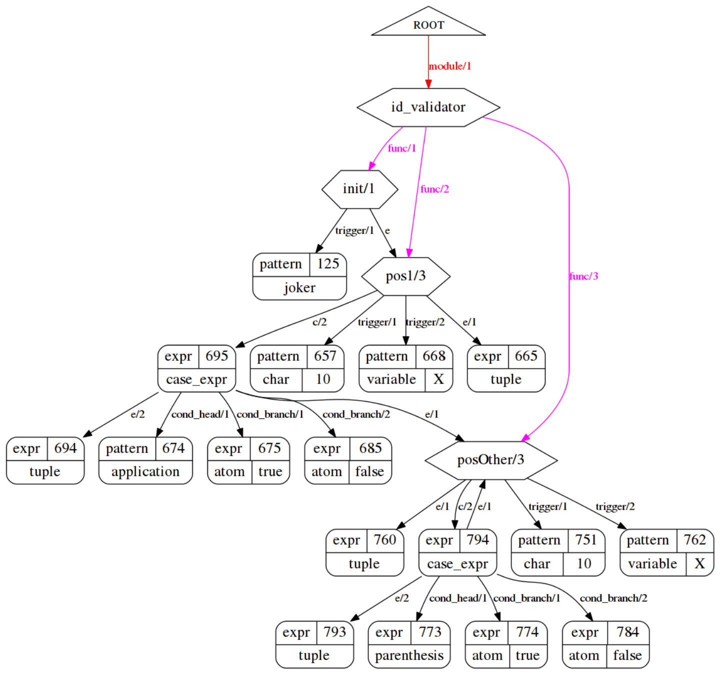


Figure 6: Reduced analysed semantic program graph, resulting from the transformation stage

5.2 Demonstration of the transformation stage

In the transformation stage we start with the analysed program graph, resulting from the analysis stage, shown in Figure 5, and reduce it to graph, that can be mapped to a state machine model relatively easily. The result of the transformation stage is depicted by Figure 6.

Since in this example the state machine we transform is small, we only have to use a few reductions. To reduce the analysed semantic program graph of larger state machines may require more kinds of reduction rules, which are described in the appendix of [18]. The first step is to contract every $\overset{\mathbf{e}}{\rightsquigarrow}$, $\overset{\mathbf{nameof}}{\rightsquigarrow}$ edge pairs into an $\overset{\mathbf{e}}{\rightsquigarrow}$ edge by applying §11. Next we contract the $\overset{\mathbf{s_0}}{\rightsquigarrow}$ edges by applying §12. While the $\overset{\mathbf{s}}{\rightsquigarrow}$ edges highlight transition functions with multiple clauses, each having the same event parameter, this small program did not have these kinds of functions, therefore the analysed program graph features only $\overset{\mathbf{s_0}}{\rightsquigarrow}$ edges. We could use the $\overset{\mathbf{t}}{\rightsquigarrow}$ edges to recognise and eliminate branches that violate the syntactic convention about tuples in function return points, containing the atom with the name of the next state, specified in the gen_fsm specification. The example program graph follows the specification, therefore these edges are not needed anymore, and can be contracted, by applying §13. As a result we get the reduced analysed program graph shown in Figure 6.

5.3 Demonstration of the synthesis stage

In this stage we map the reduced analysed program graph, shown in Figure 6, unto a state machine model, depicted by Figure 7. First, we add *init*(*init*/1) (the initial state created from the function node of `init/1`) to the state machine, since `init/1` is the only element of *Init* in this example. The rule §14 matches the node corresponding this state (i.e. `init/1`) with the function node of `pos1/3`, thus we add *state*(*pos*1/3), and a transition to the state machine. We may choose to do the substitution of $\xrightarrow{\textbf{lab}}$ right now based on Table 5, swapping it to trigger label constructed from the joker pattern pointed by the $\overset{\textbf{trigger/1}}{\rightsquigarrow}$ edge. Next, we can match both §15 and §16 on the `pos1/3` node. The former results in adding a stop state to the state machine, while the latter results in adding a choice state. To substitute $\xrightarrow{\textbf{lab}}$ for a trigger, as specified by the substitution rule, we have utilise the edge numbering to select the pattern nodes corresponding to each transitions: the pattern pointed by $\overset{\textbf{trigger/1}}{\rightsquigarrow}$ will be used to label the transition of the stop state corresponding to the tuple node pointed by $\overset{\textbf{e/1}}{\rightsquigarrow}$, and $\overset{\textbf{trigger/2}}{\rightsquigarrow}$ for the choice state corresponding to the branching expression pointed by $\overset{\textbf{e/2}}{\rightsquigarrow}$. Next, §14 and §15 matches on the node of the choice state. This time, $\xrightarrow{\textbf{lab}}$ is substituted to a guard, constructed with the appropriate expressions pointed by the $\overset{\textbf{cond}_{\textbf{head}}}{\rightsquigarrow}$ and $\overset{\textbf{cond}_{\textbf{branch}}}{\rightsquigarrow}$ edges. With the match of §14, we added *state*(*posOther*/3) to the state machine. The corresponding `posOther/3` node can be analysed similarly to `pos1/3`. Finally, we get the result of the transformation, a complete state machine, as shown in Figure 7.

