VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
DEPARTMENT OF SOFTWARE ENGINEERING

Extracting TLA$^+$ Specifications out of Elixir Programs

TLA$^+$ specifikacijų išskyrimas iš Elixir programų

Master's Thesis

Author:	Deividas Bražėnas	(signature)
Supervisor:	doc. dr. Karolis Petrauskas	(signature)
Reviewer:	partn. doc. Viačeslav Pozdniakov	(signature)

Vilnius – 2023

Abstract

In this research, we investigate a methodology that ensures the compliance of Elixir programs with TLA$^+$ specifications, developed by software engineers. The main component for linking the Elixir code with TLA$^+$ specifications are translation rules. These rules were defined and utilized during the implementation of a translation method, which is capable of converting sequential Elixir code into TLA$^+$ specifications. The correctness of the generated specifications is evaluated through model checking and refinement techniques, while the correctness of the translation method is ascertained by converting the generated specifications back to Elixir code and by conducting unit tests of the source program.

Keywords: TLA$^+$, PlusCal, Elixir, Formal methods, Distributed systems.

Santrauka

Šiame tyrime nagrinėjamas metodas, padedantis užtikrinti Elixir programos atitikimą programinės įrangos inžinieriaus kurtai TLA$^+$ specifikacijai. Vertimo taisyklės, skirtos Elixir kodą paversti į PlusCal kalbos konstrukcijas, yra esminė kodo susiejimo su specifikacija dalis. Šios taisyklės buvo apibrėžtos ir panaudotos įgyvendinant vertimo įrankį, nuoseklųjį Elixir kodą paverčiantį į TLA$^+$ specifikaciją. Sugeneruotų specifikacijų teisingumas tikrinamas modelio tikrinimu ir tikslinimu, o teisingas vertimo įrankio veikimas užtikrinamas konvertuojant sugeneruotą specifikaciją atgal į Elixir kodą ir vykdant pirminės programos vienetų testus.

Raktiniai žodžiai: TLA$^+$, PlusCal, Elixir, Formalūs metodai, Išskirstytos sistemos.

CONTENTS

Introduction

As more and more large-scale distributed systems are being developed, subtle but severe issues often arise in the production environment. This is a pretty common issue as distributed systems may be complex to understand – even a few interacting agents can lead to tens of thousands or even millions of unique system states. At that scale, it is impossible to test or even reason about every possible edge case. Thus, an obscure series of interactions among the components may lead to catastrophic issues. For example, when Amazon's Elastic Compute Cloud (EC2) hit a race condition, it caused serious downtime for major systems on the Internet [Tea11].

Formal verification is one approach to help programmers ensure the correctness of distributed systems. Formal verification of a program is the mathematical proof that it does what is expected of it. Creating a formal verification of the system's design may prevent subtle bugs from reaching the production environment, and finding bugs would be hard to find in any other technique [NRZ+15].

One of the formal specification languages, that has been successfully used on many projects in the industry (Compaq [LST+01], Intel [BL02], Paxos consensus algorithm [Lam05], Pastry distributed key-value store [LMW11]), is TLA+, which stands for Temporal Logic of Actions. It was created by Leslie Lamport and released in 1999. TLA+ is a language for modeling software above the code level and hardware above the circuit level. It is based on mathematics and does not resemble any programming language [Lam02].

TLA+ as a formal specification language is used in many worldwide-known companies, one of them being Amazon. Since 2011, engineers at Amazon have been using TLA+ to help solve complex design problems in critical systems that store and process data on behalf of their customers [New14]. In order to safeguard that data, Amazon relies on the correctness of an ever-growing set of algorithms for replication, consistency, concurrency control, fault tolerance, auto-scaling, and other coordination activities. Achieving correctness in these areas is a significant engineering challenge. These algorithms interact in complex ways to achieve high availability on cost-efficient infrastructure while coping with relentless rapid business growth. As of February 2014, Amazon has used TLA+ on 10 large complex real-world systems [New14]. In every case, TLA+ has added significant value, either preventing subtle, severe bugs from reaching production or giving enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness [New14]. Executive management of Amazon is now proactively encouraging teams to write TLA+ specs for new features and other significant design changes. In addition, in annual planning, managers are now allocating engineering time to use TLA+ [New14].

One reason for creating a specification of the system is to check if it does what we want it to. This can be done with model checking. Model checking is a computer-assisted method for the analysis of dynamical systems that state-transition systems can model [CHV+18]. This is achieved by verifying liveness and safety properties in all possible states of a finite-state system [Lam02]. Model checking can be viewed as a sophisticated form of testing.

Testing is not a substitute for good programming practice, but we do not release programs for others to use without testing them. Model checking algorithms prior to submitting them for publication should become the norm [Lam06]. For example, TLA$^+$ specifications can be checked with TLC, which works by checking invariance properties of a finite-state model of the specification [YML99].

For all software artifacts, at least two types of entities are of interest for verification: the final version of the design requirements and the source code that implements them. The main problem is that even though the system may have a perfect formal specification before the implementation, critical mistakes may be introduced during the development phase. It is great if these issues are detected in the pull requests or testing, but sometimes they are not, and system users are at significant risk. The gap between the formal specification and implementation may be reduced by using specific programming languages that allow the verification (ADA [AT20], Spec# [DeL04]), generating code from specifications (APTS [LH08], TLA$^+$ [Maf19]), or extracting specifications from the code. The latter way will be considered in the research as it allows the software engineers to concentrate on the programming, thus avoiding the formal specifications becoming an obstacle when quick changes are required.

One of the popular programming languages primarily used for heavily trafficked websites and highly scalable distributed applications is Elixir, which was released in 2012. Elixir leverages the Erlang VM (BEAM), known for running low-latency, distributed, and fault-tolerant systems [Tea]. Furthermore, as Elixir language is permeated with metaprogramming (a programming technique in which computer programs can treat other programs as their data), it allows the software engineer to access and manipulate its Abstract Syntax Tree (AST) [McC15]. That is useful for extracting TLA$^+$ specifications from the source code.

Research Object

The research object of the thesis is the verification of Elixir programs by extracting TLA$^+$ specifications from the program's source code.

Aim and Objectives of the Research

Master's thesis aims to create a method for extracting the TLA$^+$ specification from the sequential parts of the system's source code that would help to reduce the gap between the specification and implementation.

Objectives for achieving the goal of the Master's thesis:

1. Determine the State of the Art for extracting specifications from the code. Identify and analyze the existing code-to-specification transformation and transformation validation methods for various programming and specification languages.

2. Propose a code-to-specification transformation method for extracting the TLA$^+$ specifications out of sequential parts of the Elixir programs.

