STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.03

TRANSLATING ERLANG STATE MACHINES TO UML USING TRIPLE GRAPH GRAMMARS

DÁNIEL LUKÁCS AND MELINDA TÓTH

Abstract. In this paper, we present a method that transforms event-driven Erlang state machines into high-level state machine models represented in UML. We formalized the transformation system as a triple graph grammar, a special case of graph rewriting. We argue in this paper that using this well-defined formal procedure opens up the way for verifying the transformation system, synchronizing code and formal documentation, and executing state machine models among many other possible use cases. We also provide an example transformation system and demonstrate its application in action on a small Erlang state machine. We also present our evaluation of our full system implementation tested on real world Erlang state machines.

1. Introduction

In this paper, we introduce a method to generate formal UML state machine models from executable Erlang state machine source code (e.g. `gen_fsm` applications [13]). First, this transformation makes use of the RefactorErl static analysis framework [11, 18] to analyse the application source code, then it will transform and synthesize the program representation resulting from the analysis, into a UML state machine model.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15, 68Q42.

1998 *CR Categories and Descriptors.* D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques – *Computer-aided software engineering (CASE)*; D.2.1 [**SOFTWARE ENGINEERING**]: Requirements/Specifications – *Languages*; F.4.2 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Grammars and Other Rewriting Systems – *Parallel rewriting systems*; F.3.22 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages – *Program analysis*.

Key words and phrases. Erlang, triple graph grammar, UML, CASE, state machine, model transformation.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

1.1. **Motivation.** Currently, Unified Modeling Language (UML) [4] is mostly used in practice as a documentation tool to present a diverse set of views (we refer to as *models* in this paper) on high complexity software code.

As both time and tangible costs of formation and maintenance of documentation increases with software size, automatic generation of collective knowledge representation becomes more and more important as the software grows. Apart from automatically generating well-defined UML models of Erlang state machines, the focus of this paper is to achieve this in a formal way, specifically by using a model transformation approach called triple graph grammar.

Triple graph grammars [17] (TGGs) are special kind of graph rewriting systems, and as formal methods, they may have the same properties proved for them, such as correctness, completeness, determinism, and confluence. Proven transformation systems automatically produce verified documentation.

One property of special interest regarding TGG transformations is information preservation, i.e. the expectation that result models can be transformed back to the source models. Such bidirectional transformations, and the more effective model integration (finding correspondences between models) and model synchronization (completing partially complete models and correspondences) allow generation of software code from documentation and vice versa, providing a way to keep the two in sync all the time. This enables developers to work on the software in various abstraction levels.

As some UML metamodels (such as state machines) can be used to describe program semantics, model execution by automatic generation of Erlang `gen_fsm` implementations of UML state machines is also a possible future direction of this research.

Verification of the transformation system, automatic verification and synchronization of documentation, bidirectional TGGs, model execution and round-trip editing are future goals and possible applications of this research. This current paper only focuses on the target model generation aspects.

1.2. **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. A state describes a segment of the program behavior, while a state transition describes a change in such behaviors, usually triggered by an event [16].

Among many others, UML [4] is one of the more accepted standards to represent state machines. 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 behavior and wait for a possibly different set of events. etc.

Erlang [13] 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 (called *behaviors*) to support these applications provided by the built-in OTP library, including the state machine (`gen_fsm`) behavior.

The requirements of the `gen_fsm` behavior 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 [13].

In our research, we use the RefactorErl static analysis framework [18] to analyse Erlang source code. RefactorErl analyses the source code and stores discovered syntactic and semantic information in a database called the *semantics program graph (SPG)*. The framework provides several features 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.

2. Transformation pipeline

In this section, we overview our approach that we refer to as *transformation pipeline*. The pipeline can be considered as a simple function composition, where the output of earlier functions will be the input of the latter functions.

As depicted in Figure 1, the main input of pipeline is an Erlang SPG as stored by the RefactorErl framework. This is transformed into an SPG model (a highly detailed view of the transformed software code), in order to refine it into a high-level state machine model using model transformation methodology. Finally, this high-level state machine model is translated into a selected state machine model, e.g. UML.

In order to properly understand models and model transformations, we need to introduce basic modeling terminology. The Object Management Group Meta-Object Facility (OMG MOF) [3] highlights four cognitive layers of software modeling: concrete implementation, model, metamodel, and meta-metamodel, each in order a higher-level abstraction (or language) that enables expressing the layer below.