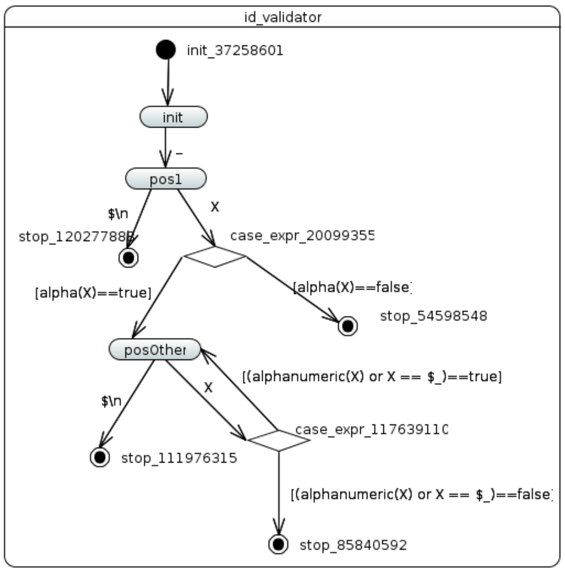


Figure 7: The state machine resulting from the synthesis stage

6 Evaluation

To test and measure the previously presented transformation method during actual physical execution, we also created a reference implementation in Erlang. Instead of creating an application that communicates with a RefactorErl instance via some remote communication protocol, we realised the implementation by extending the source code of the open source RefactorErl framework to eliminate the need of creating a bridge, and to minimise the number of dependencies. To implement the matching rules, we were able to make use of the function clause pattern matching in the Erlang language, and used memoisation technique to avoid repetitive evaluation of functions called from multiple places in the source code. We also implemented a few small extension not mentioned in this paper. For example, initialising special states to stand in place of undiscoverable states, or the recognition of a state machine based on the functions calls that start that state machine. We also used Erlang to generate the XMI files storing the resulting UML state machines, and used txtUML [13], also developed at the Eötvös Loránd University (ELTE) to generate graphical diagrams from these UML state machine models.

We executed the implemented algorithm on Erlang state machines found in large, popular, open source Erlang applications. Our sample state machines are from the sources of the Ejabberd communication server [6], the Riak distributed NoSQL database [9], and the Erlang OTP library [14]. Unfortunately, the more general state machines are less commonly used than the more specialised behaviours, like `gen_server`, and `gen_event`, which means we had to test on a smaller sample. Still, the selected sample of state machines seems to have a nice enough variety both in length and complexity.

Our results are summarised in Table 6, containing, for each Erlang state machine module the number of lines of code in the module, the number of discovered states and transitions in the resulting state machine, and the runtime of the transformation (minimum, maximum, average, median runtimes in microseconds). To measure runtime, we repeated every measurement 500 times for each file, thus creating a 500 element sample for each Erlang state machine. The runtime data does not include the time needed to load the respective Erlang applications in the RefactorErl databases, since this operation only has to be performed once for every software application, and in general, the size of the complete application source code is expected to be independent from the size of the individual Erlang state machines the application contains. Since the resulting state machines usually contain choice states, the measured number of states and edges are expected to be higher than the (ordinary) states explicitly defined by the Erlang state machines.