3. Implement the proposed transformation rules for specification extraction from the Elixir source code.

4. Prove or model check validity and adequateness of the extracted specifications. The transformation result will be considered adequate if it will allow defining refinement mapping between the high-level specification of the algorithm in question and the extracted specification. We expect to show the validity of the transformation by keeping the rules as simple as possible or using other methods found during the State of the Art analysis in this area.

Research Hypothesis

It is possible to extract correct TLA$^+$ specifications from sequential Elixir code so that they can be used to show the relation between a high-level (human-written) specification of an algorithm and its implementation in the Elixir code.

Research Method

The research will be carried out using the formal qualitative research method. The extracted specifications will be proven correct or model-checked with the existing tools (TLAPS or TLC).

Publication of Research Results

The research was presented in *Lithuanian MSc Research in Informatics and ICT (lt. Lietuvos magistrantų informatikos ir IT tyrimai)* conference and published in the proceedings [BP23].

1 Extracting Specifications from Source Code

Using specifications and models to design complex systems is a standard approach in traditional engineering domains. For example, the construction of complex and critical software for aircraft or medicine cannot be imagined without various specialized system models. Models help understand a complicated problem and its potential solutions through abstraction. Therefore, it is clear that software systems, which are often among the most complex engineering systems, can benefit from using models and modeling techniques [Sel03].

1.1 Model-Driven Engineering

Model-Driven Development (MDD) or *Model-Driven Engineering (MDE)* is a software engineering approach that applies models and model technologies to raise the abstraction level. At that level, developers are creating and evolving the software in order to simplify and formalize the various activities and tasks that comprise the software life cycle [HT06]. This paradigm is based on the assumption that "Everything is a model" [Bez05]. Due to this reason, MDE relies on three main concepts: *metamodel*, *model*, and *model transformation*. A metamodel defines the model's possible element types and structures that conform to it, similar to the relationship between grammar and corresponding programs in the programming area. There exist two types of model transformations – *model-to-model* (i.e., Eclipse ATL [Bez08]) or *model-to-text* (i.e., Eclipse Acceleo [Ecla]).

Despite the many advantages that good high-level formal software specifications bring to software design, verification by reasoning techniques, validation by simulation and testing, and documentation are still rare in the software industry (except in the case of mission-critical systems). Restrictive time and money constraints under which software is developed and the dynamic nature of software evolution are the most commonly cited problems that raise many challenges in keeping design and documentation up to date. The need for good software specifications is further underlined by the fact that most software engineers need to work on a system that was not designed or developed by them and the growing demand to document and reimplement legacy software systems [FPM+19].

1.2 Model-Driven Reverse Engineering

Reverse engineering source code to formal models is called Model-Driven Reverse Engineering (MDRE). Favre [Fav04] defines MDRE as "producing descriptive models from existing systems that were previously produced somehow." The purposes of models can vary from generating formal specifications of the system to generating UML models, business rules, or evaluating quality [RFZ17].

Usually, the MDRE process consists of two steps [RFZ17]:

1. Model generation from the source artifacts of the system.

2. Achieving a specific goal (e.g., model checking, re-documenting, etc.) by exploiting the model.

In Model-Driven Reverse Engineering, the source code often is the only available source of wisdom, so MDRE solutions start from a system model with a low abstraction level (the source code) and try to build views at higher abstraction levels. Consequently, numerous system views are needed, each corresponding to a different model. The models' generation from source code and model transformations can be automated. After that, the produced models may be analyzed by domain experts or appropriate tools and used for levering the model-driven development.

There are two main groups of MDRE solutions [BCD+14]:

- **Specific** – reverse engineering a system from a single technology, having a predefined scenario in mind (e.g., MedaEdit+ [Met], Columbus [FBT+02]);
- **General** – creating the basis for any other type of manipulation in later steps of the reverse engineering process (e.g., Moose [Tec], MoDisco [BCD+14]).

1.2.1 Main Challenges

Based on the problems stated above, Model-Driven Reverse Engineering should be able to overcome the following challenges [BCD+14]:

- **To avoid information loss due to the heterogeneity of systems** – for ensuring the high quality of the overall MDRE process, it is crucial to be able to retrieve as many artifacts as possible from the systems that are often technically heterogeneous;
- **To improve comprehension of the systems** – one of the main goals of MDRE is to improve the understanding of complicated systems. It requires going beyond the provision of simple low-level representations and deriving higher-level abstractions that contain the most relevant information;
- **To manage scalability** – most of the systems are usually large and complicated. For this reason, MDRE techniques should be able to load, query, and transform extensive models;
- **To adapt/port existing solutions to different needs** – many MDRE solutions depend on technology or scenario. That means that they target a specific technology or reverse-engineering scenario. Thus, MDRE solutions should be generic and largely reusable in various contexts.

1.2.2 General Characteristics

Model-Driven Reverse Engineering should fulfill the following characteristics [BCD+14]:

- **Genericity** – an MDRE approach should be based on technology-independent standards (e.g., metamodels) and customizable model-based components;
- **Extensibility** – an MDRE approach should rely on a decoupling of the represented information (models) and the next steps of the process;
- **Automation** – an MDRE approach should be partially or totally automated;
- **Full/Partial Coverage** – source artifacts should include interrelated components at different abstraction levels;
- **Direct (re)use and integration** – the various elements of the MDRE approach and its results (i.e, models) should be designed for reuse.

1.2.3 Standardized Models

Object Management Group (OMG) launched the Architecture Driven Modernization (ADM) Task Force [Grob] due to the growing interest in the MDRE field. Its main objective is to propose standard metamodels useful for modernization projects (e.g., migrations from obsolete technologies to more recent ones).

The first metamodel that ADM introduced is the *Knowledge Discovery Metamodel (KDM)* [Groc], with the primary purpose of defining a shared and complete representation, which guarantees the interoperability of different tools, and efficiently supporting maintenance, evolution, assessment, and modernization activities. The model is defined with a level of detail where it is capable of representing the structural concepts of object-oriented and imperative languages (e.g., classes, methods, functions, modules) [RFZ17].

ADM also defined the *Abstract Syntax Tree Metamodel (ASTM)* [Groa] metamodel to better support source code analysis activities. It represents the Abstract Syntax Tree (AST) of virtually any programming language, allowing analysis tools to target the metamodel instead of the specific language's AST. The model defines what is called *Generic ASTM (GASTM)* (e.g., definitions that recurrently apply to Abstract Syntax Trees of most programming languages). The model also allows extensions, called *Specialized ASTM (SASTM)*, to handle features specific to a single programming language [RFZ17].

Even though it sounds splendid to have a standard model and use reverse engineering tools created by other researchers, the reuse of standardized models (e.g., KDM or ASTM) is limited. One of the reasons might be that these models have extensive specification documents, which span hundreds of pages. This factor may be discouraging for the use of standard models as, most often, it is more simple to incrementally define ad-hoc metamodel that meets specific requirements [RFZ17]. Another problem is that sometimes these standardized models have scalability issues when handling extensive models (i.e., MoDisco authors reported this problem [BCD+14]).