As most of the intermediate values in the pipeline are models, we also indicated in Figure 1 the metamodels generating these models. Models of SPGs are formalized using the SPG metamodel (see Figure 8), state machines

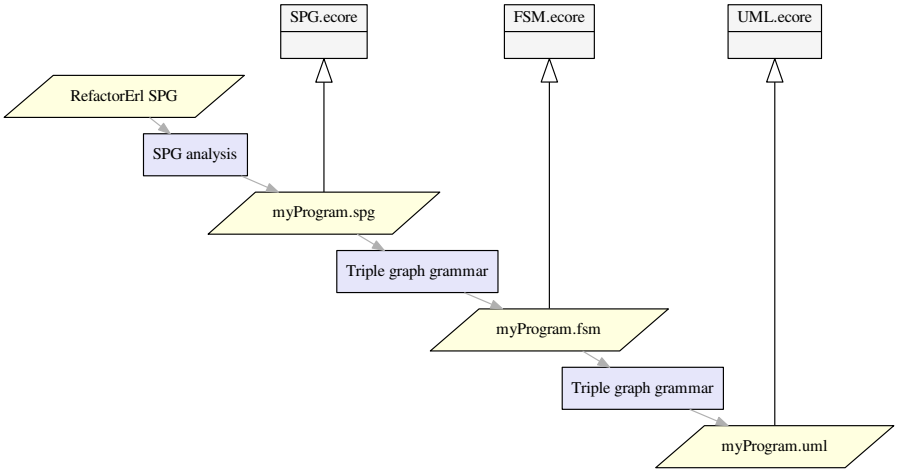


FIGURE 1. Transformation pipeline refining Erlang state machines to high-level UML models

are represented internally using the FSM metamodel (see Figure 2), and the output state machines are formalized in UML.

As it is conventional in model-driven engineering (e.g. in UML and EMF Ecore), we formally represent models using typed attribute graphs [7], where nodes and edges are mapped to types and unique key-value stores.

2.1. **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 the *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.

The first step in the transformation pipeline is the transformation of the RefactorErl SPG into an SPG model (a highly detailed view of the source program), which can be then transformed into a state machine model (a more abstract view of the source program) using standard model transformation tools. A diagram of our SPG metamodel can be found in our earlier work [15], and it also is depicted by Figure 8. During the transformation, we have to perform semantics queries, dataflow analysis and dynamic function call analysis

provided by RefactorErl [20, 19, 10], and encode the results in the model. This way we avoid re-implementing RefactorErl static analysis facilities for models.

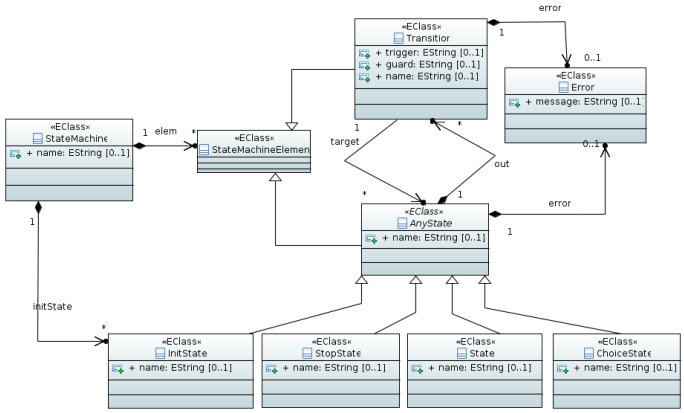


FIGURE 2. Metamodel for representing state machine models so they can be easily translated to UML

2.2. **A metamodel for describing abstract state machines.** In this section we will describe a simple state machine metamodel, already published in our earlier work [15], and depicted by Figure 2, with which we represent the target state machines of the transformation. We showed in [14] how 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 utilize from UML. 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.

2.3. **Triple graph grammars.** The pipeline also includes two model transformation steps expressed using *triple graph grammars (TGGs)*. The term *graph grammar* is a synonym for *graph rewriting system (GRS)*, where the term grammar signifies the intent of language generation as opposed to e.g. program evaluation with graph reduction. TGGs constitute a special class of graph grammars, where the graph is a *triple graph (TG)*: a side-by-side representation of two models with correspondence nodes connecting them. We may consider the correspondence nodes as hyperedges that connect multiple source and target nodes.

The domains given by the metamodels of the source and target models, plus the domain of the correspondence graph always partitions the node and edge