Table 6: Runtime test results of the implemented algorithm

File name	Lines of code	States	Transitions	Min (μs)	Max (μs)	Avg (μs)	Med (μs)
Ejabberd							
ejabberd_c2s	3128	50	79	321354	519876	339679	343389
ejabberd_http_bind	1236	28	44	137083	170189	144412	144956
ejabberd_http_ws	355	22	25	52735	72020	58953	58706
ejabberd_odbc	692	17	27	46294	64307	50179	51069
ejabberd_s2s_in	712	38	54	91495	126085	99134	99422
ejabberd_s2s_out	1367	84	113	201523	242354	209178	210240
ejabberd_service	404	26	30	51757	91737	58527	58115
eldap	1196	23	39	106526	123567	113383	113893
mod_irc_connection	1581	30	41	108223	150603	116199	116257
mod_muc_room	4501	35	81	164997	280537	172017	173004
mod_proxy65_stream	291	33	37	51259	69289	58666	58485
mod_sip_proxy	458	23	28	40422	61556	44489	45057
Riak							
riak_kv_2i_aae	695	19	31	77877	193215	80248	83386
riak_kv_get_fsm	787	20	17	31662	58225	33244	35118
riak_kv_put_fsm	1055	29	40	81231	148422	83690	87103
riak_kv_mrc_sink	439	25	41	79229	147276	81694	85265
Erlang OTP							
ssh_connection_handler	1721	56	131	3941	19818	4222	5200
tls_connection	975	72	103	3197221	3438752	3290901	3292256

Taking our use case into account, fast runtimes are not of critical importance to the success of the transformation. However, even the slowest execution time, measured with `tls_connection` module in Erlang OTP, is quite small with 3 seconds. We have to remind the reader though, that we used memoisation in the implementation to optimise the runtime: without the elimination of repetitive *SPG* branch evaluations, the algorithm would be noticeably slower.

According to Figure 8 (after eliminating the outlier `tls_connection`), the number of lines positively correlates with time needed to execute the transformation. This phenomenon can be explained by the fact, that the algorithm needs more time to analyse deeper function return points, and the depth of these return points are likely correlated with the length of the source code of the module. Although not shown here, the same connection can be observed between the states in the resulting state machines and the transformation execution time. Since we mapped branching expressions to choice states, and deeper function return points are more likely to contain such branching expressions, it is also more likely that we will create more choice states for those state machines with deeper return points and longer transformation execution times. Figure 9 also shows a positive, although less definite correlation between the number of lines and the number of states. Using the

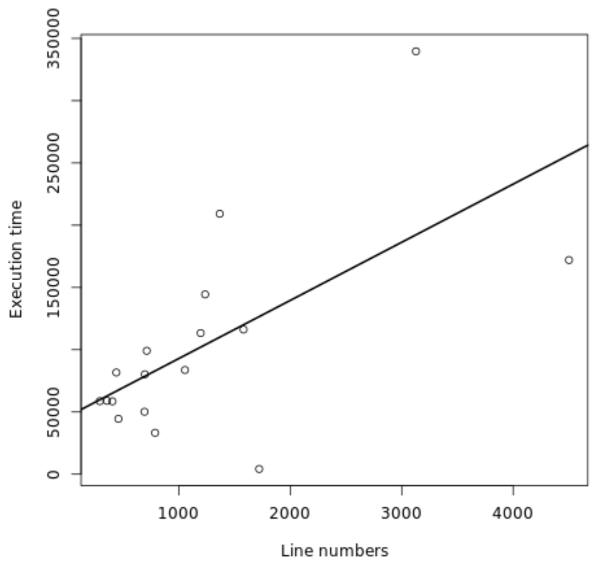


Figure 8: Algorithm execution times in terms of LoC

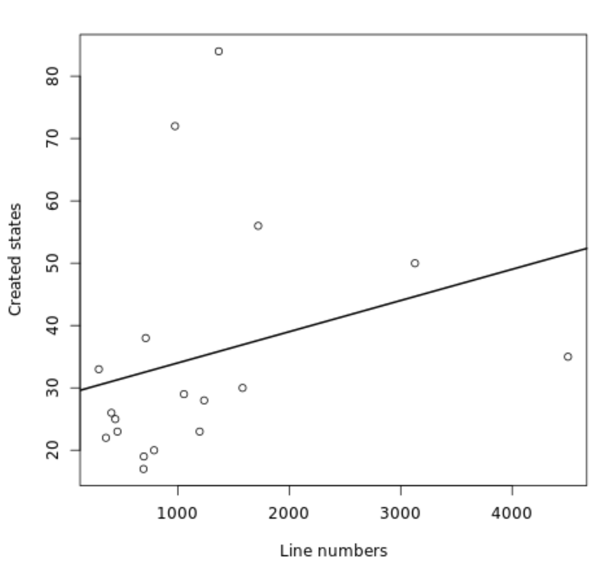


Figure 9: No. of states in results in terms of LoC

same reasoning, we conjecture that deeper function return points will have more lines, and are also more likely to contain branching expressions. Conversely, Erlang state machines with more states, and therefore more transition function definitions, are also more likely to contain more lines of code. Still, because of the low number of state machine modules in the samples, these assertions probably require more thorough research with a bigger sample size, and perhaps with the use of more advanced metrics.

