VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

Extracting TLA+ specifications out of a program for a BEAM virtual machine

TLA+ specifikacijų išskyrimas iš programinio kodo BEAM virtualiai mašinai

Master thesis

Author: Andrius Maliuginas (signature)

Supervisor: Dr. Karolis Petrauskas (signature)

Reviewer: Dr. Julius Andrikonis (signature)

Vilnius – 2024

Contents

Introduction 3
Context and relevance 3
Aim, objectives and expected results 4
Aim 4
Objectives 4
Expected results 5
1. Elixir 6
1.1. Data representation 6
1.2. Pattern matching 7
1.3. Abstract Syntax Tree (AST) 8
1.4. Process model & communication 9
1.5. Erlang/Elixir design patterns 10
1.5.1. Generic server pattern 11
2. TLA+ 12
2.1. Temporal Logic of Actions 12
2.2. TLA+ specification structure 13
3. Related works in the area 14
3.1. Sequential Elixir code translation into PlusCal 14
3.2. Translating Erlang to μCRL 15
3.3. McErlang 17
3.4. C to TLA+and Java to TLA+ 19
3.5. TLA+ to Elixir 19
3.6. Other approaches 20
3.6.1. Haskell to Coq and to Isabelle/HOL 20
3.6.2. Verdi 20
3.6.3. Bandera 21
4. Extracting specification of interprocess communication 22
4.1. Solution overview 22
4.2. Process and communication model 25
4.2.1. Process state structure 26
4.2.2. System state structure 27
4.2.3. Message structure and queue 28
4.3. Data matching in function definitions 28
4.4. Sequential code specification model 29
4.4.1. Function module structure 30
4.4.2. Function context 31
4.5. Helper TLA+ modules 31
4.5.1. *Process* module 31
4.5.2. *System* module 32
4.5.3. *Messaging* module 32
4.5.4. *GenServer* module 33
4.6. Generating specification for whole system 33
4.6.1. General structure 34
4.6.2. Message receiving actions 34
4.6.3. Other actions 36
4.7. Generator program 39
4.8. Capabilities and limitations 41
5. Verification 42

Results and conclusions 43
References 44

Introduction

Context and relevance

A distributed system is a set of independent nodes connected by a network that cooperate to solve a problem that cannot be solved individually [KS11]. In the context of computing, nodes typically are processes which are communicating through message passing (possibly through network) or some shared memory. Designing algorithms for such systems can be difficult and error prone [LMT+02]. This complexity arises from the nature of distributed systems – when there are several independent processes, each of them can fail independently and it may be hard to tell which of the processes fails first – executions in each process are happening concurrently, each of them can be at a different point of the program. Lack of single global time due to clock drift in different machines also complicates algorithm design.

Formal specification of such algorithms is supposed to help to overcome these challenges. It is a way to describe algorithms formally using mathematical formulas. Having specification allows to check algorithms for correctness and to find edge cases and non-obvious mistakes, which is especially useful for distributed algorithms due to their inherent complexity [Lam99].

However, it is not enough to have a specification for an algorithm, it also needs to be implemented to be useful. Ideally, the implementation would be a more detailed version of the formal specification. It is possible to rely on manual source code analysis to ensure that implementation conforms to the specification but this approach is slow and error prone. Another way to ensure the same is to generate source code from the formal specification. This requires having enough details about the system in it but those are very rarely included and require a lot of expertise from whoever writes the specification.

In this thesis we attempt to develop a way to show that a some implementation of the algorithm really does implement its formal specification by finding a way to map program source code into formal specification. In particular, we choose Elixir programming language for the source code and TLA+ language for specifications.

Elixir is a programming language for BEAM virtual machine. Programs for BEAM virtual machine are often distributed: they consist of one or more processes which may run on one or more machines. It is possible to have a program with one process on one machine but it is structurally similar to the distributed version of the same program. Each process is fully sequential and no data is shared between them. Communication is done only through message passing. Processes are also able to spawn other processes and form hierarchical trees but for the purpose of this thesis we treat the distributed system as a simple collection of processes where any process could potentialy communicate with any other [dcon].

For inter-process communication Elixir provides two built-in functions: `send` and `receive`. `send` sends the the given Elixir term to one given process. `receive` construct blocks the execution. In practice, however, GenServer module is preferred as it provides a more convenient way to do the same with less boilerplate code. We base our research on these GenServer module function calls and response handlers on the assumption that this module is used often enough for this to be useful.

The messages sent between the processes are regular Elixir terms. Typically, to distinguish between different kinds of messages, tagged tuples (tuples where first element is an atom) or structures are used.

To develop the mapping into TLA+ it is planned to make use of Elixir abstract syntax tree. Abstract syntax tree (AST) is a tree-like representation of the source code produced by parser. It allows software to access all source code elements (such as variables, functions, operators, etc.). In this thesis we use the AST extracted from source code, as opposed to AST extracted from compiled bytecode, because it has the advantage that it preserves annotations (and even comments, if needed) which helps to map code to abstract specification.

TLA+ is a systems specification language. It uses formulas of temporal logic of actions [LMT+02] to specify the behaviour of the system. Resulting specifications properties can be checked using TLC – a model checker provided together with TLA+ or proven with the help of TLAPS (TLA+ Proof System). It is useful for specifying distributed systems since their components operate independently from one another [LMT+02].

Aim, objectives and expected results

Aim

The aim of master thesis is to develop a way to generate TLA+ specification from Elixir source code for a distributed system.

Objectives

1. Define a way to map Elixir code to TLA+ specifications and prove its correctness. We assume that sequential code (the code which constitutes function bodies) has already been mapped and we have preconditions and postconditions of each function defined. Instead, we focus on connecting the sequential parts of the system and modelling communication between the nodes in a distributed system.

2. If necessary, define *constraints* on the program source code *for generating TLA+ specification*. These could be certain restrictions on code structure or language features which should not be used[1]. Ideally, there would be no such restrictions as this would make the mapping more widely applicable.

3. Define *requirements* for the source code so that it would be possible *to prove refinement* from original specification into generated specification. These requirements could be different from the ones which apply only to generate the specification. They can be similar as to those, which are defined for specification generation.

4. Develop software to generate TLA+ specification from Elixir source code. Given the source code it should be able to connect the specifications of functions into complete algorithm.

[1] See [AE] for restrictions on Ada language features to provide static verification

Expected results

1. Mapping from Elixir to TLA^+ and its correctness proof.
2. Source code constraints for generating TLA^+ specification.
3. Requirements for the source code for proving refinement of a more abstract specification.
4. Software to generate TLA^+ specifications.

Correctness of mapping will be shown by modelchecking.

1. Elixir

Elixir [Jur19] is a functional programming language for BEAM virtual machine. It is a second language for this virtual machine, first being Erlang. Erlang and BEAM were developed by Ericsson to serve their needs as telecommunications company – little to no downtime, distributedness, resiliency and others. As distributed systems became more widespread and in demand, Erlang features looked increasingly attractive. Elixir was developed as an alternative to Erlang. It is fully compatible with Erlang – it compiles to a common format, can run Erlang functions and most of Elixir constructs map directly into Erlang's. However, since Elixir is a more modern language, it comes with a more modern syntax, syntactic sugar and additional features which reduce code duplication [Tho18].

In the following sections we overview the syntax and features of Elixir which are important when generating specification from the source code of distributed system.

1.1. Data representation

Distributed systems communicate by message passing. While transforming source code to specification, it would be advantageous to know what kind of data is exchanged between the parts of distributed system. In Elixir messages sent between the processes can be of any data type, therefore here present a short overview of data types and their representations in the source code.

Numbers. Numbers in Elixir come in two kinds: integers and floating point numbers. Besides the fact that integers can be arbitrarily large, numbers behave the same as in other programming languages. Example representations: `53`, `1.2`, `15_000_000`.

Atoms. Atoms are named constants. They are used when the exact value of the constant is not important, just the meaning of it, similar to enumerations in C++ or Java. Boolean values `true` and `false` are atoms, as well as a special value `nil` (the equivalent of `null` in other languages). Example representation: `:some_atom`.

Tuples. Tuples are a fixed length ordered group of elements. Members of a tuple can be of any other type. Example representation: `{ true, 123 }`.

Binaries and strings. Elixir lacks a dedicated string type. Instead binaries – sequences of bytes, are used to store them. Erlang strings are called character lists, and are lists of integers which are character codes. Character lists are typically used when interacting with Erlang code. Example representations: `<<1, 2, 3, 4>>`, `"This is a string"`, `[65, 66, 67]`, `'ABC'`.

Lists. Lists are dynamically sized ordered collections of elements. They are defined recursively as head element in from of the rest of the list. Example representations: `[ 1, true, { 11, "123" } ]`, `[ head | tail ]`.

Maps. Maps are key-value unordered collections. Maps with atom keys are treated slightly differently, they have their own value access and update operators: `some_map.atom_key` to get the value and `%{ some_map | atom_key: "new value" }` to update. Example representation: `%{ one_key => one_value, other_key: other_value }`.

Structs. Structs by their functionality are identical to maps, with additional feature that atom key set is limited to those present in the definition. Example representation: `%State{ field: 0, other_field: "no" }`.

Functions and anonymous functions. Since Elixir is a functional language, functions are also a type of value. There are two kinds of functions – named and anonymous. Named functions are defined in modules and invoked by their name. They can have guards – statements which constrain the allowed values for the parameters. Often, there are multiple definitions of the same function, differentiated by different parameter patterns. In such cases, first definition which matches passed parameters is invoked. If there is no matching definition, then error is raised.

Anonymous functions are defined wherever there is a need to use them (usually inside other functions) and are bound to a variable or passed as a parameter to another function. Anonymous functions, unlike named functions, are invoked by adding a dot between parameter list and the name of the variable they are bound to: `result = parameter_fn.(1)`.

Example representations:

Named, with a guard:

```
def identity_fun(x) when is_number(x), do: x
```

Anonymous:

```
fn x -> x end
```

Although values in Elixir have types, variables are dynamic – they are of the their value is. Unlike in Erlang, it is possible to change what value is bound to the variable, that does not modify the original value since all values are immutable.

1.2. Pattern matching

Elixir makes heavy use of pattern matching. Since all data is immutable, there is no assignment operator and therefore = is actually a match operator. It allows not only to bind the value to the variable (e.g. `x = 12`) or compare two values but also to expect a certain structure in the value returned by the function and extract values from inside that structure. For example, `File.read/1` function to returns `{ :ok, contents }` in case of success and `{ :error, reason }` in case of failure. That is commonly matched as `{ :ok, contents } = File.read("file.txt")` to only get the contents of the file and have `MatchError` error raised in case error occurs.

Pattern matching can also be used for function parameters. The rules for matching values to parameters are the same as when using the match operator (=). The result of using pattern matching in this way is that the function definitions are partial and cover only a subset of the domain. If

invocation parameters do not match the patterns in the function definition, another definition is looked for. If no definitions match the passed parameters then the error is raised. It is possible to have a catch-all definion at the end to avoid the error but it is not required.

1.3. Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST) is a hierarchical representation of the parsed programming language source code usually (and most usefully) expressed using data structures of the same or some other programming language. Elixir provides native way to access the AST of its code which also includes the annotations and optionally comments [Tho18].

Figure 1 shows a simple list visitation function together with its AST. Underscores replace the metadata elements in tuples (which contain the line number of the source code where that particular syntax element is located). This was done for brevity.