1.2.4 Framework

[BCD+14] proposes a three-layer Model-Driven Reverse Engineering framework architecture, which is presented in Figure 1.

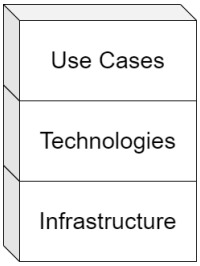

Figure 1. MDRE Framework Architecture.

Genericity and automation are provided via the *Infrastructure* layer as a set of basic bricks independent of any reverse engineering scenario. Such components offer generic metamodels and model transformations that have extensible model navigation, customization, and orchestration capabilities. In addition, they often come with the corresponding generic interfaces and extension features required for the components from the other layers to be plugged in [BCD+14].

The *Technologies* layer, which is built on top of the *Infrastructure* one, provides technology-dedicated components that stay independent from any specific reverse engineering scenario. Such components can be either technology-specific metamodels or their corresponding model discoverers and related transformations. They are the concrete bricks addressing the different kinds of systems to be potentially reverse engineered [BCD+14].

The final *Use Cases* layer provides some reuse and integration examples, which can be relatively simple demonstrators or more complete ready-to-use components implementing a given reverse engineering process. Such components are used for the realization of actual integration between components from the *Infrastructure* and *Technologies* layers. They can be either reused as-is or extended/customized for a different scenario [BCD+14].

1.3 Reverse Engineering Higher-Level Abstractions

In order to alleviate the concerns stated above, since early on [CC90] a significant amount of effort has been put into reverse engineering higher-level abstractions from existing systems. The action of automatically obtaining raw models from the system to be reverse-engineered is called Model Extraction [BCD+14].

Figure 2 shows what elements are involved in the model extraction from code that conforms to a grammar. This process is a text-to-model transformation T with its input a program P that conforms to the grammar definition G. The transformation uses P as either an Abstract Syntax Tree or a Concrete Syntax Tree. The execution of T builds a target model M_T that conforms to a target metamodel MM_T. It represents the information to be extracted, which is usually more complex than a syntax tree. The extraction process is

driven by a specification of the mappings between the grammar elements and the metamodel elements [CC90].

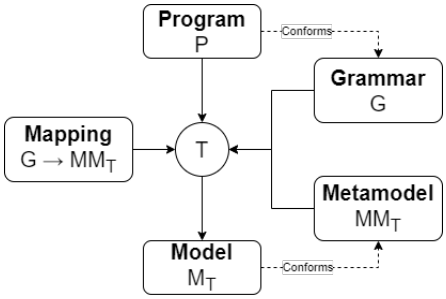

Figure 2. Model extraction process from source code.

There are two types of model discovery implementation – *two-step approach* and *direct approach*. In a two-step approach, a standardized model (e.g., GASTM) is firstly obtained from the system and then transformed into a language-specific model (e.g., TLA$^+$ model). In the direct approach, the language-specific model is built directly from the system's abstract syntax tree.

Having the process split into two steps reduces the complexity of each step and can be reused in similar MDRE applications with ease. On the contrary, if these benefits are not relevant to the user (e.g., he does not want the tool to be reusable in other approaches), one can decide to go with the direct approach as it involves fewer artifacts and offers better performance. Nevertheless, both approaches should obtain the model of the represented system at the same level of abstraction [BCD$^+$14].

1.4 Assessing Correctness of the Generated Model

The working principle of the model generator is similar to the compiler's, which translates the source code to the machine code. Thus, the same ways of assessing the correctness of generated artifacts should apply. [PA19] offers advice on how a compiler's correctness can be verified, which is presented in Figure 3.

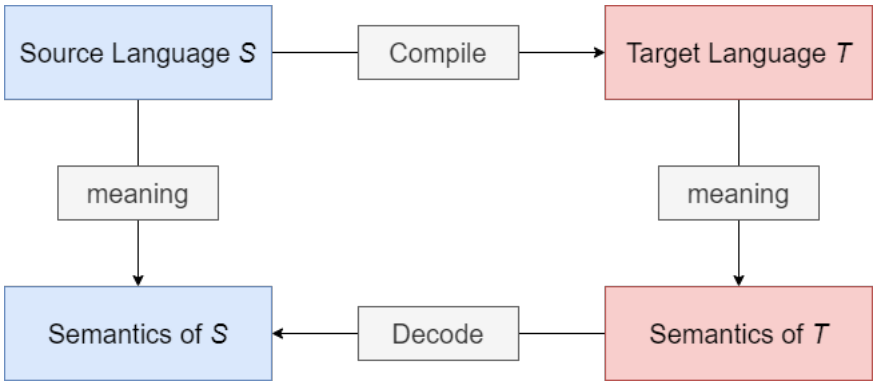

Figure 3. Process for assessing the correctness of the compiler.

The source language S is given a meaning by the source semantics and the target language T is given a meaning by the target semantics. When compiling S to T, the two

semantics are related. If the semantics of the target program that was produced by the compiler *relates* to the semantics of the source program (expressed by *decode* function), the compiler might be considered as correct. It might not be identical, but should be "somehow compatible with the semantics of source code" [PA19].

During the specification generation, some less-relevant data (e.g., code comments) is sliced away. This means that some of the code elements will not be preserved after the decoding generated specification, but that will not change the algorithm's behavior. Thus, the source code and decoded generated specification should be equivalent.

One of the ways to verify that both artifacts are equivalent is the *Translation validation* [PSS98] can be used. The concept of this process is depicted in Figure 4.

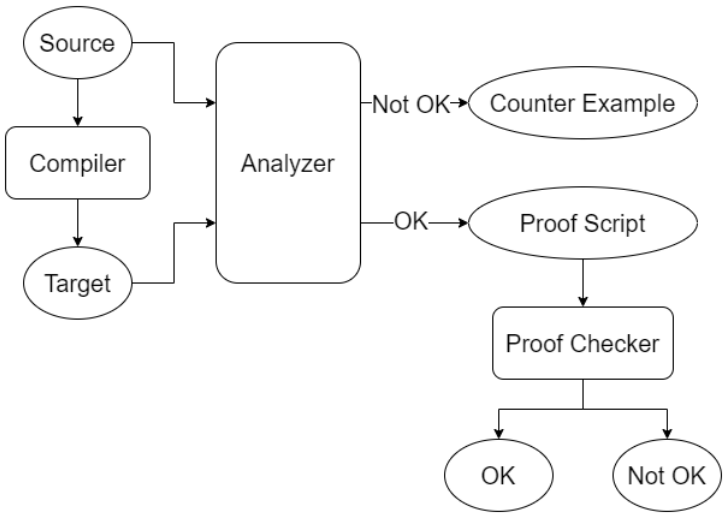

Figure 4. The concept of Translation Validation.