7 Related work

While most CASE tools support source code generation from UML models, the inverse operation, generation of UML models from source code (code-model transformation) is less prevalent. This is probably explained by that code-model transformation requires complex static and/or dynamic analysis tools. For the most popular object-oriented languages, industrial tools suitable for this task usually support the discovery of UML class diagrams and sequence diagrams from the program sources. Such tool are for example ObjectAid [8] and Eclipse MoDisco [5] for Java, Microsoft Visio [3] and Altova UModel [1] for C# s Visual Basic, Visio and Doxygen [2] for C++. Another interesting approach makes use of static and dynamic analysis techniques to detect design patterns in object-oriented source code [23].

Model transformations, a recent, but well researched area can also be of interest concerning state machine transformations. While this methodology is mostly used to specify and execute transformations between models, it can be extended to also handle entities not usually considered to be models, such as syntax trees. Closely related to the topic of our current paper, we presented a procedure to map a subgraph of the *SPG* of RefactorErl to an *SPG* model by utilising triple graph grammars [22] to transform this *SPG* model to a valid UML state machine model [19]. By employing a formal mathematical theory, this approach provably obtained advantageous properties concerning among others the correctness and completeness of its results and the efficiency of its execution. On the other hand, since it utilises high level model transformation concepts, implementations of that procedure are expected to be a magnitude slower than the directly implementable approach presented in our current paper.

Similarly to the tool introduced in the present paper, Erlesy [4] is another tool that can be used to visualise Erlang state machines. Instead of utilising a powerful, but complex static analysis framework, Erlesy uses standard Erlang tools to parse state machine source code. Erlesy does not include choice states and guards in its output, thus the results will possibly represent non-deterministic state machines. Unlike our approach, Erlesy uses loop edges to model the handle callbacks of the `gen_fsm` specification: an advantage of this approach is that it follows `gen_fsm` semantics more closely, a disadvantage is that it inevitably clutters the resulting state machine graphs with loop edges. Erlesy is a readily usable, lightweight solution to visualize Erlang state machines in various output formats, like Graphviz, PlantUML or D3.js.

There is also a mature methodology for discovering deterministic finite state machines using dynamic code analysis, called state machine induction and behavioural inference. Procedures applying this methodology execute the analysed program based on specific use case scenarios (e.g. a sequence of function calls), and collect information to generate a state machine model. This task requires the elimination of non-deterministic transitions, and transitions featuring recurring execution traces. Such state machine reducing methods are the k-tail algorithm, and the QSM algorithm. QSM offers the user various valid reductions, and proceeds to perform the reductions chosen by the user. Later methods can eliminate the need for these user dialogues by utilising static source code analysis to find good answers automatically and with high precision [27]. The Erlang language is also well suited for this task due to its statelessness and advanced program execution tracing facilities [10].

While the dynamic approach to state machine discovery enjoys benefits from the well defined methodology, the requirement for use case scenarios hinders its usability for large systems. With static analysis, it is possible to discover the complete state machine, relying only on the source code. One of the difficulties arising from using static analysis is identifying the relation between the implementation level programming patterns and high level state machine concepts. And even if we solve this problem, in some special cases it is still impossible for pure static analysis to filter out components (e.g. states, transitions) that can be never reached during program executing (e.g. because of conditions that can only be evaluated to false). Thus, it can be stated that both the dynamic and static analysis approaches have their advantages and disadvantages, therefore the choice is dependent upon the goals and requirements of the task at hand. Our approach strongly relies on static analysis, since the requirements of the `gen_fsm` behaviour [14] makes it easier to identify and analyse the programming constructs relevant to state machines, and the RefactorErl framework provides the means to perform deep and comprehensive static analysis on Erlang source code.