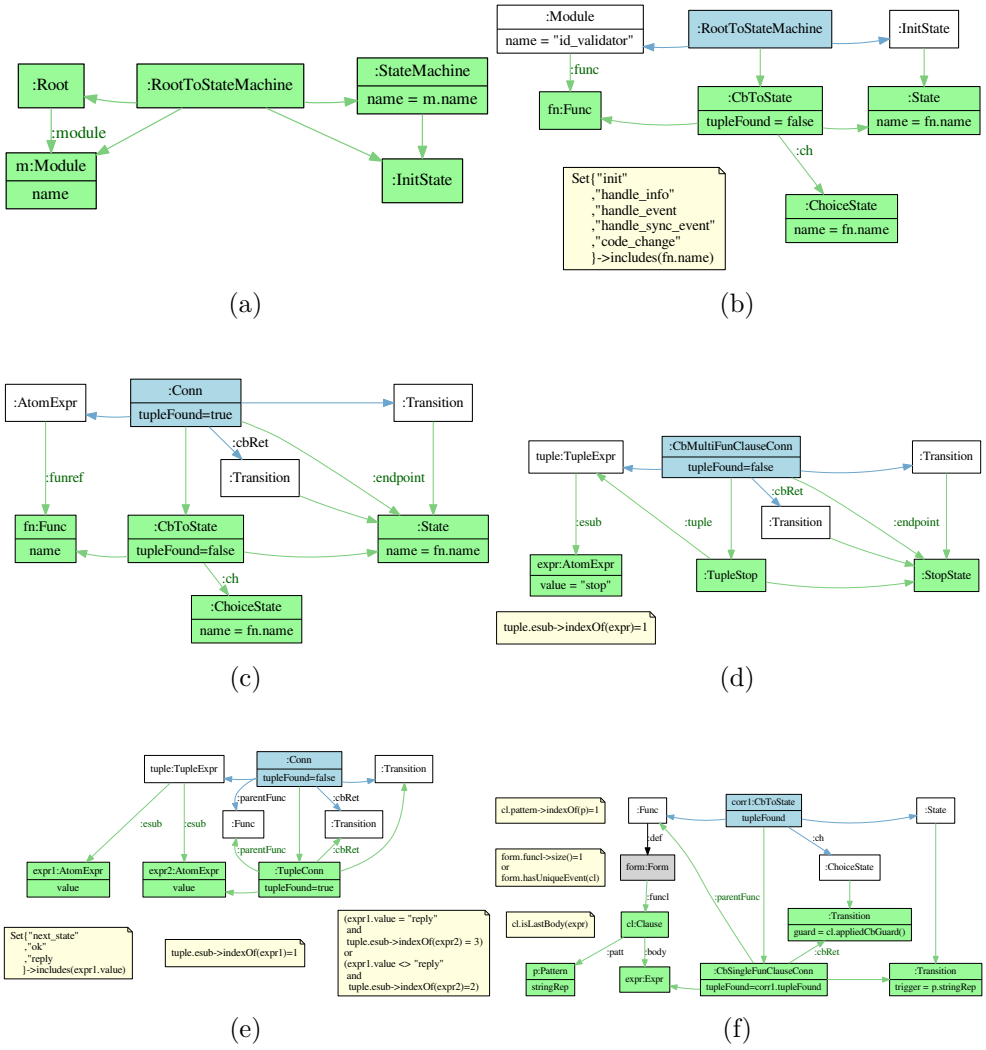


FIGURE 3. Example TGG transformation system

set of both data and rules TGs into three disjunct partitions: the *source*, the *target*, and the *correspondence domains*.

A TGG consists of two kinds of rules: axioms and production rules. TGs are generated recursively by first acquiring an initial TG using the axiom, and then applying the production rules first to the initial TG and then to successive TGs to acquire the final TG. As in GRSs, the left-hand side (LHS) of TGG

production rules is matched using graph matching to a TG redex and the matched values are substituted in the right-hand side (RHS) variables. The resulting concrete RHS is then substituted in place of the redex. We say that an element is *bound* if it was already matched or replaced at least once during the transformation, otherwise we say it is *unbound*.

In this paper, we use a concise notation for diagrammatically representing TGG rules using diagrams (see Figure 3). As side-by-side representation of LHS and RHS usually includes large number of identical nodes, we merge the two in one diagram and use color coding to keep the two comparable. White elements (called *context elements*) appear on both LHS and RHS, so these are "read-only" elements, that are matched and left unchanged. Context elements can be matched both to bound and unbound elements in the data TG. The meaning of green elements depends on whether they occur in the source domain, or not. In the source domain they denote context nodes, that can only be matched to unbound elements (thus preventing infinite applications of the same rule to the same contexts). In other domains they denote RHS-only elements (called *product elements*): instead they are instantiated when the LHS matches.

The application semantics of axioms are identical to those of production rules, but axioms never contain context nodes. This guarantees that axioms are always matched and applied before production rules.

Reusable elements denoted by the color gray (see e.g. Figure 3f) can be matched both to bound and unbound elements in the data TG, but if such a match does not exist, then these elements are created in the data TG.

Most TGG formalisms also allow rules to be accompanied by *OCL constraints*. These are boolean expressions pertaining the attributes and values of the matched nodes. If a rule is successfully pattern matched, then the OCL constrains are evaluated and the match will be accepted based on whether the expression is satisfied or not.

2.4. **Transformation system.** In this section, we present a small, simplified subset of our TGG transformation system that transforms model SPGs of `gen_fsm` programs to state machine models. This rule set was included only to illustrate and to aid in the comprehension of Section 3. To develop a TGG rule set that assumes full coverage of the Erlang language, various problems had to be solved: propagation of state between rule applications, transforming parallel paths in the graph, transforming nodes with arbitrary number of incoming or outgoing edges, transforming recursively nested expressions.

As it is difficult to demonstrate all these problems in a small example, we avoided the discussion of related details in the rules not occurring in the example. We detail in [14] the whole set consisting of 32 rules, of which the

largest one has 16 nodes. According to [12] practical TGGs have in average 15-20 rules, with 10-40 nodes each. The larger number of rules in our case is explained by the number of types and syntactic categories in the Erlang language.

As the rules in the system have disjunct LHSs and/or mutually exclusive OCL constraints, the system is guaranteed to be deterministic.

Figure 3a depicts an axiom rule. In the source domain, it matches in the data TG the bound SPG root node and module node implementing the `gen_fsm` behavior. It then generates and binds in the TG the root node of the state machine and a globally unique initial state, and a correspondence node between the source and target elements. The `name` attribute of the module will also be matched to a value, and the state machine will be created with its own `name` attribute set to this value.