Source code:

```
defp visit(lst) when is_list(lst) do
  for el <- lst do
    somefn(el + 12, :atom)
  end
end
```

AST:

```
1  {:defp, _, [
2    {:when, _, [
3      {:visit, _, [{:lst, _, nil}]},
4      {:is_list, _, [{:lst, _, nil}]}
5    ]},
6    [do:
7      {:for, _, [
8        {:<-, _, [{:el, _, nil}, {:lst, _, nil}]},
9        [do:
10         {:somefn, _, [
11           {:+, _, [{:el, _, nil}, 12]},
12           :atom
13         ]}
14       ]
15     ]}
16   ]
17 ]}
```

Figure 1. Elixir AST of a simple list visit function.

In the Elixir AST a lot of different statements are represented as a 3-member tuple: `{ :f_name, _, [params] }`. This structure is used not only for function calls (e.g. marked in red in figure 1 `somefn(el + 12, :atom)` becomes a tuple on lines 10-13) but also such constructs as `defp` which is a keyword for defining module private functions and `when` (marked in green) which defines a

parameter guard. Operators are also translated into this structure (e.g. `<-` operator, marked in blue).

Notably, `do … end` blocks do not follow the same 3-member tuple structure. They are transformed instead into a list based structure `[do: body]` where body is either a single statement (as is the case in the example) or a block of statements.

Integers and atoms are represented as themselves, as can be seen on lines 11 and 12 respectively. The same is true for floating point numbers.

1.4. Process model & communication

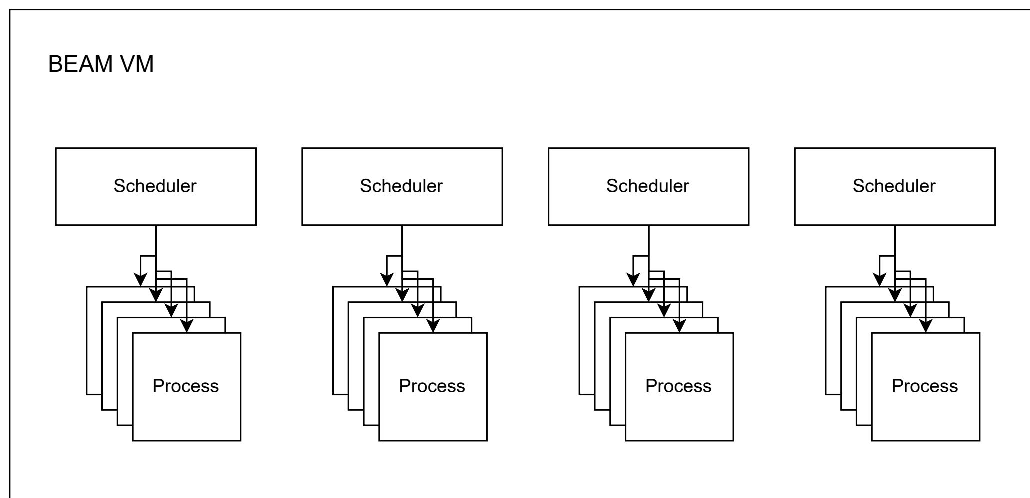

Figure 2. BEAM virtual machine process model [Jur19].

Elixir programs run on BEAM virtual machine (BEAM VM). It is based on *actor* model [Tho18] meaning that programs consist of one or more independent *actors* which are communicating with one another. In Elixir these *actors* are called processes.

Figure 2 shows process organisation within a running BEAM VM. One running BEAM VM is a single OS process and is called a node. For each CPU available on the physical machine it has a separate process scheduler which runs on a separate OS thread. Schedulers, in turn, manage the processes assigned to them which execute the application code. New processes are created using `spawn` function. Each process is uniquelly identified by its Pid.

Communication. Processes are isolated from each other and do not have any shared memory. They communicate by putting messages into their message queues. Calling `send` function puts the message into the message queue of the required process and executing `receive` statement takes the first message in the queue or, if the queue is empty, waits for some message to arrive. `receive` pattern matches on the message contents so if there are messages in the queue but they do not match any given patterns, `receive` continues waiting. All messages are sent asynchronously, if there is a need to respond, sender process id should be sent in the message so that receiving process would know where to send the response once it receives the request.

Addressing. In order to send messages, it is necessary to know which process to address them to. In Elixir process Pid acts as a global address, the problem remains then where to get the address of other processes. Parent process knows child process Pid since it is returned by `spawn` function, and can get its own Pid by calling `self`. But using only Pids becomes very confusing as number of processes grows. Then it is much more convenient to make use of process registries which associate names with the Pids. Several registries are available – there is a node local process registry, which tracks the names of the processes within current node, there is `:global` registry, which tracks processes of all connected nodes and finally programmer could implement a custom process registry.

Linking and monitoring. BEAM processes are sequential – there is no concurrency within a process. If there is a need to perform some calculations concurrently, a new process should be launched. By default death of one process does not affect other processes but this can be changed by linking the spawning (parent) and spawned (child) processes together by using function `spawn_link`. By default, if any linked processes crashes it also brings down linked process. That can be changed by calling `Process.flag(:trap_exit, true)` which makes the ends of linked process executions appear as messages in the process message queue. These messages can be read using `receive` and contain the reason of the linked process exit, allowing to distinguish between the normal exits and crashes. Structure of such message is `{:EXIT, pid, reason}`. Here `pid` is Pid of the process which has stopped execution and `reason` could be any Elixir term. In case of normal termination `reason` is `:normal`.

It is possible to setup a one-directional link between the processes where only one process is notified if other exits. Such links are called monitors. When monitored process exits, a message of the following structure is received: `{:DOWN, monitor_ref, :process, from_pid, reason}`. Here `monitor_ref` is a reference of the monitor, `from_pid` is Pid of the process which has finished executing and `reason` is the reason of exit, the same as in regular links.

Behaviour on different nodes. It is possible to connect to BEAM VM instance, possibly running on a different physical machine. Although syntax of the the communication between the processes and their linking and monitoring are the same, guarantees provided by the runtime are different. In a single node message deliveries are instant, in the same order they were sent in. In distributed environment order only guaranteed between the pairs of processes and only assuming that receiving process stays available [FS07]. The same rules apply to the process exit and monitoring messages.

1.5. Erlang/Elixir design patterns

There are a couple of design patterns used in Elixir code which are very common in BEAM programming language code. They provide a ready made solutions to a common problems, present in many different systems.

1.5.1. Generic server pattern

When developing any kind of system it is often needed to have a server which keeps state, continuously waits for requests and once received handles them in a way that may modify the state. A very simplified generic server implementation is shown in figure 3. It is a function which receives a request, calls request handler and calls itself again with the possibly modified state as a parameter. It would be tedious and error prone to implement this pattern in every program which needs it but thankfully Elixir provides a standard implementation in `GenServer` module.

```
defp loop(callback_module, current_state) do
  receive do
    {request, caller} ->
      {response, new_state} =
        callback_module.handle_call(
          request,
          current_state
        )

      send(caller, {:response, response})

      loop(callback_module, new_state)
    _ -> ...
  end
end
```

Figure 3. Simple generic server implementation [Jur19].

`GenServer` module provides two main functions for sending messages to other processes: `call` and `cast`. Both functions send request to specified process, the difference being that `call` is a synchronous request while `cast` is asynchronous, i.e. `call` waits and returns the response while `cast` returns immediately. There are also convenience functions `multi_call` and `abcast` which are meant to send requests to multiple processes.

`GenServer` module requires programmer to define handlers for `calls` and `casts`. These are `handle_call` and `handle_cast` functions respectively.

`GenServer` module also allows to define callback functions `handle_continue` and `handle_info`. `handle_continue` is used to update process state in several steps (in case of e.g. long request processing), as it is invoked if some `handle_*` function produces respective return value. In turn, it itself can produce similar return value, allowing for unlimited amount of process state updates while processing the request. `handle_info` processes all other messages not handled by other `handle_*` functions.

2. TLA+

TLA+ [LMT+02] is a distributed systems specification language. As the name implies, it is based on TLA – Temporal Logic of Actions [Lam94]. In the following sections we shortly describe the underlying formal logic and specification structure.

2.1. Temporal Logic of Actions

Temporal Logic of Actions (TLA) is a quantified modal logic extended with a concept of an action. All statements of the regular quantified predicate logic (i.e. ”simple logic”) are also valid statements in TLA.

TLA models an algorithm as a set of states with possible transitions between them. The states are defined by the particular assignment to the variables of the algorithm. These values can be of any type, but typically they are numbers (12, -3), strings (”abc”) or sets of values.

Transition between the states is defined as an action – a predicate on a pair of states *p* and *n* which is true when there exists a transition from *p* to *n*. Since an action is a predicate, it describes a particular operation on the variables, not a particular transition between two particular states. As a result, the states *p* and *n* are not specified, instead prime ($'$) operator is used to distinguish which variable values belong to *n*. For example, if during the transition from some previous state *p* to some following state *n* algorithm variable *v* increases its value then the action describing this transition will look like this: $v' = v + 1$. v' here stands for the value of the variable *v* in the next state *n* and *v* for the value in the previous state *p*.

When it possible for a system to move from one state to the other the action describing this transition is called *enabled*. From any particular state it can be possibe to transtion to several other states. In that case all actions are called *enabled*.

Temporal nature of TLA is useful to describe the properties of the whole execution (i.e. a sequence of states) of the algorithm. Temporal formulas are constructed using operators $\Box$ (always), $\Diamond$ (eventually) and $\leadsto$ (leads to). Only always operator ($\Box$) is required since others are defined through it: $\Diamond F \triangleq \neg\Box\neg F$ and $F \leadsto G \triangleq \Box(F \rightarrow \Diamond G)$.

An action when no variable changes is called a stuttering step. Such steps are permitted in TLA, they allow to specify algorithms on different levels of abstraction. However, presence of stuttering steps also allows for executions where nothing ever changes. To exclude these, two fairness properties are defined – weak and strong.

Weak fairness of action $\mathcal{A}$ is defined as:

$$\mathrm{WF}_f(\mathcal{A}) \triangleq (\Box\Diamond\neg Enabled\langle\mathcal{A}\rangle_f) \vee (\Box\Diamond\langle\mathcal{A}\rangle_f)$$

Here, $\langle\mathcal{A}\rangle_f$ means a step $\mathcal{A}$ which changes variables in tuple f. $Enabled$ is a predicate which is true if action is *enabled*. This formula is interpreted as ”$\mathcal{A}$ is infinitely often disabled, or infinitely many $\mathcal{A}$ steps occur” by Lamport in [LMT+02].

Strong fairness of action $\mathcal{A}$ is defined as:

$$\text{SF}_f(\mathcal{A}) \triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \lor (\Diamond\Box\neg Enabled\langle\mathcal{A}\rangle_f)$$

Here the meanings of symbols are the same as above. Lamport provides the following interpretation if this formula: "$\mathcal{A}$ is eventually disabled forever, or infinitely many $\mathcal{A}$ steps occur" [LMT+02].

2.2. TLA+ specification structure

TLA+ specification begins by declaring constants and variables. Constants often act as parameters or as abstract sets of objects, while variables keep system state. Constants need to be provided for modelchecking.

Adding some type information to the variables is common when specifying a distributed system since messages sent between parts of the system also need to be specified and typically they do have some structure, system processes do not usually send arbitrary values to each other. This makes it advantageous to add at least some type information to the variables of the specification, which is typically done with an invariant:

$$TypeOk \triangleq v_1 \in Set_1 \land v_2 \in Set_2 \land \ldots$$

Here v_1, v_2 are variables and Set_1, Set_2 are sets the values of v_1 and v_2 should belong to.