Both the *Source* and *Target* programs are passed as inputs to the *Analyzer* program. If it finds that the generated program correctly implements the source program, it generates a detailed proof script, which is examined and confirmed by the *Proof Checker*. If the analyzer fails to find correspondence between the source and target programs, it produces a counter-example. It consists of a scenario in which the generated code behaves differently than the source code. This means that the compiler is faulty and needs to be fixed [PSS98].

1.5 Approaches for Extraction

There exist numerous ways to extract the model from the source code. In this work, we review only the most popular ones.

1.5.1 Dedicated Parsers

The most frequently chosen strategy is creating a dedicated parser [IM12]. A dedicated parser can provide a specific solution that performs parsing and model generation tasks when given a grammar and a target metamodel. The parser is in charge of extracting an abstract

or concrete syntax tree from the source code, and the model generator traverses this syntax tree to generate the target model [IM12].

Developing a dedicated parser is a time-consuming and expensive task because the syntax tree traversals must be hardcoded to collect scattered information and resolve references. In addition, mappings are also hardcoded, which burdens maintainability. The effort required is usually alleviated by automatically extracting an AST from the source code [IM12].

1.5.2 DSL Definition Tools

The definition of textual DSLs aimed to express models in MDD solutions is another scenario in which a grammarware-modelware mapping is necessary. Since textual DSL definition tools must implement one of these mappings in order to provide the functionality of converting DSL programs into models and vice versa. These tools generate a dedicated parser and a DSL editor from the specification of the DSL's abstract and concrete syntaxes. They may therefore be considered as an alternative to developing a dedicated parser [IM12].

These tools support two approaches in order to specify both the abstract and the concrete syntaxes. In grammar-based tools, such as Xtext [Eclb] and TEF [Sch08], the developer uses an EBNF-like notation to specify both the grammar, including rules intended to specify the mapping for the corresponding metamodel and the concrete syntax. In some cases, such as in Xtext, the metamodel can also be automatically generated from the specification. On the other hand, metamodel-based tools, such as EMFText [Dev] and TCS [MLH+06], have as input a metamodel with annotations that specify the concrete syntax, and the grammar is automatically generated from this annotated metamodel [IM12].

1.5.3 Program Transformation Languages

Program transformation languages, such as Stratego/XT [Str] and TXL [NSE], could be used to extract models from source code by expressing the abstract syntax as context-free grammar rather than a metamodel. However, when such languages are used, the following limitations are encountered [IM12]:

1. A program transformation execution is a program conforming to a grammar. A tool for bridging grammarware and modelware would still be needed to obtain the model conforming to the target metamodel.

2. Grammar reuse is not promoted because each toolkit uses its grammar definition language. Moreover, each toolkit only provides a limited number of GPL grammars.

1.6 Existing Tools

As this is not a new problem in the software engineering field, there exist numerous tools that allow code-to-specification transformations.

1.6.1 MoDisco

MoDisco (Model Discovery) is an open-source Eclipse project for model-driven reverse engineering of IT systems. The primary goal of this tool is to provide support for activities dealing with legacy systems and ranging from understanding and documentation to evolution, modernization, and quality assurance [BCD+14]. The ADM Task Force of OMG cites MoDisco as an implementation example of its standards KDM, Structured Metrics Metamodel (SMM), and ASTM [RFZ17].

MoDisco approach consists of two main steps – *model discovery*, which builds the initial abstract model of the system, and *model understanding*, which analyzes the initial models and generates derived models to obtain the desired view of the system [BCD+14].

In order to fulfill the main characteristics of a Model-Driven Reverse Engineering system (Section 1.2.2), MoDisco follows a strategy of switching from a heterogeneous world of systems to a more homogeneous world of models immediately. The main principle that MoDisco uses is to quickly get initial models representing the system's artifacts without losing any of the information required for the process. These models are used as a starting point for the considered MDRE scenario. They are sufficiently accurate for that but do not represent any actual increase in the abstraction or detail levels. The resulting models can be seen as (full or partial) abstract syntax graphs that bridge the syntactic gap between the worlds (i.e., technical spaces) of source code, configuration files, etc., and the world of models. From this point on, any reverse engineering task on the system can be executed by using these models as a valid input representation [BCD+14].

1.6.2 Gra2Mol

Gra2MoL (Grammar-to-Model Transformation Language) is a Domain-Specific Language (DSL) tailored to the extraction of models from general-purpose programming languages (GPL) code. This DSL is a text-to-model transformation language that can be applied to any code conforming to grammar [IM12].

Figure 5 shows an example of how Gra2MoL is used. A Gra2MoL definition consists of a set of rules, where every rule expresses the mapping between a grammar element and a model element. The Gra2MoL definition shown in the example is very simple, and only contains the rule named `example` which transforms a `methodDeclaration` grammar element into the Method metamodel element according to the *from* and *to* parts of the rule. The *mapping* part expresses how the information of the model element is gathered from the information in the syntax tree. In this example, the name attribute of the `Method` model element is first initialized by accessing the `Name` grammar element of the `methodDeclaration` grammar element received by the rule (variable `mDec`). The `params` reference is then initialized by using the `q1` query, which collects every `param` grammar element representing the parameters of the method. Note that mappings are specified explicitly, and a specific query language is used to traverse the syntax tree [IM12].

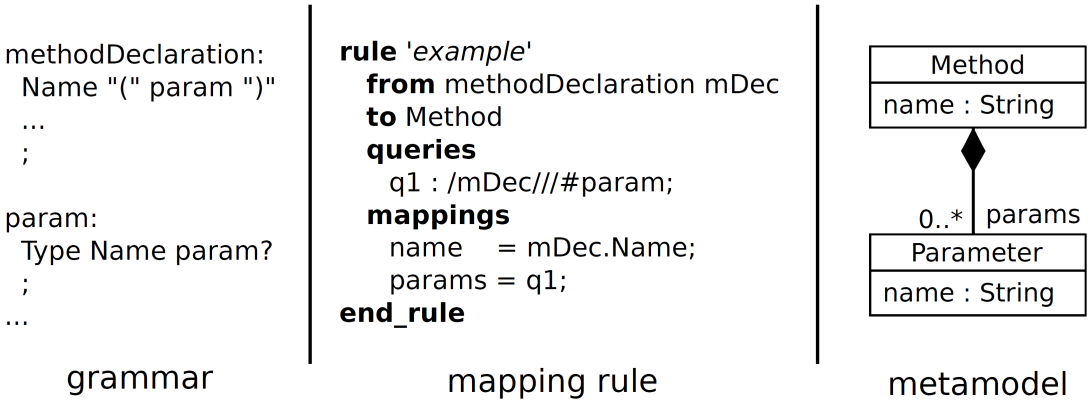

Figure 5. Simple example of a Gra2MoL mapping definition [IM12].