The production rule in Figure 3b matches the SPG nodes of the initial `gen_fsm` callback functions and creates a state for each of them, and a transition into this state from the initial state. The name of the new state will be set to the name of the matched function.

The transformation traverses the SPG model starting from the initial functions and identifies next states in the function return values. As transition functions bear the name of their source state, we can now identify the transition function to traverse next. The leaves of the search tree are stop states and already bound states.

Figure 3c illustrates the identification of new functions, assuming we already matched the atom in the predecessor functions return value. It is similar to Figure 3b, but we expect the `tupleFound` attribute of the correspondence node to be true: this signifies that the atom in question is part of the expression tree headed by a tuple. Successive correspondence nodes will have their `tupleFound` attribute set to false again, as from here on we will analyse the body of another function.

Figure 3d introduces a stop state if the return value of the transition function is a tuple whose first element is the atom `stop`.

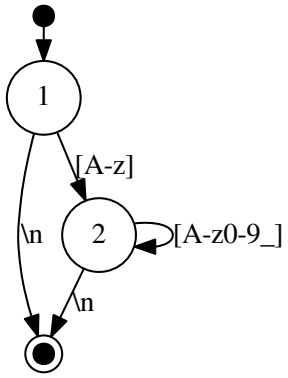
If the first element is not the `stop` atom, Figure 3e enables the traversal of the expression tree of the second tuple element by rules specific to Erlang expression types. It also sets `tupleFound` to true to allow Figure 3c to match when an atom is found. To keep the rules simple we omitted the case where the first element of the tuple is an unreduced expression, but this case can be handled similarly.

As most functional languages, Erlang also allows functions to have multiple clauses. When `gen_fsm` transition functions have multiple clauses, each clause

may declare transitions into different states, therefore in such cases, we introduce a `ChoiceState` for each event handled by the function. Clause conditions (expression patterns and guards) will be mapped to guards of the transitions commencing from the `ChoiceState`. On the other hand, when a transition function has only one clause, we do not want to introduce a `ChoiceState`, as it would only have a single transition. Figure 3f introduces a rule for handling function clauses which have a unique state machine event pattern (this includes clauses that are entirely single).

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

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

terminate(_, _,_) -> ok.