Overall, typical TLA+ specification is expressed by the formula

$$Spec \triangleq Init \land \Box[Next]_{vars} \land \text{WF}_{vars}(Next)$$

Here $Init$ is a formula specifying the initial state. $Next$ is a formula specifying all possible actions in the system, without regard whether they are enabled or not. Often in non-trivial specifications $Next$ is a disjunction of other formulas which describe more specific actions. WF is an operator specifying weak fairness. It can be replaced with SF for strong fairness if usecase requires. $vars$ is a tuple of all the variables.

3. Related works in the area

In this section we present other works related with specification generation from source code. First we present those that are more immediately relevant, such as translating to and from Elixir (or Erlang) and TLA+, while the last section contains an overview of the works less useful for our purposes here but nonetheless making important contributions to the field.

3.1. Sequential Elixir code translation into PlusCal

This thesis is based on previous work by D. Bražėnas [Bra23]. In his master thesis D. Bražėnas describes a method of translating a subset of sequential Elixir code into PlusCal specification [Lam] from which then TLA+ specification is generated, using standard TLA+ tools.

Author defines rules how to translate different parts of sequential code. We will not reproduce all the rules here, only will overview those which are useful for our work.

We borrow from [Bra23] rules for translating Elixir data values. These rules are displayed in figure 4. Rules ATOM and STRING translate Elixir atoms and strings respectively into PlusCal strings. Similarly, tuples and lists are both translated into sequences, as rules TUPLE-1 and TUPLE-2 show. Booleans and *nil* are translated as themselves.

$$\frac{\vdash \text{is_atom}(a)}{\vdash "a"} \text{ (ATOM)} \qquad \frac{\vdash \text{is_boolean}(a) \land a == \texttt{true}}{\vdash \texttt{TRUE}} \text{ (BOOL-TRUE)}$$

$$\frac{\vdash \text{is_binary}(a)}{\vdash "a"} \text{ (STRING)} \qquad \frac{\vdash \text{is_boolean}(a) \land a == \texttt{false}}{\vdash \texttt{FALSE}} \text{ (BOOL-FALSE)}$$

$$\frac{\vdash \text{is_number}(a)}{\vdash a} \text{ (NUM)} \qquad \frac{!is_single_line \vdash \{:\{\},_,a\}}{\vdash << a >>} \text{ (TUPLE-1)}$$

$$\frac{\vdash \text{nil}}{\text{NULL}} \text{ (NIL)} \qquad \frac{!is_single_line \vdash \text{is_list}(a)}{\vdash << a >>} \text{ (TUPLE-2)}$$

Figure 4. Data translation rules used in this thesis as defined by [Bra23].

Function return value is modelled as assigning a value to specific variable in specification. We take similar approach when defining our sequential code translation.

Method described in [Bra23] translates a single function body. Translation of pattern matching on passed function arguments is not defined. Translation for function calls is also limited. Although a rule for anonymous funcition call is provided, it is applicable only in specific context – when returned value is to be used immediately, not assigned to a variable, e.g. as an `if` statement condition. Generic function call translation is not defined, which allows to avoid dealing with parameter pattern matching. It is assumed that called functions (also anonymous ones) would be defined as a PlusCal operator, procedure or macro, which would allow to use them as such in the specification.

Program written for the [Bra23] is capable of producing a TLA+ specification given Elixir source code file with functions, for which specification generation is desired, marked with annotation. The same program also runs PlusCal translation into TLA+ and starts modelchecking.

3.2. Translating Erlang to µCRL

T. Arts, C. B. Earle and J. J. Sánchez Penas have developed a tool to generate µCRL [GR01] specification from Erlang code. Authors present the tool created for this purpose – etomcrl [AEP04].

etomcrl generates µCRL which is a process algebra. It means that systems are specified as collections of processes which do defined atomic actions. Actions can be composed. ($x \cdot y$ is a sequential composition, meaning that action x happens before y while $x + y$ is a alternative composition, meaning that only one of x or y is executed. Composition is possible for more than one action. In this paper sum(param: Type, act) construct is widely used and is translated into an alternative compostion of action act with the parameter param for each of the members of Type:

$$\sum_{param \in Type} act(param)$$

General approach taken by etomcrl is to represent the program as a collection of processes which communicate with each other by reading and writing messages to and from common buffer. Decision was taken to treat sending and receiving process identifiers as parts of the message instead of having separate channels for each pair of processes. Distributed processes are considered identical to concurrent ones, despite different communication semantics.

etomcrl relies heavily on design patterns used in Erlang programs. One such pattern is the *generic server*. To make use of it authors specify the call, cast and reply actions and use them from other actions when message needs to be sent. Functions handling receipt of messages use reply action to send the response.

Erlang functions can have several definitions which are selected by pattern matching the input parameters. This is not possible to do directly in µCRL so instead first Erlang code is transformed into a nested if statement and variables from the patterns are replaced into expressions involving destructor functions (such as list head function hd). See figure 5 for an example of such translation. Afterwards such function is translated as any other function would.

Initial code:

```
loop(X,[]) ->
  s(done), loop(X,X);

loop(X,[Head|Tail]) ->
  s(Head), loop(X,Tail).
```

Translated code:

```
loop(X,Arg1) ->
  if
    nil == Arg1 ->
      s(done), loop(X,X);
    is_list(Arg1) ->
      s(hd(Arg1)),
      loop(X,tl(Arg1))
    end.
```

Figure 5. etomcrl Erlang to Erlang translation of multiple function definitions [AEP04].

For message handling functions it is not enough to just translate them. They need to be somehow invoked on message receipt. In order to achieve that authors put the message handling actions

into the top level of message handling process. See figure 6 for the translation example.

Initial code:

```
handle_call({request,a},Client,{A,B}) ->
  {reply,A,{false,B}};
handle_call({request,b},Client,{A,B}) ->
  {reply,B,{A,false}};
handle_call({release,R},Client,State) ->
  {reply,ack,update(R,State)}.
```

Generated specification fragment:

```
server(Self:Term,State:Term) =
  sum(Client: Term,
    handle_call(Self,tuple(request,a),Client).
    reply(Client,element(1,State),Self).
    server(Self,tuple(false,element(2,State))))
  +
  sum(Client: Term,
    handle_call(Self,tuple(request,b),Client).
    reply(Client,element(2,State),Self).
    server(Self,tuple(element(1,State),false)))
  +
  sum(R: Term,
    sum(Client: Term,
      handle_call(Self,tuple(release,R),Client).
      reply(Client,ack,Self).
      server(Self,update(R,State))))
```

Figure 6. `etomcrl` Erlang to μCRL translation of message handling functions [AEP04].

Higher order functions present significant problem for μCRL as it is a first order language. Since usually higher order functions used were the well known ones, like `map` authors were able to define a source to source transformations for the special cases encountered which created first order equivalents for them.

Erlang modules are also handled through source to source translation. Function calls are translated into fully qualified calls (`module:function`) and translated as regular actions in μCRL. Certain standard library modules are translated only once and are reused everywhere they are invoked.

Another pattern used is *supervision tree*. `etomcrl` is able to scan and recognize the tree structure and get the worker processes but does not handle process creation outside of supervision tree. Since μCRL tooling at the time of writing did not support creation of new processes all processes are considered created at the start of execution. Also, only successful executions are considered, fault tolerance is not translated at all – it is an object of further research.

3.3. McErlang

McErlang [FS07] implements a model checker for Erlang programs. Instead of trying to produce a specification in some dedicated specification language, authors decide to check Erlang source code itself.

Running modelchecker needs three things to be provided by the user: a program to be checked, specification of the program properties, and several settings for the modelchecker. Modelchecker settings are, for example, what kind of property would be checked – safety, liveness or testing, or which abstraction module to use. Abstraction module defines how should program states be abstracted, e.g. by reducing integer variable into boolean with the meaning that its true when integer would be positive.

Using the Erlang semantics described in [CS05; Fre01] authors substitute the standard BEAM scheduler with their own to have access to program state and to be able to check all possible executions. New scheduler does not replace the BEAM scheduler but since all program processes run inside single Erlang process during modelchecking it is replaced from the point of view of the program being checked. Care is taken to preserve scheduler fairness, to minimize differences between real executions and those simulated during modelchecking. This is done by the McErlang compiler, which takes initial program and produces modified Erlang source code ready for modelchecking. Compiler also replaces the interprocess communication constructs (`send` and `receive`) by the ones written for modelchecking.

McErlang models the Erlang program runtime as a set of nodes (corresponding to physical machines) and a message queue containing all messages currently being transferred called *ether*. Each node is defined by the following tuple:

$$\langle name, processes, registered, monitors, node_monitors, links \rangle$$

Here *name* corresponds to the node name, *processes* is a list of processes running on that node, *registered* is a name server which maps node name to its Pid, *monitors*, *node_monitors* and *links* are used for process linking.

Each of the processes in turn is represented by the follow tuple:

$$\langle status, expr, pid, queue, dict, flags \rangle$$

Here *status* shows whether process is running, ready to run, waiting for the message, etc. *expr* is the function which should be executed next, *pid* is the Pid of the process, *queue* is a queue of messages sent to the process which can be read by the process, *dict* is a dictionary of the process ("the equivalent of imperative variables in Erlang" [FS07]) and *flags* – flags indicating process settings, e.g. whether child process death should also kill this process.

Messages between the nodes are tuples of sending process Pid, receiving process Pid and the sent message itself.

While it is straightforward to adapt `send` for this setting (by just putting the message into the *ether*), `receive` proves more challenging. Two types of `receive` are identified: tail recur-

sive and non-tail recursive. In case of tail recursive receive the function containing the receive construct is transformed to return `{ recv, {?MODULE,f_0,[state]} }` value. See the example of this transformation in figure 7. Here `f_0` is a function which contains whatever was inside the `receive` construct. It is invoked by McErlang runtime once message is delivered to the process. Non-tail recursive `receive` is handled with the use of stack. As in tail recursive case function is made to immediately return, except the return value in this case is different: `{letexp,{expr,{module,f,parameters}}}`. Example of this translation is shown in figure 8.

Initial code:

```
server(State) ->
  receive
    {new_state, NewState, Pid} ->
      Pid!{reply,State},
      server(NewState)
    end.
```

Translated code:

```
server(State) ->
  {recv, {?MODULE, f_0, [State]}}.

f_0({new_state, NewState, Pid}, [State]) ->
  {true,
  fun ({new_state, NewState, Pid}, [State]) ->
    evOS:send(Pid,{reply,State}), server(NewState)
  end};
f_0(_, _) -> false.
```

Figure 7. McErlang translation of tail-recursive `receive`s [FS07].

Initial code:

```
server(State) ->
  {ok, NewState} = doRequest(State),
  server(NewState).
```

Translated code:

```
server(State) ->
  {letexp, {doRequest(State), {?MODULE, f_1, []}}}.

f_1({ok,NewState}, []) ->
  server(NewState).
```

Figure 8. McErlang translation of non tail-recursive `receive`s [FS07].

3.4. C to TLA⁺and Java to TLA⁺

Authors of [MLH⁺14] present a tool for generating a TLA⁺ specification of a concurrent C program called C2TLA+. It only handles a subset of C language produced by CIL [NMR⁺02] but the output is a TLA⁺ specification which can be checked with TLC.

[MLH⁺14] models the program as a collection of processes each identified by the unique id and running on a separate thread, i.e. concurrently. Communication between processess is assumed to be through shared memory.

Similarly, authors of [LN11] describe a tool for generating TLA⁺ specification from a Java bytecode. The same program model is used: a program is a collection of threads each with own call stack and shared heap and code spaces.

