Extracting TLA$^+$ Specifications Out of a Program for a BEAM Virtual Machine

Andrius Maliuginas, Karolis Petrauskas

Vilnius University, Faculty of Mathematics and Informatics,
Institute of Informatics, Didlaukio g. 47, LT-08303, Vilnius
andrius.maliuginas@mif.stud.vu.lt, karolis.petrauskas@mif.vu.lt

Abstract. Formal specifications are mathematical descriptions of the desired system functionality. Since they are usually written separately from the software itself, it is important to ensure that the software implements what the specification requires. A common approach to achieve this is to have a specification detailed enough to generate source code but those are rarely written due to expertise required. If code is not generated, then currently there is no straightforward way to reliably show that implementation conforms to initial formal specification. This research attempts to define a way to extract formal TLA$^+$ specification by translating Elixir source code and generating detailed specification to give the system developer the ability to show that it refines the initial one.

Keywords: TLA$^+$, Elixir, translation, specification refinement, distributed systems, message passing.

1 Introduction

Distributed systems are well known for their complexity [1]. As a result, many methods have been developed to prevent mistakes during their development, formal specifications being one of them. However, even having a formal specification is not a guarantee of a correct system – there is also a matter of ensuring that implementation conforms to the specification. Since manual analysis is slow and error-prone, several automated methods have been developed over the years to simplify the process, such as generating implementation code from a rather detailed specification (e.g. [2]). In this paper we attempt to go the other way – to develop a method to extract a detailed TLA$^+$ specification from Elixir source code. We do that by defining the Elixir source code translation into TLA$^+$ and generating the detailed specification. Later, refinement mapping could be shown between the generated specification and a more abstract one, thus demonstrating, that implementation has the same properties as the abstract specification.

DOI: https://doi.org/10.15388/LMITT.2024.14

This approach allows avoiding frequent manual changes to detailed specification as the code changes, which results in development process simplification and a decrease in developer expertise required.

We define translation for source code written in the Elixir programming language [3]. It is a language for BEAM virtual machine, which, due to its process model, makes it easy to develop a distributed system. Elixir also has extensive abstract syntax tree manipulation capabilities [4], which helps with source code analysis.

TLA$^+$ [1] has been chosen as a target for our translation. It is a formal specification language, developed to address the challenges posed by specifications of distributed systems. It is a mathematical specifications language, which makes it programming language agnostic and allows specifying systems on a higher level than code.

There have been attempts to develop specification extraction in the past for Erlang programming language, which is another language for BEAM virtual machine, e.g. [5], which translates Erlang into μCRL specification language. We base our work on previous work done for Elixir and TLA$^+$ – [6], which develops a way to translate and generate sequential code into PlusCal and from there into TLA$^+$. In this paper, the focus is on extracting specification for interprocess communication – how messages are sent between the processes. We base our translation on GenServer module usage – it is an Elixir standard library module that simplifies the development of processes that receive messages and keep state [4]. This allows us to look at the system from a higher level and abstract details which are not important for message passing between processes.

In this paper term "translation" refers to the process of turning one language into another, in our case Elixir into TLA+. Term "generation" refers to the automated creation of detailed specification files which contain the translated source code.

2 Distributed systems model

We model a distributed system as a set of processes, which send messages to and receive from a global set of inflight messages. Each process is completely synchronous and independent from others. We consider the set of messages in flight unordered; messages can be delivered to processes in any order. We also assume that processes do not crash, they cannot be created nor destroyed.

We base source code translation on GenServer Elixir module usage, i.e. we consider only implementations that use functions from this module to communicate between the processes. We consider such a decision justified since the GenServer module is a part of the standard library and is commonly used for such tasks.

3 Sequential code translation

Sequential code specification extraction is out of the scope of this investigation. However, we partially define it to the degree that is necessary to extract specification for message passing. Here we present a basic outline of our translation method, albeit incomplete. It is based on an idea developed in earlier work [6].

Since Elixir is a functional programming language, it is convenient to translate sequential code in units of functions. Therefore, each function is expected to be translated into a separate TLA$^+$ module. In Elixir it is possible to give several definitions for the same function, which would be differentiated by passed arguments – during runtime, the first definition, where arguments match parameter types, is executed. In general, it would be more widely applicable to have such pattern matching done inside the function module, however, for our purposes, it was sufficient to treat such definitions as separate functions.