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.
```

(a)



(b)

FIGURE 4. Source code and schematic diagram of the Erlang state machine that accepts the language of identifiers

3. DEMONSTRATION

The goal of this section is to demonstrate the example transformation system on a small Erlang state machine. To keep the example system and the demonstration section small and concise, we only transform the first part of the model SPG of this program. We included a trace of the transformation of the full syntax tree in [14].

Figure 4a depicts the code of an Erlang `gen_fsm` state machine that accepts the language of identifiers, i.e. those words (event sequences) that start with a letter and continue either with letters, numbers, or underscores (see Figure 4b). Initially, the `gen_fsm` behavior first executes the `init/1` function, and sets

the current state to the state name (`pos1`) inside the tuple returned by this function. When an event is sent to the state machine, the `gen_fsm` behavior will evaluate the transition function corresponding to the current state (`pos1/3`) and now in turn sets the current state to the return value of this function. The state machine stops (and either rejects or accepts the event sequence received beforehand) when one of the transition functions return a tuple with the (`stop`) atom.

Figure 5 depicts the transformation trace, i.e. the triple graph resulting from applying the transformation system in Section 2.4 to the model SPG of this program. Nodes not featured in the final result were omitted for the sake of simplicity. The left-hand side of the TG stores the original model SPG unchanged, while the right-hand side stores the resulting state machine. In middle, the correspondence graph tells us the history (i.e. the rule application sequence) of the transformation.

First, the axiom rule (Figure 3a) was the only rule that could have been matched to the root node, and thus the state machine root was created. Next, the rule in Figure 3b is matched, as `init/1` is part of the five callback functions expected by the `gen_fsm` behavior specification. We handle these functions similarly to transition functions, and therefore we create a special state for `init/1` too. On the only function clause, the rule in Figure 3f is applied. This rule selects the last expression of the function body and assigns a transition corresponding to this clause. The transition will be labeled by the wildcard trigger which is the first parameter pattern of the clause. As the selected expression may be nested, further rules must be applied to find the name of the next state in this expression. In the current case, the expression is a tuple and matches the rule in Figure 3e, since the first element of the tuple is the atom `reply` and its third element is an atom. The `tupleFound` attribute of the correspondence node is set, so that rules aimed to match only subexpressions of the tuple may match. And indeed, the rule in Figure 3c matches: it finds that the third element of the tuple is the atom `pos1`, and thus it binds the node representing the transition function `pos1/3` and creates the corresponding state.

As `pos1/3` has two clauses the rule in Figure 3f matches both. In the case, where the event is the end-of-line character, the result is a tuple with the `stop` atom as its first element, thus the rule in Figure 3d matches and creates a stop state. The last expression of the other clause is a branching expression. A choice state will necessarily correspond to these expressions, and then each branch will have a corresponding transition from the choice state. For brevity, we omitted the rules needed to perform these transformations, along with the rest of the trace. Both can be found in [14].