Verifying a distributed system with these tools is not possible as there is no notion of message passing between the processes, each agent in such system would be treated in isolation. Although in general it is possible to model message passing as shared memory communication, a lot of complexity associated with network communication would be lost (for example, there is no latency in shared memory). Also, adding message passing into generated specification would have to be manual.

3.5. TLA⁺ to Elixir

Authors of [MVK22] are solving an opposite problem than this work, namely they are trying to generate Elixir code from TLA⁺ specification. This approach is widespread and well known [COR⁺01; GKM⁺08] but it was not yet done for TLA⁺. In addition to source code unit tests are generated.

The article treats TLA⁺ specification as a collection of transitions between program states where if certain formula is true for a pair of states (current state and next state) then that transition is valid and can happen. Sometimes several states may follow the current one – this is one of the ways non-determinism is expressed in TLA⁺.

TLA⁺ specification is assumed to be at the highest level just this: $Spec = Init \land \Box[Next]_{vars}$ where *Init* is initial state of the variables and *Next* a disjunction of all possible transitions between the states.

Variables from the specification are stored in a global Elixir variable conveniently called `variables`. Generated program itself consists of a `main()` function which repeatedly calls `next()` which corresponds to the *Next* formula in the specification. `next()` function returns a set of possible next states based on the current state. If there is only one state, then it becomes the next one, otherwise next state is chosen using the Oracle – an unspecified function which given the a set of possible states returns the next one. In this way, the Oracle encapsulates the non-determinism of the specification in one place. Actual implementation of the Oracle is expected to be provided by the programmer.

In TLA⁺ the system is modelled as a unit so there is no notion of different processes. However, when distributed systems are specified, different actions are meant to be happening in different

processes. To generate such code properly, configuration is required to signify which actions in the specification are meant to be done by one process and which ones by the other. Also, all interprocess communication is done by an Oracle. Each node tries to obtain a lock from this Oracle to read the whole state on startup and later to communicate changes to its state. This Oracle can be separate from private process Oracle which only handles non-determinism which does not need any additional information.

3.6. Other approaches

This section contains works which are less useful than those overviewed above but which nonetheless make an important contribution to the field.

3.6.1. Haskell to Coq and to Isabelle/HOL

There has been an effort to improve verification of programs written in Haskell programming language which has resulted in `hs-to-coq` [SBR+18] and `haskabelle` [Haf10] tools.

`hs-to-coq` generates Coq [Tea22] specification for total Haskell (no partial function definitions allowed) programs. It relies heavily on syntactic and semantic similarities of Haskell and Coq, which allows for a very straightforward translation of many Haskell features (e.g. algebraic data types, function applications, basic pattern matching). `hs-to-coq` first parses the source code obtaining the abstract syntax tree, then renames the identifiers which need to be renamed and then generates Coq code. Notably, `hs-to-coq` does not desugar the code during parsing to keep generated Coq as similar to original Haskell as possible.

`haskabelle` also relies on the similarity between Isabelle language and Haskell. There is no restriction that Haskell source code should be total but in cases when the tool is not capable of translating some part of the code, manual user input is required to fill in the gaps. Also, places where such gaps are allowed to be generated need to be marked with a special comment in the source code, otherwise translation fails. Syntax *adaptations* can be defined to circumvent the failures but it is not clear how they are specified and applied.

Both tools only consider a single program, distributed programs and communication between them are not considered at all.

3.6.2. Verdi

Verdi [WWP+15] is another example of code to specification approach. It is a framework for implementing and verifying distributed systems. It allows specifying and implementing the system in Coq [Tea22], choosing certain fault model the system is supposed to handle and prove that it does that, and obtaining OCaml [MMH13] code which can be executed on the servers directly. Verdi proofs are composable, i.e. application logic and fault tolerance can be proven correct separately and later it can be shown that together they remain correct. This has a side effect that a programmer can implement the system in a simpler setting (e.g. with reliable network and no node failures) and later switch to a more complex one. To facilitate this transition *verified system transformers*, which

modify the model and the proof that it still conforms to the initial specification, are also a part of the framework.

System transformers are of two basic types – network transformers and crash transformers. Each of them produces system which is able to handle different network and node conditions, e.g. duplicated messages and node failures. In the final program they are implemented as wrappers over functions produced by initial specification.

3.6.3. Bandera

Bandera [CDH+00] is tool for modelchecking Java programs. It is capable of generating any kind of specification language as long as translation module between its internal representation (called BIR) and desired specification language is provided. As published, it was capable of producing only Promela models.

Bandera works by first transforming Java code to a first intermediate language called Jimple, upon which slicing (removal of statements which do not change the functionality, such as logging statements) and abstraction are performed. Jimple keeps tight correspondence to original Java code – given the Jimple node it is possible to retrieve Java abstract syntax tree node of the original program. Slicing and abstraction steps serve to decrease the state space during modelchecking. Slicing removes the statements which have no effect on functionality as well as statements and components which do not affect the property which is being checked.

Once simplified, Jimple code is transformed into the BIR – another intermediate language, which also maintains high correspondence with its input. BIR models the program as a set of asynchronous processes running guarded commands. System is not considered distributed and there are no interprocess communication constructs. BIR constructs serve as a basis for translation into another specification languages with their own tooling, which then do the actual model checking.

4. Extracting specification of interprocess communication

In this section we present the solution to the problem defined at the beginning. First we show an overview of the solution, basic principle how it is supposed to work. The rest of this sections provides the details of the parts of the solution.

4.1. Solution overview

This section presents a general overview of how Elixir module should be transformed into TLA⁺. We take an example module (fig. 9) and show what its parts are translated into. Also, an example modelchecking run is given to show the idea of what behaviour is specified by the generated specification.

```
1  defmodule ExampleServer do
2      use GenServer
3  
4      # Module API
5      def send(n) do
6          GenServer.call(:other, {:server, n})
7      end
8  
9      # Callbacks
10     @impl GenServer
11     def init({}) do
12         {:ok, 0}
13     end
14 
15     @impl GenServer
16     def handle_cast({:client, num}, _state) do
17         {:resp, n} = send(num + 1)
18         {:noreply, n}
19     end
20 
21     @impl GenServer
22     def handle_call({:server, num}, _state) do
23         {:reply, {:resp num}, num}
24     end
25 end
```

Figure 9. Example Elixir module which uses GenServer.

We base specification generation on GenServer module usage. In practice it means starting generation from those modules which use GenServer functions and implement the callbacks. Typically such modules have a clear structure – its functions are split into two groups: module API and callbacks.

Module API group functions, as the name suggests, provide the API for the module and often wrap the GenServer function calls. According to their purpose they are called with different argu-

ments from different places in the rest of the system. Bodies of the functions are translated as a series of expressions (called ”lines” in generated specification), each of which changes the process state. The same method should be used to translate any sequential code.

Elixir

```
1 def send(n) do
2     GenServer.call(
3         __MODULE__,
4         {:server, n}
5     )
6 end
```

TLA⁺

$$
\begin{array}{l}
\mathit{line1}(proc) \triangleq \\
\quad \text{LET} \\
\qquad n \triangleq P!arg(proc, 1) \\
\quad \text{IN} \\
\qquad P!bind(proc, \text{"n"}, n)
\end{array}
$$

$$
\begin{array}{l}
\mathit{line2}(proc) \triangleq \\
\quad \text{LET} \\
\qquad to \triangleq \text{CHOOSE } p \in Processes : p \neq proc.self \\
\qquad n \triangleq P!val(proc, \text{"n"}) \\
\quad \text{IN} \\
\qquad GenServer!call(proc, to, \langle \text{"SERVER"}, n \rangle)
\end{array}
$$

$$
\begin{array}{l}
\mathit{line3}(proc) \triangleq \\
\quad P!return(proc, P!return_value(proc))
\end{array}
$$

Figure 10. Simplified example of a function module for `send` function from figure 9.

Figure 10 displays a simplified example of sequential code translation. There are three TLA⁺ operators defined. *line1* operator assigns arguments with which the function was called to the parameter names. It is generated from the function definition signature (line 1). *line2* operator corresponds to *GenServer* module function call on lines 2 to 5 and has the same meaning. The last operator does not appear in Elixir code as it implements the Elixir language feature that the result of the last statement in the function definition is what is returned when function is called. We provide a more detailed description of generated function modules in section 4.4.1, however it generally is out of scope of this thesis.

Callback functions are not supposed to be called directly by the rest of the system code, they are to be called only by the GenServer module, mostly to handle incoming messages. Each of the function definitions is generated into two specification parts: function module and a message receiving action. Function module is produced exactly the same as for any other sequential code. Message receiving action is fully separate action in the generated specification which for any process which is ready to receive the message, selects the message from the global message queue and ”calls” the handler function on that process with the received message and stored process state passed as parameters. Actual handler behaviour is delegated to the handler function module. More detailed description and a translation example is provided in section 4.6.2.

Message sending is modelled simply as putting it into the the global message queue. GenServer module functions which send messages to multiple processes (`abcast` and `multi_call`) are modelled as sending multiple messages, one for each known process. More precise addressing is not implemented at the moment. All produced messages are sent at the same time – messages to send are included in the result of the function module expression and are put into global message queue right after they are produced.

Synchronous communication in addition to sending the message also requires waiting for the response. This is done by putting the process into a blocked status. While process is blocked no messages can be processed until response is received. A pair of actions, one to collect the responses from all processes, and another to form and deliver the received responses, model this aspect of the communication. See section 4.6.3 for definitions and detailed descriptions of these actions.

In the translated specification we distinguish between and refer to four kinds of states – process state, application state, system state and specification state. When any of these states is refered to in the text below, it is done with the following meanings in mind.

Process state. It is the state of a single process, containing all the things needed to execute sequential code on that process. It contains information about which expression of which function a process should execute next, its call stack, return value of last called function, etc. More detailed description of the structure of this state is given in section 4.2.1.

Application state. This state is the state which manipulated by any Elixir program which is using GenServer module. When message is received, this state is passed into message handling function and later is returned from there, possibly modified. This state is fully defined by the user and is a part of system state.

System state. This state stores the values required to ensure correct message delivery to processes. It corresponds to the internal Elixir GenServer module functions state and contains such things as which message process is currently processing, messages process is waiting a reply to, known application state, etc. More detailed description of this state is given in section 4.2.2.

Specification state. This state does not appear in the system implementation, instead it refers to the state any TLA+ specification goes through during modelcheking. When talking about generated specification, this state means a combination of all other states.

Modelchecking the specification generated from the code given in figure 9 with two processes (p_1 and p_2) and with the initial message queue containing a single client message $m_1 \triangleq [from \mapsto c_1, to \mapsto p_1, msg \mapsto \langle \text{"CLIENT"}, 1\rangle]$ from client c_1 would result in the following sequence of steps the specified system goes through:

1. *init* function is executed on at least p_1, which sets up initial application state. For p_2 this can happen later, but before first message is received by p_2.

2. Message receiving action for *handle_cast* is enabled for process p_1 and calls *handle_cast* function on the receiving process. Received message is removed from the message queue.

3. Several specification states are produced by the *handle_cast* function operators which only change p_1 process state until *GenServer.call* function is called from the inside of *send* function TLA+ module (fig. 10).

4. *GenServer.call* puts the supplied message into the message queue and blocks p_1. As a result of this expression, message queue contains a single message $m_2 \triangleq [from \mapsto p_1, to \mapsto p_2, msg \mapsto \langle \text{"SERVER"}, 2\rangle]$ and process p_1 is in status *blocked* waiting for reply to that message.