1.6.3 C2TLA$^+$

C2TLA$^+$ is the specifications extractor for C programs. The specification and verification process is illustrated in Figure 6. The first step of the process is to translate from an implementation provided by one or more *.c* files to a TLA$^+$. Before translation, the C files are parsed and normalized according to CIL (C Intermediate Language). Normalization to CIL makes programs more amenable to analysis and transformation. After obtaining the AST of the C program, C2TLA$^+$ generates the TLA$^+$ specification. The whole system is composed of TLA$^+$ modules resulting from C translation or manual specifications that come from different sources [MLH$^+$14].

After all the modules are integrated to form the complete specification given to TLC to generate the model and verify the properties (or refinements), if a property is not satisfied, TLC reports a trace that leads to the incorrect state. TLC also provides coverage information, i.e., the number of times each action was "executed" to construct a new state. Using this information, it is possible to identify actions that are never "executed" and might indicate an error in the specification. Both the trace and coverage information can be translated back to C [MLH$^+$14].

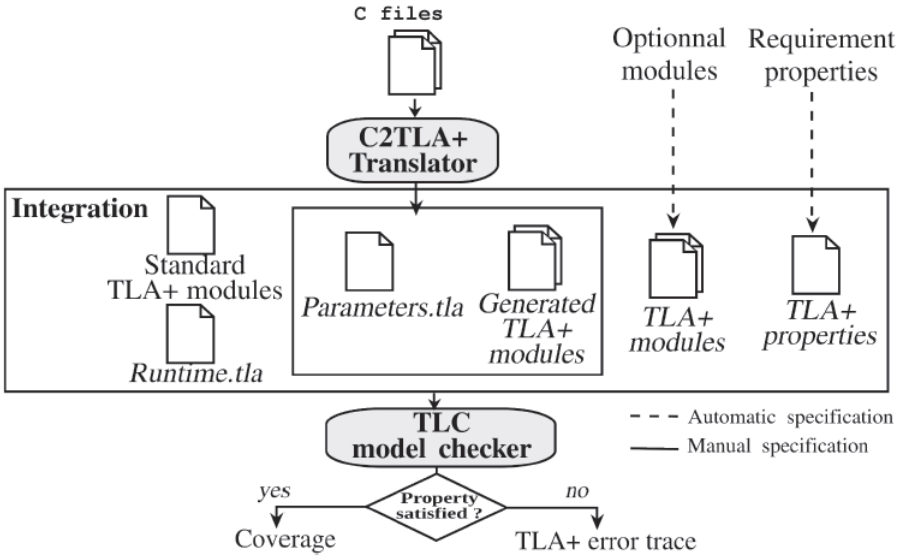

Figure 6. Specification and Verification process of C2TLA$^+$ [MLH$^+$14].

1.6.4 Bandera

Bandera is a component-based Java programs model extractor. One of the primary goals of *Bandera* is to provide automated support for the model construction and error trace interpretation techniques [CDH$^+$00].

Specifically, *Bandera* uses slicing to automate irrelevant component elimination, abstract interpretation to support data abstraction, and a model generator that allows significant flexibility in setting bounds for various system components. The tool also includes a collection of data structures for automatically mapping model-checker error traces back to the source level and facilities for graphical navigation of these traces. *Bandera* relies on conventional dataflow, control-flow, and dependency analyses, slicing and specialization transformations, as well as several supplementary analyses to produce compact models [CDH$^+$00].

Bandera uses a component-based architecture for model extraction to maximize scalability, flexibility, and extensibility. The architecture of *Bandera* is similar to an optimizing compiler. Compilers use multiple intermediate languages to stage the transformation to machine code, *Bandera* uses multiple intermediate languages to stage the transformation from Java to model-checker input languages. Just as a conventional compiler relies on sophisticated static analyses and transformations to produce optimized code, Bandera relies on conventional dataflow, control flow, dependency analyses, slicing, and specialization transformations [CDH$^+$00].

1.6.4.1 Soot/Jimple

Bandera is built on top of the Soot compiler framework, which translates Java programs to the intermediate language Jimple - one of several intermediate languages supported by Soot. The authors of *Bandera* developed a Java-to-Jimple-to-Java compiler (JJJC) for maintaining a close correspondence between a Java source code and Jimple representation.

Given a node in a program's Jimple representation, JJJC can return the corresponding node in the Java abstract syntax tree for the program, and vice versa [CDH+00].

1.6.4.2 Slicer

Given a program P and some statements of interest $C = \{S_1, ..., s_k\}$ from P called the slicing criterion, a program slicer will compute a reduced version of P by removing statements of P that do not affect the computation at the criterion statements C. When checking a program P against a specification ϕ, *Bandera* uses slicing to remove the statements of P that do not affect the satisfaction of ϕ. Thus, the specification ϕ holds for P if and only if ϕ holds for the reduced version of P (i.e. the reduction of P is sound and complete with respect to ϕ) [CDH+00].

1.6.4.3 Abstraction-Based Specializer

The Bandera Abstraction-Based Specializer (BABS) provides automated support for reducing model size via data abstraction. This is useful when a specification to be checked does not depend on the program's concrete values but instead depends only on the properties of those values. With an appropriate definition of abstraction, the specialization engine will transform the source code into a specialized version where all concrete operations and tests are replaced with abstract versions that manipulate tokens representing the abstract values [CDH+00].

1.6.4.4 Back End

The *Bandera* back end is like a code generator, taking the sliced and abstracted program and producing verifier-specific representations for targeted verifiers. The back end accepts a restricted form of Jimple and produces BIR (Banderas Intermediate Representation). For each supported model checker, there is also a translator component that accepts the program represented in BIR and generates input for that model checker [CDH+00].

BIR is a guarded command language for describing state transition systems. The primary purpose of BIR is to provide a simple interface for writing translators for target verifiers-to use *Bandera* with a new verifier, one must only write a translator from BIR to the input language of the verifier [CDH+00].

1.6.5 Stepwise Abstraction Extractor

Another approach to generating code directly from AST into formal specification is to derive formal software specifications by a sequence of (semi) automated transformations. Every transformation increases the level of abstraction of the previous specification in the sequence. This process can be done in a (semi) automated way and thus result in a valuable tool to improve the current software (reverse) engineering practices. The simple but critical observation in this regard is that the process of stepwise refinement from high-level

specification down to implementation provided by the ASM method can be applied in reverse order. Thus, it may be used for the (semi) automated extraction of high-level software specifications from source code [FPM+19].