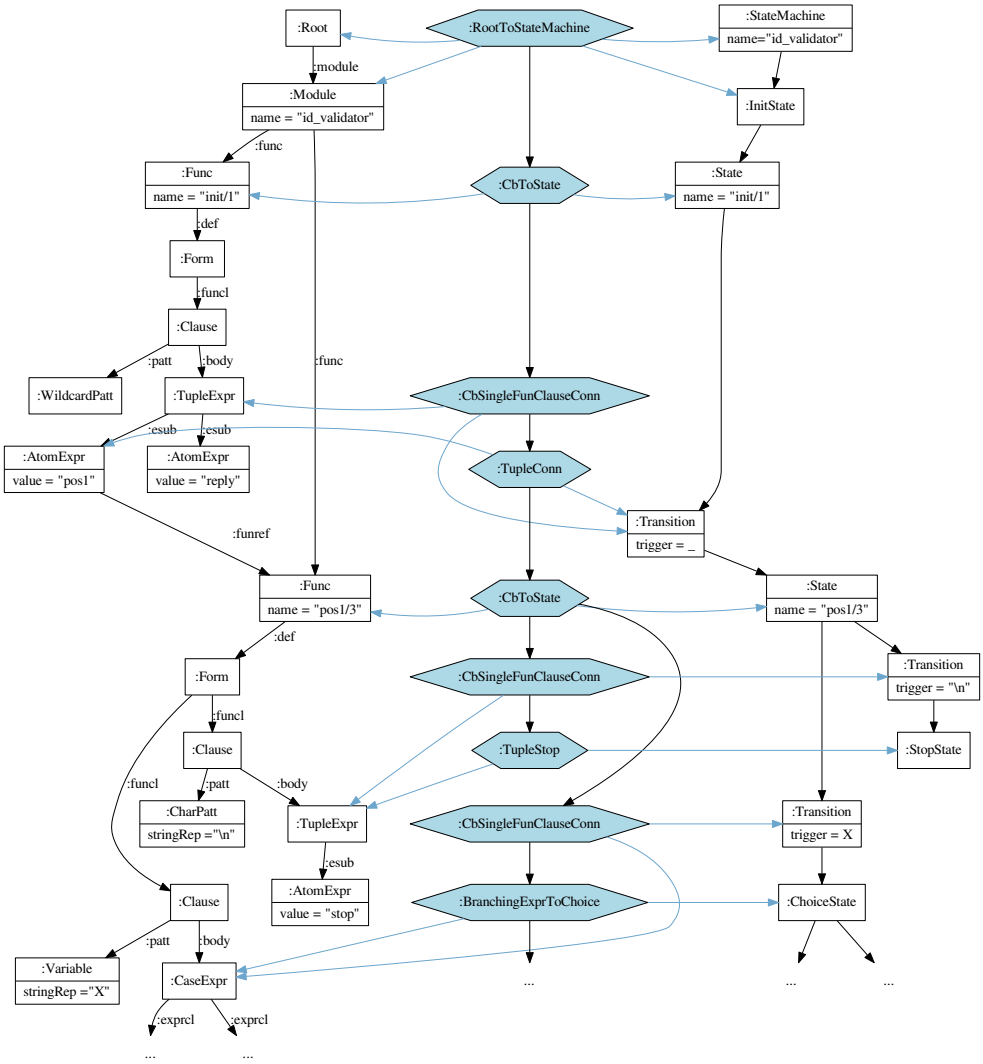


FIGURE 5. Partial trace of the transformation applied to the SPG of the Erlang state machine in Figure 4a

The final result of the transformation (including the translation between our abstract state machine metamodel and UML) is shown by Figure 6 visualized by txtUML [6].

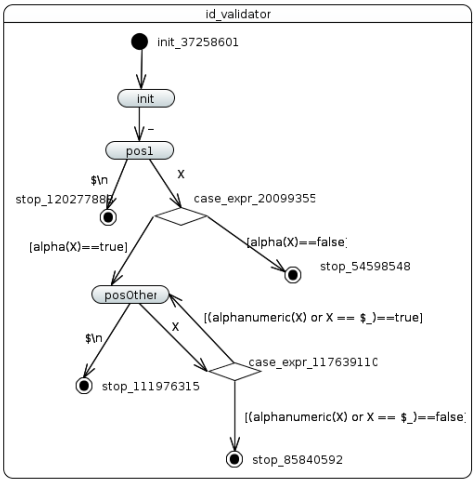


FIGURE 6. The final UML state machine resulting from the transformation of the Erlang state machine in Figure 4a

4. EVALUATION

In this section, we discuss and evaluate our implementation of the approach presented in this paper for transforming Erlang state machines to UML. As a superset of the small example transformation system introduced earlier, our implementation consists of 32 rules, of which the largest one has 16 nodes, and in our experience it is capable of transforming arbitrary Erlang state machines implementing the `gen_fsm` behavior. This larger system is detailed in [14].

In the implementation, we used the RefactorErl framework and its semantic query, dataflow analysis, and dynamic function call analysis facilities to construct the model SPG in EMF Ecore. We implemented the TGG transformation system in the TGG Interpreter [8] tool, as it supports custom correspondence metamodels, OCL constraints, and reusable nodes. Finally, we used the txtUML [6] framework to visualize UML state machines.

For evaluating the implementation we select state machine modules from large, open source Erlang projects: the Ejabberd communication server, the Riak distributed NoSQL database, and the Erlang OTP library. The UML state machine model resulting from the transformation of a smaller module is depicted in Figure 7. For each module, we list the number of lines of code, the average, variance, median, and extrema of required time (in miliseconds) for performing the transformation based on 10 measurements, and the number of states and transitions in the result. States include choice states and entry and exit states, so the number of states and transitions in the model may exceed

the actual number of states in the original Erlang state machine. Time values only concern the transformation of SPG metamodel instances to abstract state machine metamodel instances. The time needed to load the Erlang application in RefactorErl is mostly independent of the state machine module, as in most cases its (often much larger) dependencies also have to be loaded. In our experience, the time needed to translate abstract state machines to UML was negligible compared to the SPG transformation.

Module	LoC	#States	#Transitions	Avg (ms)	Var (ms)	Med (ms)	Min (ms)	Max (ms)
Ejabberd								
ejabberd_c2s	3128	46	90	31614.7	9504.77	28402.0	22933	47799
ejabberd_http_bind	1236	22	23	26350.2	7598.95	23545.0	21857	47411
ejabberd_http_ws	355	14	13	4924.8	429.09	4742.5	4543	5879
ejabberd_odbc	692	7	10	3128.3	185.25	3109.5	2827	3498
ejabberd_s2s_in	712	34	48	22819.6	3478.57	22498.0	17400	29095
ejabberd_s2s_out	1367	80	104	75686.1	6822.30	73770.0	65274	86216
ejabberd_service	404	22	23	17991.3	3778.44	17035.0	13501	24403
eldap	1196	19	32	27353.0	7569.92	25156.5	21774	47353
mod_irc_connection	1581	26	25	28931.5	9540.48	24327.0	18330	40679
mod_muc_room	4501	32	73	37370.5	7095.74	36863.0	29746	52170
mod_proxy65_stream	291	29	32	14975.9	3016.94	13558.0	12955	22192
mod_sip_proxy	458	19	24	10418.7	1423.34	10162.5	8329	12292
Riak								
riak_kv_2i_aae	695	15	22	11688.4	2191.49	10927.0	10183	17181
riak_kv_get_fsm	787	16	16	5521.9	1408.34	4878.0	4324	8377
riak_kv_put_fsm	1055	25	39	12630.0	1822.41	12816.0	10862	16779
riak_kv_mrc_sink	439	14	23	7795.4	1302.02	7417.0	6455,	10546
Erlang OTP								
ssh_connection_handler	1721	42	65	67467.6	3578.80	66465.5	63174	73920
tls_connection	975	61	80	56788.5	3906.75	55821.5	50383	63514