5. Message receiving action for *handle_call* becomes enabled for process p_2 as there is a message matching the required condition ($m.msg[1] = \text{"SERVER"}$) so *handle_call* is called on p_2.

6. Several specification states are produced by *handle_call* function TLA+ module operators which execute the behaviour of the function and set its return value which contains the response to the caller and changed process state.

7. Response to p_1 is taken from return value and put to the message queue. p_2 system and application states are also updated accordingly with the handler return value. p_2 returns to waiting status and is ready to receive further requests.

8. Actions for blocked processes to receive the responses become enabled for p_1, first *waiting_responses*, then *deliver_responses* and message from message queue is put as a return value from *GenServer.call* function.

9. p_1 finishes processing the original request, returns from the message handler function. All states are updated, and process becomes ready to accept other messages.

These steps do not correspond one-to-one with specification states, they only demonstrate a general sequence.

4.2. Process and communication model

It was chosen to model distributed Elixir system as a set of processes which send messages into and read from a global message queue. Each process is independent from others. Global message queue is not ordered, messages in it can be delivered in any order. This allows to have a non-deterministic message delivery.

No distinction is made between Elixir processes and nodes – in Elixir interprocess communication looks exactly the same regardless of whether processes are on the same node or on different. There are differences in message delivery guarantees but these are out of scope of this work.

A set of existing processes is assumed to be known in advance and should be provided by the user. In TLA+ this is done by specifying the value for constant, e.g. `Processes`. Process lifecycle, creating and destroying processes is out of scope of this work.

Process at any given moment can be in one of the five possible statuses: *initialized*, *waiting*, *processing*, *finished* or *blocked*. Status *initialized* is a one-time status for each process, which

occurs directly after `init` function finishes executing. In this status application state returned by `init` function is saved to later pass it into the message handler function. In status *waiting* the process is not doing anything, and is just waiting for some message to arrive. Status *finished* is a intermediate state between *processing* and *waiting* used to process the handler return value, update process state and immediately moves the process to status *waiting*. Replies to synchronous calls are sent out in this status as well. Status *processing* is unique among other process statuses since it does not have a single explicit marker in the process state. A process is considered to be in status *processing* when it is executing any sequential code and is not in any other status. Status *blocked* is used to mark the process which is waiting for replies to the messages it sent. Transitions between these statuses are shown in figure 11.

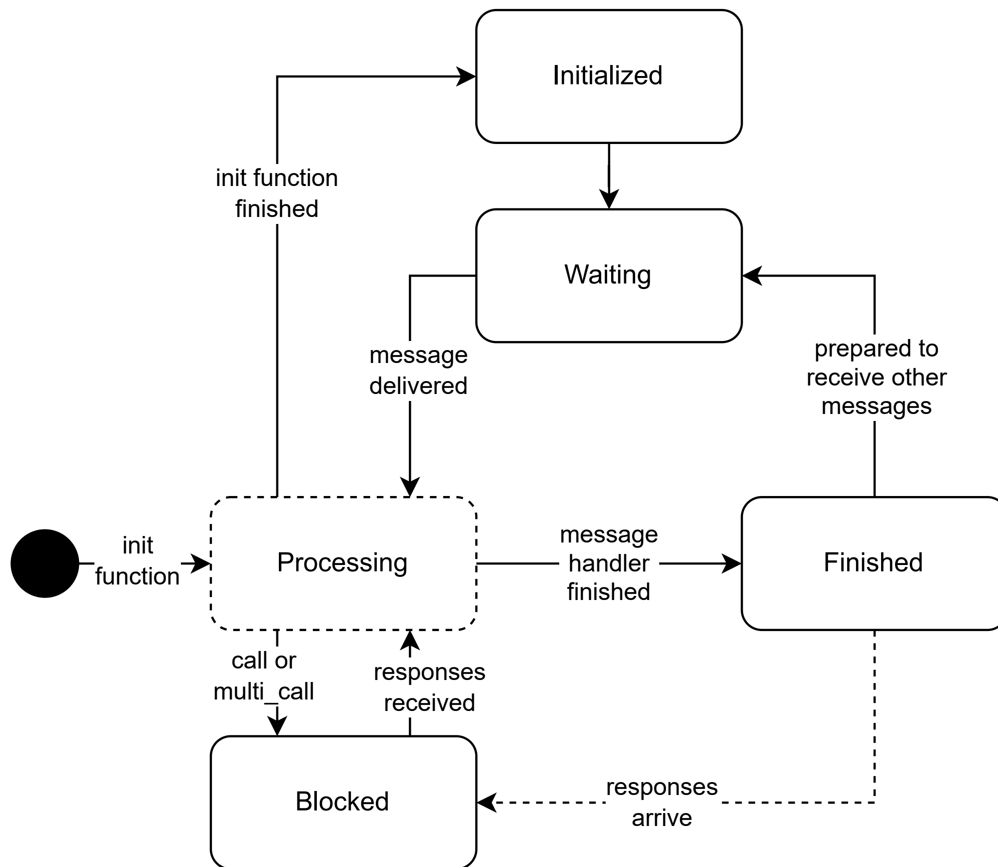

Figure 11. Process statuses and possible transitions between them.

4.2.1. Process state structure

Process state structure is heavily influenced by TLA⁺ specification which is generated from PlusCal specification as this is the format which was used in [Bra23]. Process state is modelled as a TLA⁺ record with the following fields:

- **self.** Process identifier. It is provided to simplify message creation. It is set in the initial state of generated specification and does not change.
- **pc.** Program counter. Specifies what process is currently doing. It is also a record of two fields: *fn* and *line*. Usually, *fn* contains the name of the function module and *line* signifies the

expression in that module which will be executed next. There are four special values of the *fn* field, one for each non-processing state. When process is in those states, *line* is not used.

- **stack.** Represents the stack of running process. Allows to have calls to other functions the same in TLA⁺ as in Elixir, which simplifies translation. During function call, current *pc* and *context* are saved here and restored upon return. Passed arguments are also stored here. See operator descriptions in section 4.5.1 for more detailed description of what is put onto the *stack*.

- **context.** Keeps known variables and their values in some scope. Is saved on the stack when calling other functions and a new one is created in its place. Allows to have variable scoping in TLA⁺ function modules and independent function module generation since each module does not need to know about other function's variables. This component is described in more detail in section 4.4.2.

- **return_value.** Keeps value returned by the last called function. Allows returning a value from one function to the other. This field is also used to generate the reply when synchronous request is being handled.

- **sent_msgs.** Process message output buffer. Messages sent by the process are put here and moved to global message queue after expression is executed.

4.2.2. System state structure

System state variable structure is meant to store the state relating to communication between the processes. It mostly represents the information kept by internal Elixir GenServer module function variables. Overall system state is represented by a function which maps each process to a record with the following fields:

- **state.** Application state, as it is returned by the `init` function and message handler functions. Represents user defined program state, which is passed into message handler function when message is received.

- **processing_message.** Stores the whole message which is currently being processed. Set when message is received and is used to identify which message a reply will be for.

- **wait_replies_to.** A set of messages process is waiting replies to. Messages are added into this set when synchronous call is issued to know which messages have not yet received a reply.

- **arrived.** A set of messages which are replies to the previously sent synchronous requests. Replies are stored here until all awaited replies are received and they can be delivered to the caller. This is required to ensure `multi_call` function works as expected.

4.2.3. Message structure and queue

Global message queue is modelled as a simple set of records with the following fields:

- **id.** Unique message identifier. It is used to determine which message some response is responding to. Currently implemented as a always increasing natural number.
- **from.** Sending process identifier. Used during response generation to determine which process needs to be sent the response.
- **to.** Receiving process identifier. Used in handler actions, to determine on which process handler function needs to be called.
- **msg.** Actual message sent. Set by the user or by the sending function module. Is passed into handler function module as a parameter.
- **reply_to.** An identifier of the message this message is replying to. If a message is not a response, then it is set to 0.

Introduction of unique message identifiers was done to make timeout during synchronous requests more viable. To prevent the delivery of the response which was sent earlier than the request the simplest solution was chosen – always increasing number as a message identifier. Although timeouts are outside of the scope of this work, since their modelling would require modelling process lifecycle, it was deemed that having a possibility to expand in this direction was better than not having one.

4.3. Data matching in function definitions

Data patterns in Elixir function definitions are transformed into matching conditions during specification generation. Data structures during this translation are modelled as described in [Bra23].

This approach currently cannot differentiate lists and tuples as both are transformed into TLA+ sequences (rules TUPLE-1 and TUPLE-2). Given the following two function headers:

```
def handle_call({:some_msg, {1, 2}, some_var}, state)

def handle_call({:some_msg, [1, 2], some_var}, state)
```

Listing 1: Similar function parameter patterns, one using a tuple, other using a list.

the following identical match conditions would be generated:

$$msg[1] = \text{"SOME_MSG"} \land msg[2][1] = 1 \land msg[2][2] = 2 \quad (1)$$

Usage of TLC on generated specification is hindered by runtime error when comparing values of different type. For example, these two definitions would not be possible to modelcheck:

```
def handle_call({1, some_var}, state)

def handle_call({"some str", some_var}, state)
```

Listing 2: Unmodelcheckable combination of function headers.

TLC needs compared values to be of the same type so if modelchecking for generated specification were attempted, TLC would fail with runtime error when trying to compare 1 to "some str"[2].

To avoid this issue message patterns have to be of the similar structure, to make use of TLC's minimal boolean expression evaluation. This means that message prefixes should be of the same type and sufficiently disambiguate between the messages, for example:

```
def handle_call({:some_msg_1, {1, 2}, some_var}, state)

def handle_call({:some_msg_2, [1, 2], some_var}, state)
```

Listing 3: Handler function headers with different atoms as message prefixes.

4.4. Sequential code specification model

Sequential code generation is out of scope of this work but is nonetheless necessary to have a complete picture of how interprocess communication is modelled since it defines the exact behaviour of the system and produces the messages to other processes. Specification generation for sequential code is deliberately incomplete, just the basic principles are presented. For a detailed exploration on how to model sequential code see [Bra23].

In general, sequential code is loosely based on how PlusCal specification is translated into TLA⁺ since Elixir code is translated into PlusCal by [Bra23]. Although modifications were made to make the translations modular and allow usage in specification for the whole system, the possibility to modelcheck each function in isolation is preserved.

Similarly as in [Bra23], we take sequential code in units of functions – each Elixir function definition becomes a TLA⁺ module. This introduces a degree of composability into the generated specification, making it a bit easier to understand as well as giving the possibility to run modelchecking on function modules separately from the system as a whole.

In Elixir it is allowed to have several definitions for a function which are differentiated at runtime by pattern matching the parameters with the given arguments. This functionality is not available at this moment, each such definition is considered a separate function and differentiation between them should be done where they are called.

In the rest of this section we present the parts of the function TLA⁺ module and their purpose.

[2]We have found a way to get around this although that was too late to used for this thesis. See section 5 for a short description of it.

4.4.1. Function module structure

Function module represents a single Elixir function definition. Only *Processes* and *NIL* are defined as constants in function modules, meaning that function module needs to be provided existing processes and a value used instead of Elixir `nil` by the user. Elixir function behaviour is expressed by several operators for each expression in function definition and several operators and formulas which are common for all function modules. These common operators and formulas constitute a function module interface and further on will be referred to as such. Common function module interface allows to treat each function module equally in the higher level TLA+ modules, thus simplifying generation.