8 Conclusions

In this paper, we introduced a method to transform Erlang state machines, implementing the `gen_fsm` behaviour, to state machines described by UML, or similar state machine languages. To analyse Erlang programs, we used the RefactorErl static analysis framework. We also defined an UML-compatible state machine representation to denote the target state machines of the transformations. The presented method consists of three stages: an analysis stage, a transformation stage, and a synthesis stage. In each stage, a backtracking algorithm is executed on the output of the last stage (or, in the case of the first stage, the RefactorErl Semantic Program Graph), and each stage utilises a separate set of transformation rules by means of pattern matching. The algorithms themself are general enough not to depend on any static analysis framework, only the presented transformation rules depend on the structural details of the RefactorErl Semantic Program Graph. By separating the algorithm and the rules, we made it more easier to the extend of the transformation: one just has to add more rules to the rule sets. By specifying the bulk of the transformation logic by rules, we also made it possible to parallelise the algorithm to optimise it for environments with multiple processing units. We demonstrated the execution and the results of each stage by a small example. We were able to realise a relatively fast implementation of the algorithm. After testing our implementation on state machines selected from the source code of large, popular, open source Erlang applications, we concluded that the time needed to perform the transformation on an Erlang state machine is positively correlated with the number of lines of code in the program code of the state machine. We also found similar correlation between the transformation execution time and the states created by the transformation. We explained both phenomenon by conjecturing that deeper return point expressions need more time to be analysed, probably have more lines of code, and contain more branching expressions.

The evaluation of our implementation evidences that the method presented in this paper can be used to construct UML state machine models, even from large and complex Erlang state machines. Still, several opportunity remains for improvement, for example by extending the analysis with more elaborate RefactorErl queries to discover even more state machine elements, or by targeting an even larger subset of all the features provided by the UML state machine language, or even by preparing the method to generate executable state machines.

References

[1] Altova UModel 2016. `http://www.altova.com/umodel.html`. Accessed: 2016-06-30.

[2] Doxygen - Generate documentation from source code . `http://www.stack.nl/~dimitri/doxygen/`. Accessed: 2016-06-30.

[3] Microsoft Visio . `http://visio.microsoft.com/`. Accessed: 2016-06-30.

[4] Visualising Erlang development . `https://github.com/haljin/erlesy`. Accessed: 2016-06-30.

[5] Eclipse MoDisco . `http://www.eclipse.org/MoDisco/`. Accessed: 2016-06-30.

[6] Ejabberd, robust scalable and extensibe XMPP Server. `https://www.ejabberd.im/`. Accessed: 2016-06-30.

[7] Object Management Group. OMG Unified Modeling Language Superstructure. `www.omg.org/spec/UML/`. Accessed: 2016-06-30.

[8] ObjectAid. `http://www.objectaid.com/home`. Accessed: 2016-06-30.

[9] Riak KV, distributed NoSQL database. `http://basho.com/products/riak-kv/`. Accessed: 2016-06-30.

[10] Thomas Arts and Cecilia Holmqvist. In the need of a design... reverse engineering Erlang software. 10th International Erlang User Conference, EUC. 2004.10.

[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[12] Francesco Cesarini and Simon Thompson. *Erlang Programming.* O'Reilly Media, 2009.

[13] Gergely Dévai, Gábor Ferenc Kovács, and Ádám Ancsin. Textual, executable, translatable UML. Proceedings of 14th International Workshop on OCL and Textual Modeling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014) Valencia, Spain, September 30, 2014., pages 3-12.

[14] Ericsson AB. *Erlang Reference Manual.* `http://www.erlang.org/doc/reference_manual/part_frame.html`.

[15] Dániel Horpácsi and Judit Kőszegi. Static analysis of function calls in erlang. *e-Informatica Software Engineering Journal*, 7:65–76, 2013.

[16] Zoltán Horváth, Lászó Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babe-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, Jul 2009.

[17] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[18] Dániel Lukács. Erlang állapotgépek elemzése és transzformálása UML-re. Scientific Students' Associations Conference, ELTE, Budapest, Hungary, 2016.

[19] Dániel Lukács. Erlang állapotgépek modell alapú és transzformációja UML-re. Scientific Students' Associations Conference, ELTE, Budapest, Hungary, 2016.

[20] Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML(TM).* Cambridge University Press, New York, NY, USA, 2004.

[21] Miro Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems.* Electronics & Electrical. Taylor & Francis, 2009.

[22] Andy Schürr. *Specification of graph translators with triple graph grammars*, pages 151–163. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[23] Nija Shi and Ronald A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 123–134, Sept 2006.

[24] Melinda Tóth and István Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer, 2012.

[25] Melinda Tóth, István Bozó, Zoltán Horváth, and Máté Tejfel. First order flow analysis for Erlang. In *Proceedings of the 8th Joint Conference on Mathematics and Computer Science (MACS), ISBN:978-963-9056-38-1*, 2010.

[26] Melinda Tóth, István Bozó, Judit Kőszegi, and Zoltán Horváth. Static Analysis Based Support for Program Comprehension in Erlang. In Acta Electrotechnica et Informatica, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), ISSN 1338-3957 (online), pages 3-10.

[27] Niel Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 209–218, Oct 2007.