The stepwise abstraction method consists of two phases – *Ground specification extraction* and *iterative high-level specification extraction* [FPM+19].

1.6.5.1 Ground Specification Extraction

Ground specification extraction is the first step of parsing the system's source code to translate it into a behaviorally equivalent ASM. Here the term behaviorally equivalent is used in the precise sense of the ASM thesis, i.e., in the sense that behaviorally equivalent algorithms have step-by-step the same runs. Thus the ground specification is expected to have the same core functionality as the implemented system [FPM+19].

This phase has two steps in it [FPM+19]:

1. By means of eKnows[1], the source code is parsed into a generic AST representation. Besides concrete language syntax, this intermediate representation also abstracts from language-specific semantics regarding control flow.

2. Rules for rewriting AST nodes related to control flow (e.g., loop, conditional statements) and assignment statements are provided. This eliminates intermediate variables and constructs assignment statements that only contain input/output parameters of the analyzed algorithms.

1.6.5.2 Iterative High-Level Specification Extraction

After the ground specification extraction phase is completed, the ground ASM specification is used as a base to extract higher-level specifications through a semi-automated iterative process. The implementation of the method must, at this point, present the user with different options to abstract away ASM rules and data [FPM+19].

1.6.5.3 Advantages of Stepwise Abstraction Method

The stepwise extraction of specifications via Abstract State Machines has the following benefits [FPM+19]:

- Precise yet succinct and easily understandable specifications at desired levels of abstraction;

- Each abstraction/refinement step can be proven correct if needed. This enables, for instance, proving that the implementation satisfies the requirement;

[1] https://www.scch.at/scch/presse-medien/detail/software-analyse-habt-in-programmen-verstecktes-wissen

- Only the first abstraction from source code to ASM rules depends on the programming language of the implementation. Subsequent abstractions only rely on general principles and transformations of ASM rules;
- The initial abstraction from source code to ASM can potentially be done entirely automatically via rewriting rules;
- Can be used for reverse engineering/understanding source code;
- Can be used for producing finite state machines for model-based testing. For instance, using a refinement of the extracted high-level ASM models to finite state machines.

2 Elixir in Distributed Computing

Elixir is a dynamic, functional language for building scalable and maintainable applications [Tea].

The motivation for generating code from Elixir source code is based on several reasons:

1. Reliable concurrency and inter-process communication from Erlang's virtual machine (BEAM) usage, which is essential to support concurrent and distributed systems that are a focus of TLA$^+$.

2. The function paradigm is declarative over operational, similar to the TLA$^+$ syntax. This helps to keep correspondence between definitions in each language.

3. BEAM provides platform transparency, making the generated code suitable for a bigger number of environments.

4. Relatively new programming language with modern syntax.

Now we will analyze an algorithm *Wasper*[2] to understand better what Elixir language constructs are used in the development of distributed computing algorithms.

2.1 Type Definitions

Defining custom types helps to communicate the intention of the code and increases its readability. Custom types can be defined within the modules via the `@type` attribute. Custom types provide a more descriptive alias of the type and are helpful when used together with function specifications (see Section 2.3). Custom types in Elixir are implemented via the `map` structure [Tea].

The syntax of type definition is to put the custom type name on the left side of the `::` and the properties of the type on the right side. An example of struct type definition can be seen in Listing 1.

```
@type t(obj) :: %GPA{
      mod: module(),
      obj: obj,
      me: any,
      process_own: boolean()
    }
```

Listing 1: Custom type definition in Elixir.

[2] `https://github.com/kape1395/wasper` (the repository is private at the time of writing).

2.2 Value Updating

As variables in Elixir are immutable (variable's value cannot change during the execution of a program), in order to update a value, it should be reassigned to the variable. As update is a common operation, Elixir has a syntactic sugar for this, using `|` operator [Tea]. This operation can be seen throughout the *Wasper* algorithm (see Listing 2).

```
gpa = %{gpa | obj: obj}
```

Listing 2: Struct value update in Elixir.

2.3 Function Specifications

Function specifications are used for documenting the function signatures. In code, function specifications are written with the `@spec` attribute, typically placed immediately before the function definition. Specifications can describe both public and private functions. The function name and the number of arguments used in the `@spec` attribute must match the function it describes. Function specifications can use both built-in types provided by Elixir, like integer or boolean, and user-declared types [Tea].

The syntax of function specification is to put the function and its input on the left side of the `::` and the return value's type on the right side. An example of function specification can be seen in Listing 3.

```
@spec input(t(obj), any()) :: ret(t(obj)) when obj: any()
def input(gpa = %GPA{mod: mod, obj: obj}, input) do
    handle_result(gpa, mod.handle_input(obj, input))
end
```

Listing 3: Function signature specification via the attribute in Elixir.

2.4 Pattern Matching

Another commonly used construct is pattern matching. It allows developers to easily destructure data types such as tuples, lists, or maps and is one of the foundations of recursion in Elixir [Tea]. In *Wasper* algorithm, pattern matching is used often used for destructing inputs of a function (see Listing 4).

```
def message(gpa = %GPA{mod: mod, obj: obj}, message) do
    handle_result(gpa, mod.handle_message(obj, message))
end
```

Listing 4: Pattern matching function's input parameters in Elixir.

2.5 Guards

In Elixir, guards are a way to augment pattern matching with more complex checks. They are allowed in a predefined set of constructs where pattern matching is allowed. Guard

clause places further restrictions on the parameters in a function, such as the data type or allowed number range. Guards can be used in functions, case expressions, anonymous functions, and other places. If no guard clause is matched, the exception is raised [Tea]. An example of guard usage in the *Wasper* algorithm can be seen in Listing 5.

```
defp combined_msgs(peers, msgs, outputs) when map_size(msgs) == 0 do
    ...
end

defp combined_msgs(peers, msgs, outputs) do
    ...
end
```

Listing 5: Function guards in Elixir.

2.6 `Case` Statements

A `case` statement can be considered as a replacement for the `switch` statement in imperative programming languages. `Case` takes a variable/literal and applies pattern matching to it with different cases. If any case matches, Elixir executes the code associated with that case and exits the case statement. If no match is found, it exits the statement with an `CaseClauseError` that displays no matching clauses were found [Tea]. An example of the `case` statement in *Wasper* algorithm can be seen in Listing 6.

```
case step do
    {:next, next_est} when target_r == r -> ...
    {:next, next_est} when target_r > r -> ...
    :same -> ...
end
```

Listing 6: `Case` statement in Elixir.