Several TLA+ operators and formulas comprise the function module interface:

- ***lines*.** A set of integers, one for each expression in a function. For example, if function consists of 5 expressions, then $lines \triangleq 1 \,..\, 5$. It is used to check if certain expression can be executed.
- ***name*.** A string constant, holds function name. Its purpose is to be used by the overall system specification module to check which function a process is currently running.
- ***line_enabled(proc, l)*.** Evaluates to TRUE if given process is at the point where it has to execute the given expression. Usually delegates to *Process!line_enabled* with *name* and *lines* as parameters and thus serves only as a simplification for the higher level module.
- ***line_action(proc, line)*.** Executes given function expression on the given process. The result is next process state. Depending on given expression number, delegates to particular expression operator defined in the same module.

Functionality of the function is specified in a series of expression operators. Each expression is a discrete process step and corresponds to a logical Elixir process step during execution. For example, given the following Elixir code line,

```
x = first_fun(second_fun(100, third_fun()), "constant str")
```

Listing 4: Example Elixir code line

it should be translated into the following expression operators:

$$line1(proc) \triangleq P!call(proc, \text{"third_fun"}, \langle\rangle) \tag{2}$$

$$line2(proc) \triangleq P!bind(proc, \text{"__tmp1"}, P!return_value(proc)) \tag{3}$$

$$line3(proc) \triangleq P!call(proc, \text{"second_fun"}, \langle 100, P!val(proc, \text{"__tmp1"})\rangle) \tag{4}$$

$$line4(proc) \triangleq P!bind(proc, \text{"__tmp2"}, P!return_value(proc)) \tag{5}$$

$$line5(proc) \triangleq P!call(proc, \text{"first_fun"}, \langle P!val(proc, \text{"__tmp2"}), \text{"constant str"}\rangle) \tag{6}$$

This example demonstrates that a single Elixir expression can yield several expressions in generated TLA+ function specification, e.g. variable assignments, and nested function calls cannot be accomodated otherwise.

Operators from P module, like $P!call$, are defined in a separate helper module *Process*. See section 4.5.1 for a more detailed description of this module and its operators. This module is included into each function module with INSTANCE TLA+ command.

It is important to note that the last expression of each function must be a return statement: $P!return(proc, \text{"any value"})$. Without this it is not possible to return to the calling function.

4.4.2. Function context

Function context is defined as a way to set and access function variables by their name, to provide a way for each function to have a separate variable scope and to simplify context switching during calls and returns from and to other functions.

Function context is a set of records with fields *name* and *value* which associate a variable name with its value. There are two operators defined for function contexts in *Context* TLA+ module – *get_val* and *set_val* for getting and setting the value to some variable name. *Context* module is used internally by *Process* module (which is described in section 4.5.1) and its operators are not used during specification generation, they are exposed through the *Process* module operators.

4.5. Helper TLA+ modules

To simplify specification generation and to reduce duplication of common funcitonality, several helper TLA+ modules were defined. In this section we describe their responsibilities and operators.

These modules are meant to be included into other modules using INSTANCE TLA+ command as follows:

$$\begin{aligned} &\textsc{local}\ P \triangleq \\ &\qquad \textsc{instance}\ Process\ \textsc{with}\ Processes \leftarrow Processes, \mathit{NIL} \leftarrow \mathit{NIL} \end{aligned} \tag{7}$$

INSTANCE command substitutes the module constants with what is given on right side of $\leftarrow$ and makes resulting operators available through local module name, in this example its P.

Such separation provides a layer of abstraction in specifications and makes them less verbose and simpler to generate. Also, such reuse would be helpful if there were some properties proven for provided operators – those properties could be used to prove properties of all modules which use the included modules.

Most of module operators defined in helper modules are deterministic – their result depends solely on their arguments. Those operators which are not deterministic are explicitly indicated as such.

4.5.1. *Process* module

Process TLA+ module provides operators to manipulate process state. Mostly this means changing program counter (*pc*) and program *stack*. This is accomplished by defining several deterministic operators, which take process structure as parameter and produce next process state. These operators could be divided into two main groups.

First group of operators is related with function calls and contains *call*, *return* and *to_finished* operators. *call* and *return* operators implement generic function call, one issues it, while other allows to return back from it. *to_finished* operator makes use of the *call* operator to execute a call to a non-existent function, which puts the process into status *finished*. See solution overview in section 4.1 for a detailed description of process statuses.

The other group of processes is concerned with retieval and modification of data available in the function. There are two kinds of such data – values passed as function arguments and local variables. The former kind of data can only be read and for that purpose operator *arg* is defined, which allows to retrieve a value of the argument, given its index (as it often is in TLA⁺ – index is 1-based). The latter kind is served by two operators – *bind* and *val*. These operators allow binding a value to and retrieving value bound to a given name, thus enabling variable assignments and references.

Finally there is one operator which does not belong in either of the groups – *inc_pc*. Its purpose is to increment the passed *pc*. It produces a new *pc* by just incrementing the line number in the old one. This operator is often used by other operators and helper modules.

4.5.2. *System* module

System TLA⁺ module is responsible for overall system state management. It provides operators to update system state, keep track of what message is being processed and replies received. As such, it performs the internal functionality of Elixir GenServer module, and tracks state needed for it.

There are 5 operators defined by this module, which can be divided into two groups according to the context of their use. First group enables proper reply delivery for GenServer module *call* and *multi_call* function calls and it consists of three operators: *set_wait_replies_to*, *received_response_for* and *clear_arrived*. First one stores the sent messages for which replies need to be received before returning them from GenServer module function call. *received_response_for* performs second part of this process – it removes the message from wait list and adds its reply to the list of arrived replies. The last one, *clear_arrived* simply clears the arrived replies list and is used as a last step of the process, to avoid repeated deliveries of subsequent GenServer *call* invocations.

The second group of operators consists of two unrelated operators – *init* and *set_app_state*. The former initializes empty system state, thus hiding the details while the latter one allows to save the application state between the message handler function calls.

4.5.3. *Messaging* module

Messaging TLA⁺ module provides operators for message queue management. This module differs from other helper modules by having a variable *nextMsgId* which needs to be substituted during module instantiation. Currently it is expected that *nextMsgId* is a non-decreasing natural number since it allows to have unique message identifiers. Variable updates are also expected to be done where the module is used, not in the module itself. *Messaging* module defines five operators: *full_msgs*, *bulk_send*, *reply*, *drop* and *is_a_reply_to*. The rest of this section describes them.

full_msgs is a construction operator. Given a set of records with fields *from*, *to* and *msg* it returns a complete set of messages which are ready to put into message queue. It is used in all message sending operators. *full_msgs* is the only not fully deterministic operator since it uses a module variable *nextMsgId* to assign message identifiers and requires variable to be modified externally.

bulk_send and *reply* operators are used to send messages. They construct the full messages using *full_msgs* and add them to the message queue. *bulk_send* adds several messages to the message queue and is meant to handle Elixir functions which send several messages at once, such as *abcast* and *multi_call*. *reply*, on the other hand sends a single message, which is a response to the previously sent message. It assigns the *reply_to* field in message structure.

drop operator simply removes given message from the queue. It is defined only for better code organisation.

Operator *is_a_reply_to* is a boolean operator which given two messages returns true if the first one is a reply to the second one. It is used to detect replies in the message queue and deliver them to the waiting process.

4.5.4. *GenServer* module

GenServer TLA+ module is meant to provide TLA+ definitions to Elixir *GenServer* module functions. Currently it contains simplified definitions for *call* and *cast* functions, as well as their equivalents for multiple receivers – *multi_call* and *abcast*. Although limited, these functions are enough to model two main communication patterns – synchronous and asynchronous.

cast(proc, to, msg) is the simpler one of the two. Since cast is asynchronous, it does not need to make sending process to wait for response from the receiving process. This allows us to just add the message to process message output buffer.

call(proc, to, msg) sends the message synchronously. It works the same way as *cast* except it also blocks the calling process so that it starts waiting for the reply. Although *call* is not a function module, reply is made available as a return value, which simplifies sequential code generation.

multi_call and *abcast* work the same as their single receiver counterparts, except messages are sent to all known processes. It should be possible to have a more precise addressing but currently no such functionality is implemented.

All these operators create messages of the form $[from \mapsto f, to \mapsto t, msg \mapsto m]$. This is not a complete message form but its enough information for *Messaging* module operators to create the full messages.

4.6. Generating specification for whole system

In this section we present how the pieces described above are combined to generate a complete specification for a distributed system. First we will present the general structure of the specification and what things are expected to be modelled as and then we will describe in more detail the parts of the specification which are related to the interprocess communication.

In all examples operators of helper modules *Process*, *Messaging* and *System* are used. They are instantiated as shown in section 4.5 with local names P, M and S respectively.

4.6.1. General structure

Generated specification requires several sets to be defined by the user. *Processes* and *Clients* are the sets of available processes and clients. Processes are those which participate in the specified system, while clients are external participants which are sending messages to the system. Constant *InitialMessages* is a set of messages to be put into message queue in the initial specification state. These messages must be TLA⁺ records of the same structure as produced by GenServer module operators: $[from \mapsto f, to \mapsto, msg \mapsto m]$. Here f and t must come from either *Processes* or *Clients* while m is fully user defined and is an actual message as it will be received by the t. Constant *InitParams* must be a function which maps system processes to the arguments of `init` function. Finally, *PreInitialAppState* constant defines initial application state to be used before *init* function produces actual initial state for the process.

The state of generated system specification is stored in three variables: *procState*, which keeps the process state of each process, *sysState*, which keeps the system state of each process and *messageQueue* which is a global set of all messages currently in transit. *procState* is initialized in a way where each process is about to execute the `init` function. *messageQueue* is initialized to the *InitialMessages* constant which allows user to control the exact scope of modelchecking.

After constants and variables are defined, local helper modules and function modules are instantiated under their own namespaces using INSTANCE TLA⁺ operator, as shown in section 4.5.

Module instantiations are followed by generated handler actions, one for each GenServer handler function. These actions are described in more detail below, in section 4.6.2.

After the handlers actions common to all specifications are defined: *after_init*, *function_lines* *handler_finished*, *waiting_responses* and *deliver_responses*. *after_init* is an action which processes init function result an saves the initial application state. *function_lines* is a catch-all action for executing any line of any function in any process. *handler_finished* prepares a process which has just finished executing a message handler function to accept another message. *waiting_responses* and *deliver_responses* are responsible for response delivery to waiting processes. These actions are described in more detail below (section 4.6.3).

Finally, the *Init*, *Next* and *Spec* formulas are added. *Init* defines initial state of the specification – on each process `init` function was called and its execution is to begin. *Next* describes a specification step and is defined as a disjuction of all actions – both message receiving and predefined by this template. Lastly, *Spec* describes the specified system as a whole and is of the customary form $Spec \triangleq Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next)$.

4.6.2. Message receiving actions

Message receiving actions are generated from GenServer handler function signatures. We use only `handle_cast` and `handle_call` for this purpose. We exclude `handle_continue` and `handle_info` since we assume that those wishing to check whether a certain implementation has

certain properties will be interested in a specific algorithm implemented in that system and to implement algorithms `handle_cast` and `handle_call` functions are more natural choice.

The purpose of message receiving actions is to take a message from *messageQueue* and pass it together with application state into correct handler function as arguments. They also remove the message from *messageQueue*. Figure 12 shows an example handler action which would be generated from the given signature.

Since multiple function definitions differentiated by pattern matching arguments are modelled as separate function modules with pattern matching to be done outside of function itself (as described in section 4.4), message receiving actions are also responsible for it. In case where it is needed to match on the incoming message, pattern matching conditions are generated into the message receiving action (line a in the figure 12). See section 4.3 for detailed description of how data matching conditions are generated.