TABLE 1. Runtime evaluation results

The time needed weakly positively correlates with the number of lines of code (LOC). We can explain this by assuming that more complex state machines also need more time to be processed, and complexity grows linearly in the best case (exponentially in the worst case) with the depth of Erlang expressions (e.g. larger call chains), which in turn may correlate with the LOC. And indeed, based on Table 1, transformation time grows linearly with the number of states in the state machines. Future work may consider more elaborate source code and model metrics to provide better estimates of transformation time and output.

Comparing the time demand of the TGG approach to our less formal, more implementation-centered approach in [15], we can conclude – based on the performance of the current implementation and TGG Interpreter – that the price of the formal guarantees provided by TGGs is the decreased runtime efficiency of the transformation. Still, we believe the advantages provided by TGGs make this approach superior, especially as a reference implementation, that can be used as a foundation for later optimizations.

5. Related work

This work is the continuation of our earlier research published in [15]. There, we present a less formal approach to automatically generate UML models from Erlang state machines. The algorithm presented there is an extended depth first search that selects the neighbouring nodes to discover based on predefined rules, while it simultaneously constructs the output state machine. While this procedure was more efficient, it lacked several features TGG promises for future research: verification of transformation properties, model executability, and model synchronization. We intend this current paper as a foundation to realize these features.

One work with similar goals is Erlesy [1], a readily usable, lightweight solution to visualize Erlang state machines in various output formats, like Graphviz, PlantUML or D3.js. 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.

There is also a mature methodology for discovering deterministic finite state machines using dynamic code analysis, called state machine induction and behavioral 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. The Erlang language is also well suited for this task due to its statelessness and advanced program execution tracing facilities [5].

To implement the transformation system introduced in this paper, we used the TGG Interpreter [8] tool. A survey of various TGGs can be found in [9], that compares MoTE, eMoflon and TGG Interpreter regarding their usability, expressivity and provided formal guarantees. All three tools are based on the EMF framework [2]. Other related tools are Henshin-TGG, EMorF, and OMG QVT.

6. Conclusions and Future Work

In this work, we presented an approach to transform Erlang state machines into high-level state machine models represented in UML using triple graph grammars. For demonstrating this approach, we provided an example transformation system, which we used to explain basic ideas about the semantics of triple graph grammars and the core problems regarding transformation of Erlang syntax trees to high-level state machine models. For the full system consisting of 32 TGG rules, we referred the reader to our earlier technical report. Our implementation of the system used the static analysis facilities of the RefactorErl framework to construct the model SPG in EMF Ecore, and we

implemented the transformation system in the TGG Interpreter tool. We also evaluated the results and efficiency of this implementation.