We treat Elixir functions as a series of expressions that are executed one after another. We expect sequential code specification to reflect this – generated specification should consist of a series of operators each of which is a translation of an expression in Elixir. These expressions should be deterministic, that is, given the current process state, they should produce the next process state. For example, given the following Elixir function:

```
def send(n) do
    other_function(n + 1)
end
```

it could be translated as a set of TLA$^+$ operators shown in Listing 1.

$$\begin{aligned}
&line1(proc) \triangleq \\
&\quad P!call(proc, \text{"other\_function"}, \langle P!arg(proc, 1) + 1\rangle) \\
&line2(proc) \triangleq \\
&\quad P!return\big(proc, P!return\_value(proc)\big)
\end{aligned}$$

Listing 1. Example function expression translations.

In the example above, function body, consisting of a single expression is translated as two separate expressions, *line1* and *line2*. The first one represents the function call together with incrementing its parameter by one while the second one returns the result of the previous function call to the caller. As is evident by this example, not all Elixir expressions are represented as separate expressions in the translation (e.g. parameter increment), nor each operator in translated specification is explicitly reflected in the source code (e.g. function return).

We make use of our Process TLA$^+$ module, which provides operators to access and control the process state. They allow to abstract away the details of common actions away from function modules, making them simpler to translate automatically. Like function expression operators, they are also completely deterministic. This module is included locally in each function module with INSTANCE TLA$^+$ command, as shown in Listing 2. INSTANCE command applied as shown includes all the identifiers of the Process module under the namespace *P*, with Process module constant *Processes* replaced with *Processes* identifier from the current module [7].

$$\mathrm{LOCAL}\ P \triangleq \mathrm{INSTANCE}\ Process\ \mathrm{WITH}\ Processes \leftarrow Processes$$

Listing 2. Process module inclusion in function modules.

Function expression operators are meant to be local to the function module. The rest of the generated specification uses *line_enabled* and *line_action* operators. *line_enabled* operator is meant to check if some process is supposed to execute any expression in the current function module. Typically, it should delegate to the Process module operator of the same name. Similarly, the *line_action* operator is given the current process state and a line to execute on that process state and delegates to a correct expression in the module.

Elixir GenServer module function calls are not translated as regular function calls. Instead, we define TLA$^+$ GenServer module, which operators serve as direct equivalents.

4 Specification generation for the entire program

The entire distributed system specification is generated from a template, gaps in which are filled in with source code parts translated into TLA$^+$. This template defines a general execution model for the entire distributed system and handles message deliveries between the processes.

The state of the entire system is split between three variables: *procState*, *sysState*, and *messageQueue*. The last of these, *messageQueue*, is a set of all messages which still have not been received while others store the state of the system itself as it is known for each process. *procState* contains mostly the values used in specification parts that describe the sequential code execution, e.g. function modules. For example, the value of the *procState* variable determines which function expression should be executed on any given process. Meanwhile, *sysState* contains values required for distributed system specification, e.g. what message is currently being processed. *sysState* contains part of the internal GenServer Elixir module functions state. Such separation increases the modularity of the whole method and simplifies the model-checking of any part of sequential code separately from the rest of the generated specification.

Communication between the processes is modelled by a combination of actions, some of which are generated from source code, while others are predefined. Message-receiving actions are generated from GenServer module callback functions handle_cast and handle_call headers. Listing 3 shows how the following GenServer handler function header is translated:

```
def handle_cast({:client, num}, state)
```

The main purpose of the formula in Listing 3 is to match the message in the *messageQueue* and call the respective message handling function with the actual message and current process state as parameters. The actual functionality of the message handler function is to be specified by the function module, the same as for any other sequential code.

```
$handler1 \triangleq$
    $\exists m \in messageQueue, t \in Processes$
        $\land\, m.to = t$
        $\land\, m.msg[1] = \text{"CLIENT"}$
        $\land\, P!waiting(procState[t])$
        $\land\, procState' = upd\_proc\_state(t,$
            $P!call($
                $P!to\_finished(procState[t]),$
                $handle\_cast!name,$
                $\langle m.msg,\ sysState[t].state \rangle ))$
        $\land\, messageQueue' = M!drop(messageQueue, m)$
        $\land\, sysState' = upd\_sys\_state\big(t, S!set\_reply\_to(sysState[t], m)\big)$
        $\land\, \text{UNCHANGED}\ nextMsgId$
```

Listing 3. Message receiving action example.

Other actions related to message passing are there to ensure the system state is updated as expected after the received message is handled and to take care of synchronous communication. *handler_finished* action does both jobs simultaneously – it updates the system state after the handler finishes and sends out the response message, which may be returned by the handler function. The other two actions, *waiting_responses* and *deliver_responses* are there to correctly translate GenServer multicall function call which sends the same message to several recipients and waits for their responses. We do not provide definitions for these actions here due to space constraints; definitions can be found in the code repository[1].