Before executing a call to the handler function (line b), process state is manipulated in such a way that return from handler function would bring the process into status *finished*. This is done using *Process* module operator *to_finished* (see section 4.5.1). Process status *finished* is handled by a single action *handler_finished* which is described in the next section (4.6.3).

Received message is also saved in the system state (line c) to use it later to generate the reply in action *handler_finished* (see section 4.6.3 for the description).

Elixir

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

TLA⁺

$$
\begin{aligned}
&\mathit{handler_1} \triangleq \\
&\quad \exists m \in messageQueue, t \in Processes \\
&\qquad \land m.to = t \\
&\qquad \land m.msg[1] = \text{"CLIENT"} && \text{(a)} \\
&\qquad \land P!waiting(procState[t]) \\
&\qquad \land procState' = upd_proc_state(t, \\
&\qquad\quad P!call(&& \text{(b)} \\
&\qquad\qquad P!to_finished(procState[t]), \\
&\qquad\qquad handle_cast!name, \\
&\qquad\qquad \langle m.msg, sysState[t].state \rangle)) \\
&\qquad \land messageQueue' = M!drop(messageQueue, m) \\
&\qquad \land sysState' = upd_sys_state(t, \\
&\qquad\qquad S!set_processing_message(sysState[t], m)) && \text{(c)} \\
&\qquad \land \text{UNCHANGED } nextMsgId
\end{aligned}
$$

Figure 12. Message receiving action example.

4.6.3. Other actions

This section describes TLA⁺ actions which are common to all specifications. They can be split into two groups: sequential code internals and message delivery internals. There is also *after_init* action, which handles the result of sequential code execution while being a specialisation of a message delivery action.

Actions in sequential code internals group are less numerous and are mainly responsible for enabling sequential code execution. There is one action (*function_lines*) and one operator (*fn_line*) in this group. *fn_line* is defined primarily to shorten the *function_lines* action and to reduce duplication.

Purpose of *function_lines* action is to execute a function expression on some process which is allowed to do that (i.e. is in status processing) and update the process state afterwards. An example of such action is provided in equation 8. It shows *function_lines* action as it would be generated for the example module in figure 9. Overall, this action is a large disjunction, with as many clauses as there are function modules – one clause per function module. Each of the clauses is a *let* statement, setting two values – *line_enabled* (line 8a), which is true if some line of some function can be executed on some process, and *line_action* (line 8b), which is a result of executing that line on that process. These values are passed into *fn_line* operator (line 8c) which we describe below.

$$
\begin{aligned}
& function_lines \triangleq \\
& \quad \exists p \in Processes : \\
& \qquad \lor \exists l \in send!lines : \\
& \qquad\quad \text{LET} \\
& \qquad\qquad line_enabled \triangleq send!line_enabled(procState[p], l) && \text{(8a)} \\
& \qquad\qquad line_result \triangleq init!line_action(procState[p], l) && \text{(8b)} \\
& \qquad\quad \text{IN} \\
& \qquad\qquad fn_line(p, line_enabled, line_result) && \text{(8c)} \\
& \qquad \lor \exists l \in handle_cast!lines : \\
& \qquad\quad \text{LET} \\
& \qquad\qquad line_enabled \triangleq handle_cast!line_enabled(procState[p], l) \\
& \qquad\qquad line_result \triangleq handle_cast!line_action(procState[p], l) \\
& \qquad\quad \text{IN} \\
& \qquad\qquad fn_line(p, line_enabled, line_result) \\
& \qquad \lor \ldots \textit{(other function modules)} \ldots
\end{aligned}
$$

Equation 8: Shortened *function_lines* action as it would be generated for the Elixir module in section 4.1.

fn_line operator describes what is done after executing the function module expression. It is given in the equation 9. It has two parameters passed into it – *line_enabled* and *line_result*, corresponding to the results of *Process* module operators *line_enabled* and *line_action*. Concrete values

from exact function module are passed into this operator in the *function_lines* action. The body of *fn_line* a single LET statement with two values set – *becomes_blocked* and *complete_messages*. *becomes_blocked* is true if process should go into status *blocked* and *complete_messages* are messages which were sent by some process as a result of last function module expression with supporting information (mainly message identifier) set. Operator checks if given line expression can be executed (line 9a), assigns all specification variables – updates process state (line 9b), puts all sent messages into the global queue (lines 9c and 9d) and updates the application state if needed (line 9e). P, M and S are local instances of *Process*, *Messaging* and *System* modules respectively. See sections 4.5.1 for *Process* module, 4.5.3 for *Messaging* module and 4.5.2 for *System* module operator descriptions.

$$
\begin{aligned}
&fn_line(process, line_enabled, line_result) \triangleq \\
&\quad \textsc{let} \\
&\qquad becomes_blocked \triangleq P!blocked(line_result) \\
&\qquad complete_messages \triangleq M!full_msgs(line_result.sent_msgs) \\
&\quad \textsc{in} \\
&\qquad \land\ line_enabled && \text{(9a)} \\
&\qquad \land\ procState' = upd_proc_state(process, line_result) && \text{(9b)} \\
&\qquad \land\ messageQueue' = M!bulk_send(messageQueue, complete_messages) && \text{(9c)} \\
&\qquad \land\ nextMsgId' = nextMsgId + Cardinality(complete_messages) && \text{(9d)} \\
&\qquad \land\ \textsc{if}\ becomes_blocked\ \textsc{then} \\
&\qquad\qquad sysState' = upd_sys_state(process, && \text{(9e)} \\
&\qquad\qquad\qquad S!set_wait_replies_to(\\
&\qquad\qquad\qquad\qquad sysState[process], \\
&\qquad\qquad\qquad\qquad complete_messages)) \\
&\qquad\quad \textsc{else} \\
&\qquad\qquad \textsc{unchanged}\ sysState
\end{aligned}
$$

Equation 9: *fn_line* operator.

Message delivery group of actions consists of four independent actions: *handler_finished*, *waiting_responses* and *deliver_responses*. The first one prepares the process to handle the next message after it has finished with the previously started handler function module. Equation 10 displays how this action is defined. The other two are required to ensure proper message response delivery to processes which did blocking requests (using *GenServer!call* operator) and are displayed in equations 12 and 13.

handler_finished action is only enabled for processes which are in status *finished* (line 10a) and its purpose is to send the reply (if any) and update the application state. Since Elixir handler functions can return `{:reply, reply, new_state}` or `{:noreply, new_state}` we need to check the first member of the sequence returned by the handler function TLA⁺ module (line 10b). Case when no reply is needed is simpler – we just update the application state (line 10e). When

$$
\begin{array}{ll}
\mathit{handler_finished} \triangleq & \\
\quad \exists p \in \mathit{Processes} : & \\
\qquad \land\ P!\mathit{finished}(\mathit{procState}[p]) & \text{(10a)} \\
\qquad \land\ \mathit{procState}' = \mathit{upd_proc_state}(p, P!\mathit{return}(\mathit{procState}[p], \mathit{nil})) & \\
\qquad \land\ \text{LET} & \\
\qquad\quad \mathit{return_type} \triangleq \mathit{procState}[p].\mathit{return_value}[1] & \\
\qquad \text{IN} & \\
\qquad\quad \text{CASE } \mathit{return_type} = \text{"REPLY"} \rightarrow & \text{(10b)} \\
\qquad\qquad \land\ \mathit{messageQueue}' = M!\mathit{reply}(& \text{(10c)} \\
\qquad\qquad\quad \mathit{messageQueue}, & \\
\qquad\qquad\quad \mathit{procState}[p].\mathit{return_value}[2], & \\
\qquad\qquad\quad \mathit{sysState}[p].\mathit{processing_message}) & \\
\qquad\qquad \land\ \mathit{sysState}' = \mathit{upd_sys_state}(p, & \text{(10d)} \\
\qquad\qquad\quad S!\mathit{set_app_state}(& \\
\qquad\qquad\qquad \mathit{sysState}[p], & \\
\qquad\qquad\qquad \mathit{procState}[p].\mathit{return_value}[3])) & \\
\qquad\qquad \land\ \mathit{nextMsgId}' = \mathit{nextMsgId} + 1 & \\
\qquad\quad \Box\ \mathit{return_type} = \text{"NOREPLY"} \rightarrow & \text{(10e)} \\
\qquad\qquad \land\ \mathit{sysState}' = \mathit{upd_sys_state}(p, & \\
\qquad\qquad\quad S!\mathit{set_app_state}(& \\
\qquad\qquad\qquad \mathit{sysState}[p], & \\
\qquad\qquad\qquad \mathit{procState}[p].\mathit{return_value}[2])) & \\
\qquad\qquad \land\ \text{UNCHANGED } \langle \mathit{messageQueue}, \mathit{nextMsgId} \rangle & \\
\qquad\quad \Box\ \text{OTHER} \rightarrow \text{UNCHANGED } \langle \mathit{sysState}, \mathit{messageQueue}, \mathit{nextMsgId} \rangle &
\end{array}
$$

Equation 10: *handler_finished* action.

reply is returned, it needs to be put into the global message queue. This is done by constructing the full reply message with the indication which message we are replying to and putting it into the message queue (line 10c). We also update the application state (line 10d).

after_init is a specialisation of the *handler_finished* action where instead of message handling function, *init* function return value is being processed. This action is displayed in equation 11. The purpose of this action is in the line 11a – save return value of `init` function as application state.

waiting_responses action is meant to collect the responses from all processes which the message was sent to. Since it is needed to wait for all responses before returning from *multi_call* it is necessary to have an action which accepts a single response and saves it somewhere to be returned later when all other responses are received. As it is currently defined in equation 12, responses are matched with messages waiting for response by *reply_to* field in the response (line 12a), which contains the identifier of one of the messages sent to other processes. Once the message is matched it is saved to the application state and is removed from the set of messages we are waiting a response

$$
\begin{aligned}
&after_init \triangleq \\
&\quad \exists p \in Processes : \\
&\quad\quad \land P!initialized(procState[p]) \\
&\quad\quad \land procState' = upd_proc_state(p, P!return(procState[p], \mathit{NIL})) \\
&\quad\quad \land \text{LET} \\
&\quad\quad\quad\quad init_ok \triangleq procState[p].return_value[1] \\
&\quad\quad\quad\quad init_value \triangleq procState[p].return_value[2] \\
&\quad\quad\quad \text{IN} \\
&\quad\quad\quad\quad \text{IF } init_ok = \text{"OK"} \text{ THEN} \\
&\quad\quad\quad\quad\quad sysState' = upd_sys_state(p, \\
&\quad\quad\quad\quad\quad\quad S!set_app_state(sysState[p], init_value)) \qquad \text{(11a)} \\
&\quad\quad\quad\quad \text{ELSE} \\
&\quad\quad\quad\quad\quad sysState' = upd_sys_state(p, \\
&\quad\quad\quad\quad\quad\quad S!set_app_state(sysState[p], \{\})) \\
&\quad\quad \land \text{UNCHANGED } \langle messageQueue, nextMsgId \rangle
\end{aligned}
$$

Equation 11: *after-init* action.

to (line 12b).

Such *waiting_responses* definition also allows to define a message timeout action. An additional action would be required with the same conditions as *waiting_responses* but different values assigned to the primed variables. It would terminate the waiting for the responses prematurely, thus simulating a timeout. We do not define timeout action since it would require modelling the whole process lifecycle which is outside the scope of this work.