2.7 Comprehensions

In Elixir, it is common to loop over an Enumerable, often filtering out some results and mapping values into another list. Comprehensions are syntactic sugar for such constructs – they group those common tasks into the `for` special form. Comprehensions can return the result as a list or can insert into the different data structures by passint the `:into` option to the comprehension [Tea]. The latter way is used more often in the *Wasper* distributed algorithm (see Listing 7)

```
msgs = for p <- peers, into: %{}, do: {p, [{:READY, i, m, h}]}
```

Listing 7: Comprehension in Elixir.

2.8 Processes

In Elixir, all code runs inside processes. Processes are isolated from each other, run concurrent to one another and communicate via message passing. Processes are not only the basis for concurrency in Elixir, but they also provide the means for building distributed and fault-tolerant programs [Tea].

Elixir's processes should not be confused with operating system processes. Processes in Elixir are extremely lightweight in terms of memory and CPU. Because of this, it is not uncommon to have tens or even hundreds of thousands of processes running simultaneously [Tea].

Wasper uses `send` and `receive` functions for communication and specifies a timeout with `after` function, which will execute if no messages arrive in specified amount of time (see Listing 8).

```
receive do
  {:stdout, ^os_pid, more} ->
    read(n - byte_size(partial),
    %{state | read_buf: more},
    [partial | acc])

after
  10000 -> {:error, :timeout}
end
```

Listing 8: Data receive process in Elixir.

2.9 Abstract Syntax Tree

Elixir supports metaprogramming, which allows manipulating the programs dynamically – this is achieved by manipulating *abstract syntax tree* (AST), which is the representation of code as a tree-based data structure. Elixir uses this data structure to run Elixir code: either by interpreting it directly: executing the commands in the AST by recursively walking it; or by compiling it: translating the AST into another format, namely BEAM bytecode instructions, which then are saved to disk as `.beam` files. In runtime, these `.beam` files are then loaded and executed more efficiently than the interpreter can [Sch19]. This process is presented in Figure 7.

Elixir AST node is represented by a tuple of 3 elements `{marker, metadata, children}`. *Marker* is some kind of an atom (an atom is a constant whose value is its own name. In some languages it is simply a symbol) and *metadata* is a keyword list that contains annotations about the metadata node (line number, file, column number, etc.). The *children* is either a list of child AST nodes or an atom 7, The Elixir expression `2 * x` looks like this `{:*, [], [2, {:x, [], nil}]}` in Elixir AST.

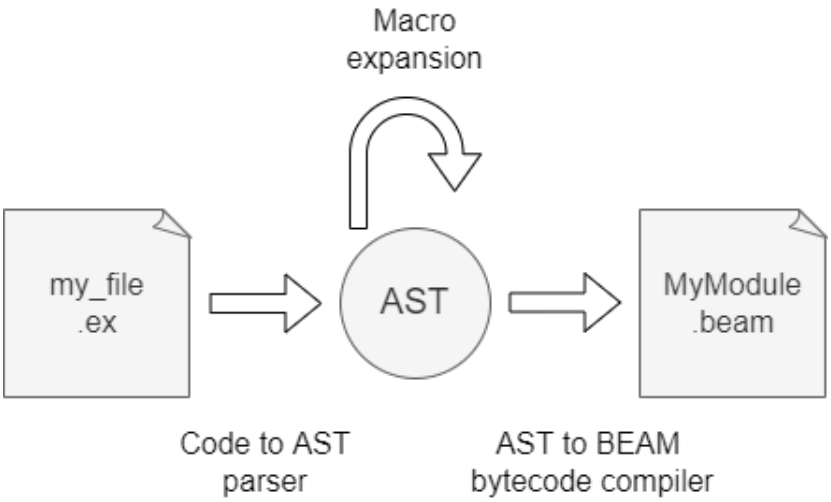

Figure 7. How Elixir compiler uses AST.

AST is a regular Elixir data structure. It can be parsed and manipulated by the code – it is possible to write a parser that walks the tree and uses pattern matching to pluck specific chunks of code and manipulate or evaluate them.

An AST of the function from Listing 4 is presented in Listing 9. Note that for the reader's convenience, the metadata node is replaced with an underscore (_) as, in this case, it does not contain any useful information.

```
{:def, _, [
   {:message, _, [
      {:=, _, [
         {:gpa, _, nil},
         {:%, _, [
            {:__aliases__, _, [:GPA]},
            {:%{}, _, [
               mod: {:mod, _, nil},
               obj: {:obj, _, nil}]}]}]},
      {:message, _, nil}]}, [
     do: {:handle_result, _, [
        {:gpa, _, nil},
        {{:., _, [
            {:mod, _, nil}, :handle_message]}, _, [
           {:obj, _, nil},
           {:message, _, nil}]}]}]]}
```

Listing 9: AST of the `message` function from *Wasper*.

3 Overview of TLA$^+$ Syntax

TLA$^+$ [Lam02] is the specification language of the Temporal Logic of Actions (TLA). TLA is a variant of linear temporal logic introduced by Leslie Lamport for specifying and reasoning about concurrent systems.

The main principles of TLA$^+$ will be presented by analyzing *Bracha* reliable broadcast protocol.

3.1 Bracha RBC Protocol

Bracha's Reliable Broadcast is one of the most important build blocks for Byzantine Fault Tolerant protocols in the Asynchronous model. In this standard setting, there are n parties, where one is designated as the *leader*. The malicious threshold adversary can control at most $f < n/3$ parties. The leader has some input value M, and a party that terminates needs to output a value [Bra87].

The protocol satisfies the following properties:

- *Agreement* – if an honest node outputs a message `M` and another honest node outputs a message `M'`, then `M` = `M'`;
- *Validity* – if the broadcaster is honest, all honest nodes eventually output the message;
- *Totality* – if the honest node outputs a message, then every honest node eventually outputs a message.

We have used the algorithm presented in the [DXR21] article as it doesn't imply sending ECHO messages upon receiving *T+1* `READY` messages. The pseudo-code from [DXR21] is presented in Algorithm 1. The t means that "Given a network of n nodes, of which up to t could be malicious".

Algorithm 1 Pseudo code of Bracha RBC [DXR21].

```
1:  // broadcaster node only
2:  input M
3:  send ⟨PROPOSE, M⟩ to all nodes
4:  // all nodes
5:  input P(·) // predicate P(·) returns true unless otherwise specified
6:  upon receiving ⟨PROPOSE, M⟩ from the broadcaster do
7:      if P(M) then
8:          send ⟨ECHO, M⟩ to all nodes
9:  upon receiving 2t + 1 ⟨ECHO, M⟩ messages and not having sent a READY message do
10:     send ⟨READY, M⟩ to all nodes
11: upon receiving t + 1 ⟨READY, M⟩ messages and not having sent a READY message do
12:     send ⟨READY, M⟩ to all nodes
13: upon receiving 2t + 1 ⟨READY, M⟩ messages do
14:     output M
```

The protocol uses three types of messages: `PROPOSE`, `ECHO`, and `READY`. A `(PROPOSE, M)` message means that process p wishes to broadcast the value `M`. A `(ECHO, M)` message means that its sender knows that p sent `M` because it received `(PROPOSE, M)` message from p. A `(READY, M)` message means that its sender knows that M is the only value sent by p and that it is ready to accept `M` because it received enough `(ECHO, M)` or `(READY, M)` messages. When a process receives enough `(READY, M)` messages, it Accepts `M` as the value sent by p, knowing that all other correct processes are bound to accept `M` too [Bra87].