One certain use case for a tool generating high-level models from code is automatic documentation generation, but TGGs promise possibilities way beyond that. Future research may consider the possibilities of developing a system for reverse transformation to achieve executable UML state machines. A bidirectional TGG system may make automatic synchronization of code and documentation possible and open up the way for round-trip editing to enable development on the most adequate abstraction level.

As TGGs are a special kind of graph rewriting systems, they may have the same properties formally proved for them, such as correctness, completeness, determinism, and confluence. To enable automatic verification of the resulting documentation artifacts, future research may also set out to prove the transformation system introduced in this paper.

References

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

[2] Eclipse Foundation. Eclipse Modeling Framework (EMF). `https://eclipse.org/modeling/emf/`. Accessed: 2015.11.30.

[3] Object Management Group. OMG Meta Object Facility (MOF) Core Specification. `http://www.omg.org/spec/MOF/`. Accessed: 2015.11.30.

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

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

[6] Dévai, G., Kovács, G. F., and Ancsin, A. 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.

[7] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[8] Greenyer, J. and Rieke, J. (2012). Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata. In Schürr, A., Varró, D., and Varró, G., editors, *Applications of Graph Transformations with Industrial Relevance*, pages 222–237, Berlin, Heidelberg. Springer Berlin Heidelberg.

[9] Hildebrandt, S., Lambers, L., Holger, G., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., and Schürr, A. (2013). A Survey of Triple Graph Grammar Tools. In *Bidirectional Transformations*, volume 57, pages 1–18. EC-EASST.

[10] Horpácsi, D. and Kőszegi, J. (2013). Static analysis of function calls in erlang. *e-Informatica Software Engineering Journal*, 7:65–76.

[11] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A. N., Nagy, T., Tóth, M., and Király, R. (2009). 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 Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania.

[12] Kindler, E. and Wagner, R. (2018). Triple graph grammars: Concepts, extensions, implementations, and application scenarios.

[13] Logan, M., Merritt, E., and Carlsson, R. (2010). *Erlang and OTP in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[14] Lukács, D. (2016). Erlang állapotgépek modell alapú és transzformációja UML-re. Scientific Students' Associations Conference, ELTE, Budapest, Hungary.

[15] Lukács, D., Tóth, M., and Bozó, I. Transforming Erlang finite state machines. In CEUR Workshop Proceedings 2046: pp. 197-218. (2018) Proceedings of the 11th Joint Conference on Mathematics and Computer Science (MACS16). Eger, Hungary, 20-22 May, 2016.

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

[17] Schürr, A. (1995). *Specification of graph translators with triple graph grammars*, pages 151–163. Springer Berlin Heidelberg, Berlin, Heidelberg.

[18] Tóth, M. and Bozó, I. (2012). 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.

[19] Tóth, M., Bozó, I., Horváth, Z., and Tejfel, M. (2010). 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.*

[20] Tóth, M., Bozó, I., Kőszegi, J., and Horváth, Z. 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.

7. Appendix

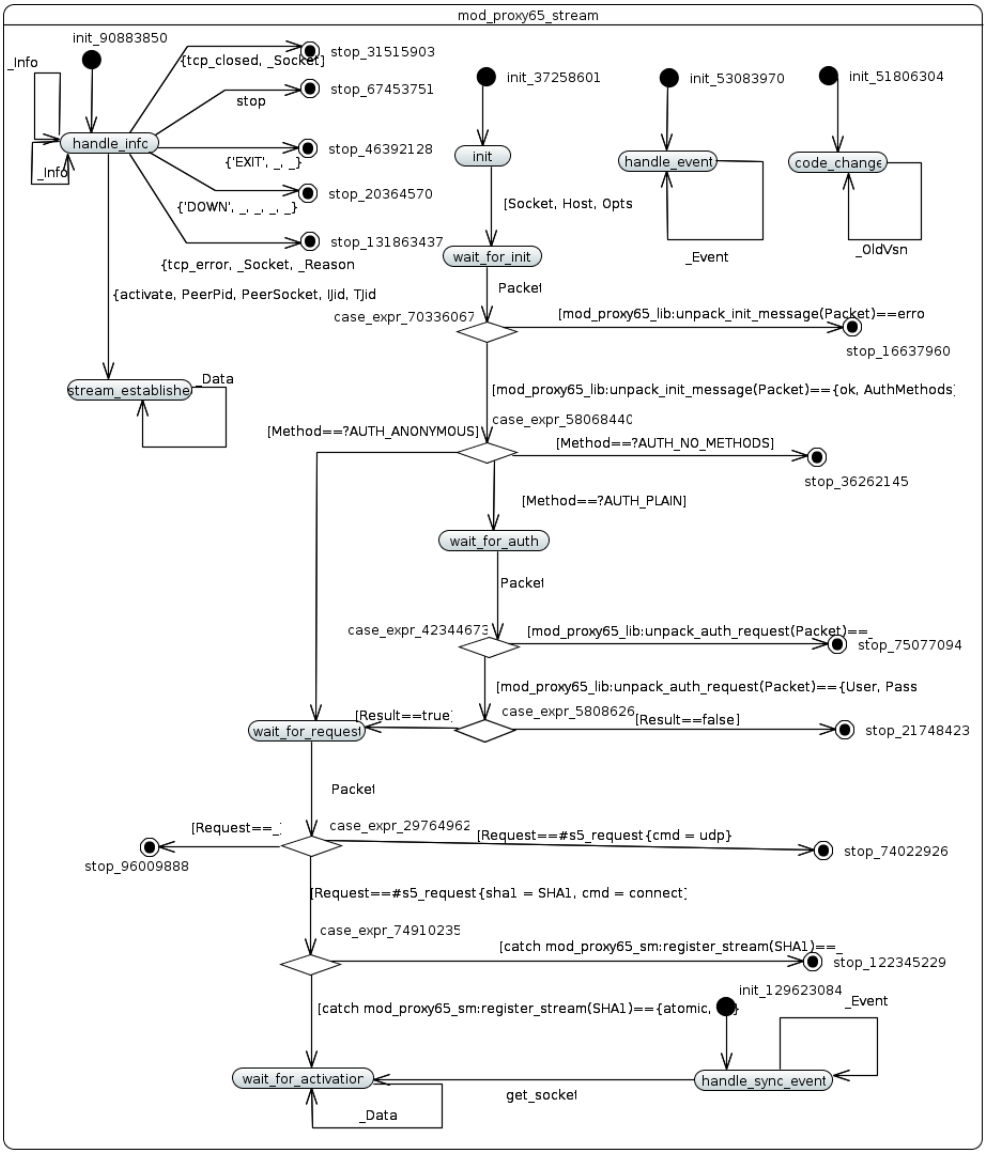


FIGURE 7. The final UML state machine resulting from the transformation of the `mod_proxy65_stream` state machine in Ejabberd

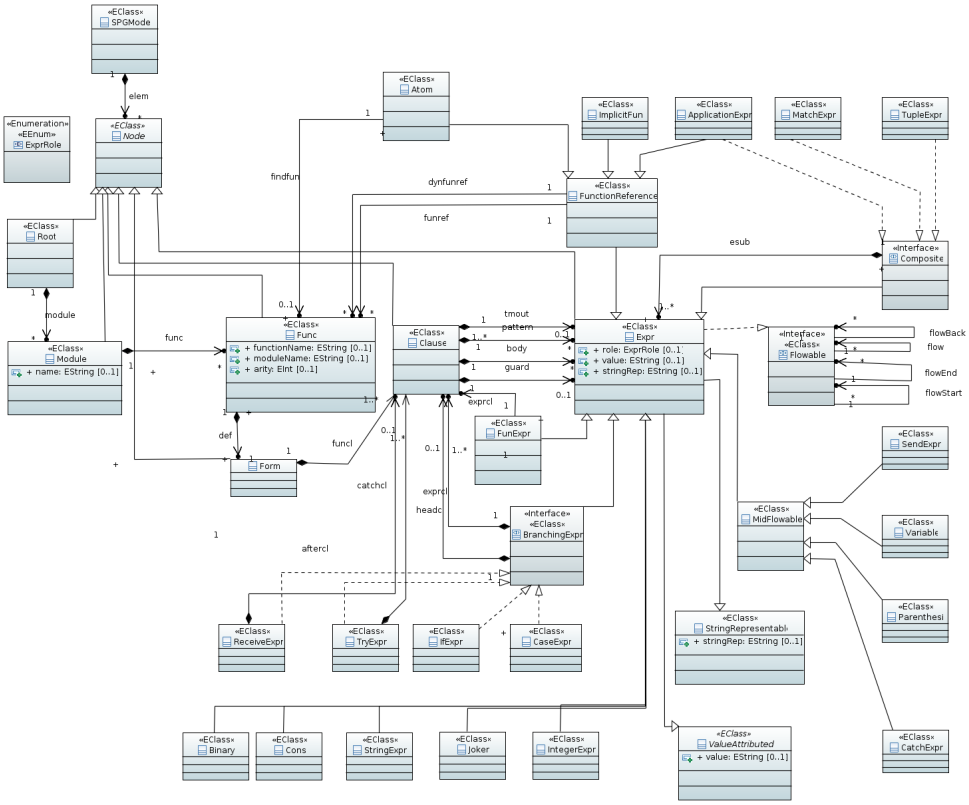


FIGURE 8. Metamodel for representing RefactorErl semantic program graphs as models

EÖTVÖS LORÁND UNIVERSITY, BUDAPEST, HUNGARY
Email address: {dlukacs,tothmelinda}@caesar.elte.hu