Sequential code execution is specified by *function_lines* action. It is defined as a disjunction of formulas of the structure shown in Listing 4. The entire disjunction is also existentially quantified to select any process, which allows to model-check different expression execution orderings for a group of processes. *fn_line* operator is displayed in Listing 5. If some function expression can be executed, it updates the process state, sends out all produced messages and starts waiting for replies to the synchronous messages sent.

```
$\exists l \in function!lines:$
    LET
        $line\_enabled \triangleq function!line\_enabled(procState[p],$
        $line\_result \triangleq function!line\_action(procState[p], l)$
    IN
        $fn\_line(p, line\_enabled, line\_result)$
```

Listing 4. Structure of function module expression execution block.

```
$fn\_line(process, line\_enabled, line\_result) \triangleq$
    LET
        $becomes\_blocked \triangleq P!blocked(line\_result)$
        $complete\_messages \triangleq M!full\_msgs(line\_result.sent\_msgs)$
    IN
        $\wedge\ line\_enabled$
        $\wedge\ procState' = upd\_proc\_state(process, line\_result)$
        $\wedge\ messageQueue' = M!bulk\_send(messageQueue, complete\_messages)$
        $\wedge\ nextMsgId' = nextMsgId + Cardinality(complete\_messages)$
        $\wedge$ IF $becomes\_blocked$ THEN
            $sysState' = set\_wait\_replies\_for(process, complete\_messages)$
          ELSE
            UNCHANGED $sysState$
```

Listing 5. *fn_line* operator definition.

[1] https://github.com/mr-frying-pan/master

In all listings provided in this section, we use operators from modules referred to as *M* and *S*. These names stand for Messaging and System TLA$^+$ modules, respectively. Similarly to the Process module described in Section 3 these modules provide operators for their respective areas – message passing and system state modifications. They are included in the specification in the same way as the Process module – using INSTANCE command.

5 Work in progress

Experiment to verify the applicability of the developed method to a realistic algorithm is currently in progress. We have generated a specification for our implementation of Bracha reliable broadcast [8]. We attempt to show that the generated specification is a refinement of an abstract Bracha reliable broadcast specification. Abstract specification of Bracha reliable broadcast, our Elixir implementation and generated specification for it are available in the source code repository[2].

Message-passing part of the specification was generated according to the proposed method. To perform model-checking, sequential code specification is also needed. Since sequential code generation is outside the scope of this investigation, it was written manually. Despite that, manually written function modules retain the required operators so that they can be used in generated specification with minimal changes to it.

We try to show the refinement with model-checking, by showing that abstract specification holds as a property when model-checking generated specification. So far, an initial refinement mapping has been defined; however, the correctness of the mapping is yet to be shown, and we continue tuning the refinement.

6 Conclusions

The developed translation method is modular, different modules encapsulate their respective areas well. If necessary, it is possible to prove properties for any module separately, for both predefined modules and

[2] Repository can be found in https://github.com/mr-frying-pan/master.
Abstract specification is in gen_spec/tla/BrachaRBC.tla.
Our Elixir implementation is in bracha/lib/bracha.ex.
Main generated specification file is gen_spec/tla/bracha.tla.

sequential code modules. We attempt to limit the state explosion by having completely deterministic operators where it is possible to have them, making the number of states dependent on initial inputs.

Future work in the area is needed to further limit state explosion since currently there are a lot of orderings sequential expressions could be executed in, in addition to the message delivery orderings.

Also, more work is needed to obtain a fully functional specification generator. Currently, we generate only overall specification, without the function modules while the bulk of functionality for some algorithm often would be implemented as sequential operations. The sequential code generator is currently being developed and will have to be incorporated into the existing one once it is finished.

Synchronous communication between processes also requires future work, especially the specification of timeouts. It is possible to add timeouts into our specification, but it would require handling process failures and errors.

References

[1] L. Lamport, J. Matthews, M. Tuttle and Y. Yu, “Specifying and verifying systems with TLA+,” in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002.

[2] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst and T. Anderson, “Verdi: A Framework for Implementing and formally verifying distributed systems,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[3] S. Juric, Elixir in Action, Manning, 2019.

[4] D. Thomas, Programming Elixir ≥ 1.6: Functional |> Concurrent |> Pragmatic |> Fun, Pragmatic Bookshelf,, 2018.

[5] T. Arts, C. B. Earle and J. J. S. Penas, “Translating Erlang to μCRL,” in *Proceedings. Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004*, 2004.

[6] D. Bražėnas, *Extracting TLA+ Specifications out of Elixir Programs,* Vilnius: Vilnius University, 2023.

[7] L. Lamport, Specifying systems: the TLA+ language and tools for hardware and software engineers, Addison-Wesley, 2002.

[8] G. Bracha, “Asynchronous Byzantine agreement protocols,” *Information and Computation,* vol. 2, no. 75, p. 130–143, 1987.