Once all responses are received making them available to the function module is pretty straightforward. This is accomplished by the *deliver_responses* action shown in equation 13. All responses received and saved with *waiting_responses* action are mapped into tuples $\langle sender, message \rangle$ and returned back to the sending function operator (line 13a). This process allows function modules to pretend that *GenServer!call* and *GenServer!multi_call* behave like function calls simplifying the sequential code generation.

4.7. Generator program

A program prototype was written to extract specification from Elixir source code. Given the file which uses the Elixir GenServer module it is able to generate the overall TLA⁺ specification. In this section we describe the structure and capabilities of this program, as well as explain its usage.

Developed prototype consist of two parts – Elixir AST parser and TLA⁺ generator. AST parser is built using Elixir built-in AST extraction functions and recursively descends down the AST until required parts are found. In particular, we look for GenServer module usage and, if that is present, message handling function definitions. After handler function definitions are extracted, they are

$$
\begin{aligned}
&waiting_responses \triangleq \\
&\quad \exists p \in Processes, m \in messageQueue : \\
&\quad\quad \land P!blocked(procState[p]) \\
&\quad\quad \land sysState[p].wait_replies_to \neq \\
&\quad\quad \land m.to = p \\
&\quad\quad \land \exists w \in sysState[p].wait_replies_to : \\
&\quad\quad\quad \land m.reply_to = w.id && \text{(12a)} \\
&\quad\quad\quad \land sysState' = upd_sys_state(p, && \text{(12b)} \\
&\quad\quad\quad\quad S!received_response_for(sysState[p], w, m)) \\
&\quad\quad\quad \land messageQueue' = M!drop(messageQueue, m) \\
&\quad\quad\quad \land \text{UNCHANGED } \langle procState, nextMsgId \rangle
\end{aligned}
$$

Equation 12: *waiting_responses* action.

$$
\begin{aligned}
&deliver_responses \triangleq \\
&\quad \exists p \in Processes : \\
&\quad\quad \land P!blocked(procState[p]) \\
&\quad\quad \land sysState[p].wait_replies_to = \{\} \\
&\quad\quad \land procState' = upd_proc_state(p, \\
&\quad\quad\quad P!return(\\
&\quad\quad\quad\quad procState[p], \\
&\quad\quad\quad\quad \{\langle resp.from, resp.msg \rangle : resp \in sysState[p].arrived\})) && \text{(13a)} \\
&\quad\quad \land sysState' = upd_sys_state(p, S!clear_arrived(sysState[p])) && \text{(13b)} \\
&\quad\quad \land \text{UNCHANGED } \langle messageQueue, nextMsgId \rangle
\end{aligned}
$$

Equation 13: *deliver_responses* action.

passed into TLA⁺ generator. It maps the extracted handlers into respective TLA⁺ snippets and puts them into predefined specification template. Resulting TLA⁺ specification is printed to the standard output and can be redirected to the file using OS shell utilities.

Due to the scope of this thesis we do not generate any specification for sequential code. Program also does not analyze the whole Elixir project to find the modules which use GenServer module as indicating the file manually is sufficient for demonstration purposes.

Program is made available as a Mix (Elixir build tool) task and can be run using `mix gen_spec <filename>` command, where `<filename>` is a path to the file for which specification should be generated. This command has to be issued from the `gen_spec` folder in our repository for Mix task to be found. Program requires at least Elixir version 1.14.

4.8. Capabilities and limitations

In this section we present an informal evaluation of developed solution, examine its strengths and weaknesses.

The solution described in this thesis has several important strengths:

1. Described solution is modular. Created helper TLA+ modules encapsulate their respective areas well, if needed they can be improved upon separately from the rest of the specification. The same is true for the function modules concept, each function module is fully independent from others.

2. It is possible to modelcheck and prove properties for each function module in separately. If definitions of other used functions and a variable to assign process structure are added, it is possible to run modelchecking on any function in isolation.

3. State explosion is limited by the fully deterministic process execution. As long as no communication between processes is done during message processing, the number of states single process goes through will be the same for any message. This allows user to manage the state explosion by changing the initial message set and a set of processes.

Current solution also has several weaknesses:

1. State explosion limiting is not adequate for complex systems. Current sequential code specification generates quite a lot of intermediate states between the states which are important for communication. When running TLC, different orderings of those states will be checked, although that has no effect overall. It would be interesting to see if its possible to decrease the number of those states, however sequential code generation is outside the scope of this work.

2. There is no way to create or delete processes and adding it would require big changes to the current method. Process lifecycle management is of scope for this thesis.

5. Verification

We show correctness of our extraction method by showing that generated specification refines previously written abstract one. For this purpose we use our implementation of Bracha reliable broadcast [Bra87] toghether with its abstract specification[3]. We succeed in showing that the abstract specification holds as a property of generated specification, thus showing the refinement [LMT+02].

We generate a detailed specification from a single file. Due to the scope of this thesis, we only generate the message receiving part of the algorithm, the rest is sequential code, for which specification had to be written manually. Nonetheless, sequential part of the specification fully follows our method of sequential code translation to minimize changes in the generated part.

Abstract specification from generated one differs in one fundamental way – messages are not removed from the queue in the abstract one while they are removed from it in generated one, since this is closer to actual implementation. To get around this, we add additional action for modelchecking – *UpdAllSentMsgs* (see equation 14). It replicates messages added to the message queue into a new variable, which is then used for refinement mapping. This allows us to adapt to fundamental differences between generated and abstract specifications without changing generated specification.

$$\begin{aligned} UpdAllSentMsgs \triangleq\ & allSentMsgs' = allSentMsgs \\ & \cup \{m \in messageQueue' : m.msg[1] \in \{\text{"PROPOSE"}, \text{"ECHO"}, \text{"READY"}\}\} \end{aligned}$$

Equation 14: Append-only message queue update action.

Modelchecking was run with two processes in the system on a machine with 6-core AMD Ryzen 5 PRO 5650U processor and 16 GiB RAM. TLC was passed option `-workers 8` to make use of multiple cores available. Run took 3.3 seconds, 61419 specification states were found, 30880 of which were distinct. The same process with 3 processes did not finish after more than 10 hours with more than 190 million distinct states generated.

[3] Both are available in source code repository `https://github.com/mr-frying-pan/gen-tla-spec`. Implementation in bracha/lib/bracha.ex, abstract specification in gen_spec/tla/BrachaRBC.tla. Abstract specification was taken from `https://github.com/iotaledger/wasp/blob/6efe78da8ad661701a154d3e0ec534d0c9244a3b/packages/gpa/rbc/bracha/BrachaRBC.tla`

Results and conclusions

In the course of this thesis a method to extract TLA+ specification from Elixir source code was developed, thus achieving the aim of this research. We also verify it by modelchecking a generated specification for a non-trivial algorithm and showing that it refines the abstract specification.

In this work we have produced the following results:

1. A method of TLA+ specification extraction from Elixir source code. We have shown that it is possible to show that an implementation of non-trivial algorithm conforms to its specification. Developed method imposes two constraints on the source code for which specification can be generated:
   - GenServer Elixir module usage. We base our translation on the functions provided and required by this module.
   - Message structure similarity. As described in section 4.3, messages must have similar structure.
2. A program was developed, which given a file which contains Elixir module definition produces TLA+ specification for that system.
3. An example specification was generated and a refinement of an abstract specification of the same algorithm was shown.
4. An abstract datatype was defined[4], where a value of any concrete type is a TLA+ function with a single element in its domain and a value of that function is the value translated according to the rules defined in [Bra23]. This translation allows to avoid the TLA+ and TLC limitation on comparing values of different types and thus to define an equivalent of Elixir `match?` function.

Produced results can be found in thesis repository: `https://github.com/mr-frying-pan/gen-tla-spec`

From this work we make the following conclusions:

1. It is possible to define a translation from Elixir source code into TLA+ for a distributed system such that it is possible to generate automatically and refinement can be shown for generated specification. This allows to verify that system implementation conforms to the initial specification.
2. Developed translation method is modular. Each helper module is independent from each other and from the rest of the specification. This means they can be improved in isolation and properties can be proven for their operators. The same is true for each function module.

Method of translation of Elixir source code to TLA+ has been published in the Proceedings of the Conference "Lithuanian MSc Research in Informatics and ICT". It was also presented at the same conference.

[4] See gen_spec/tla/Type.tla in source code repository

References

[AE] AdaCore and Capgemini Engineering. Spark reference manual. `https://docs.adacore.com/spark2014-docs/html/lrm/the-standard-library.html`. Accessed: 2023-01-10.

[AEP04] Thomas Arts, Clara Benac Earle, and Juan José Sánchez Penas. Translating Erlang to µCRL. In *Proceedings. Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004.* Pp. 135–144. IEEE, 2004.

[Bra23] Deividas Bražėnas. *Extracting TLA+ Specifications out of Elixir Programs*. MA thesis, Vilnius University, 2023.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[CDH+00] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Păsăreanu, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering*, pp. 439–448, 2000.

[COR+01] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Tech. rep., Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.

[CS05] Koen Claessen and Hans Svensson. A semantics for distributed Erlang. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, pp. 78–87, 2005.

[dcon] Elixir documentation contributors. Elixir documentation. `https://hexdocs.pm/elixir/1.14.3/Kernel.SpecialForms.html`, Accessed: 2023 January.

[Fre01] Lars-Åke Fredlund. *A framework for reasoning about Erlang code*. PhD thesis, Mikroelektronik och informationsteknik, 2001.

[FS07] Lars-Åke Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pp. 125–136, 2007.

[GKM+08] David A Greve, Matt Kaufmann, Panagiotis Manolios, J Strother Moore, Sandip Ray, José Luis Ruiz-Reina, Rob Sumners, Daron Vroon, and Matthew Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(1):15–46, 2008.

[GR01] Jan Friso Groote and Michel A Reniers. Algebraic process verification. In *Handbook of process algebra*, pp. 1151–1208. Elsevier, 2001.

[Haf10] Florian Haftmann. From higher-order logic to haskell: there and back again. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 155–158, 2010.

[Jur19] S. Juric. *Elixir in Action*. Manning, 2019. ISBN: 9781617295027.

[KS11] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed computing principles, algorithms and systems*. Cambridge University Press, 2011.

[Lam] Leslie Lamport. PlusCal tutorial. URL: `https://lamport.azurewebsites.net/tla/tutorial/intro.html`. Accessed: 2024-05-20.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[Lam99] Leslie Lamport. Specifying concurrent systems with TLA+. *Calculational System Design*:183–247, 1999.

[LMT+02] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pp. 45–48, 2002.

[LN11] Hannes Lau and Uwe Nestmann. Java goes TLA+. In *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*, pp. 117–124. IEEE, 2011.

[MLH+14] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barkaoui, and Serge Haddad. An approach for verifying concurrent C programs. In *8th Junior Researcher Workshop on Real-Time Computing*, pp. 33–36, 2014.

[MMH13] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc., 2013.

[MVK22] Gabriela Moreira, Cristiano Vasconcellos, and Janine Kniess. Fully-tested code generation from TLA+ specifications. In *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing*, pp. 19–28, 2022.

[NMR+02] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, pp. 213–228. Springer, 2002.

[SBR+18] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 14–27, 2018.

[Tea22] The Coq Development Team. The Coq proof assistant, version 8.16, 2022-09. DOI: `10.5281/zenodo.7313584`. URL: `https://doi.org/10.5281/zenodo.7313584`.

[Tho18] D. Thomas. *Programming Elixir ≥ 1.6: Functional |> Concurrent |> Pragmatic |> Fun*. Pragmatic Bookshelf, 2018. ISBN: 9781680506136.

[WWP+15] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas 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*, pp. 357–368, 2015.