The protocol is divided into steps corresponding to the message types. In each step, a process waits until it receives enough messages that permit it to send the next message type (including those received at previous steps), then it sends a message to all the processes and moves to the next step [Bra87].

The protocol has the following steps:

- *Broadcast*, which sends the message `(PROPOSE, M)` to all nodes;
- *Receive Propose*, which sends the message `(ECHO, M)` once p receives `(PROPOSE, M)` message;
- *Receive Echo*, which sends a `(READY, M)` message (if that was not sent yet) once p receives $2t + 1$ `(ECHO, M)` messages;
- *Receive Ready Support*, which sends a `(READY, M)` message (if that was not sent yet) once p receives $t + 1$ `(READY, M)` messages;
- *Receive Ready Output*, which sets output to M once p receives $2t + 1$ `(READY, M)` messages.

3.2 Bracha RBC Protocol's TLA$^+$ Specification

The implementation of a TLA$^+$ specification for a Bracha reliable broadcast protocol (see section 3.1) can be found in a Wasp (node software developed by the IOTA Foundation to run the *IOTA Smart Contracts* (ISC in short) on top of the *IOTA Tangle* repository [3].

The specification implements the protocol's steps, defined in Section 3.1. Model checking is used to check whether the specified protocol satisfies the required properties.

Note that the TLA$^+$ specification of Bracha RBC uses simple TLA$^+$ constructs and does not cover complex scenarios, but should be enough to explain the main principles.

3.3 Declarations, Definitions, and Assumptions

TLA$^+$ specifications are partitioned into modules. The beginning of the module's body consists of declarations, definitions, and assumptions. They are presented in Figure 8.

[3] `https://github.com/iotaledger/wasp/blob/develop/packages/gpa/rbc/bracha/BrachaRBC.tla`

Usually, bigger specifications are made up of a number of modules. One module can be included in another by using `EXTENDS` statement at the beginning of the module [Lam02]. Bracha RBC specification uses `FiniteSets` and `Naturals` modules that are standard modules of the TLA$^+$.

In TLA$^+$, a declaration is a statement that introduces a new symbol or variable into the specification. Two types of declarations are used most frequently – *constants* and *variables*. Constant declarations introduce a new constant symbol with a fixed value that remains the same throughout the specification. The value of the constant is defined in the model checking configuration instead of the specification. Variable declarations introduce a new variable that can take on different values during execution. Usually, all variables of the module are listed inside the `vars` tuple (the symbol $\triangleq$ means *equal by definition*) [Lam02].

In TLA$^+$, a definition is a statement that assigns a name to a TLA+ expression or formula and can be used as a shorthand for that expression or formula throughout the specification [Lam02]. The Bracha RBC specification has multiple definitions in order to make the specification simpler and more readable. For instance, `TypeOK` definition checks whether the specified variables are correct. This definition is invoked as invariant – it should be satisfied at every step.

In TLA$^+$, an assumption is a statement that is taken to be true without proof and is used to simplify the specification or reasoning about it. Assumptions in TLA$^+$ are similar to axioms or postulates in other mathematical systems and are often used to define the basic properties of the system being modeled or to introduce additional constraints or assumptions that simplify the specification [Lam02]. The Bracha RBC specification contains multiple assumptions, one of them being *NodeAssms*. It defines that sets of correct and faulty nodes are finite, that no node is correct and faulty at the same time, and that there is at least one correct node.

MODULE *BrachaRBC*

EXTENDS *FiniteSets*, *Naturals*

Constants

CONSTANT CN Correct nodes.
CONSTANT FN Faulty nodes.
CONSTANT $Value$ The broadcasted value.

Variables

VARIABLE $bcNode$ The broadcaster node.
VARIABLE $bcValue$ Value broadcasted by a correct BC node.
VARIABLE $predicate$ Predicates received by the nodes.
VARIABLE $output$ Output for each node.
VARIABLE $msgs$ Messages that were sent.

$vars \triangleq \langle bcNode, bcValue, predicate, output, msgs \rangle$

Definitions

$AN \triangleq CN \cup FN$ All nodes.
$N \triangleq Cardinality(AN)$ Number of nodes in the system.
$F \triangleq Cardinality(FN)$ Number of faulty nodes.
$Q1F \triangleq \{q \in \text{SUBSET } AN : Cardinality(q) = F + 1\}$ Contains ≥ 1 correct node.
$Q2F \triangleq \{q \in \text{SUBSET } AN : Cardinality(q) = 2 * F + 1\}$ Contains $\geq F + 1$ correct nodes.
$QNF \triangleq \{q \in \text{SUBSET } AN : Cardinality(q) = N - F\}$ Max quorum.
$QXF \triangleq \{q \in \text{SUBSET } AN : Cardinality(q) = ((N + F) \div 2) + 1\}$ Intersection is $F + 1$.

$NotValue \triangleq \text{CHOOSE } v : v \notin Value$
$Msg \triangleq [t : \{\text{“PROPOSE”}, \text{“ECHO”}, \text{“READY”}\}, src : AN, v : Value]$
$HaveProposeMsg(n, vs) \triangleq \exists\, pm \in msgs : pm.t = \text{“PROPOSE”} \land pm.src = n \land pm.v \in vs$
$HaveEchoMsg\ (n, vs) \triangleq \exists\, em \in msgs : em.t = \text{“ECHO”} \land em.src = n \land em.v \in vs$
$HaveReadyMsg\ (n, vs) \triangleq \exists\, rm \in msgs : rm.t = \text{“READY”} \land rm.src = n \land rm.v \in vs$

$TypeOK \triangleq$
- $\land\ msgs \subseteq Msg$
- $\land\ bcNode \in AN$
- $\land\ \lor bcNode \in CN \land bcValue \in Value$
  $\lor bcNode \in FN \land bcValue = NotValue$
- $\land\ predicate \in [CN \to \text{BOOLEAN}]$
- $\land\ output \in [CN \to Value \cup \{NotValue\}]$

Assumptions

ASSUME $QuorumAssms \triangleq N > 3 * F$
ASSUME $NodesAssms \triangleq$
- $\land\ IsFiniteSet(CN)$
- $\land\ IsFiniteSet(FN)$
- $\land\ CN \cap FN = \{\}$
- $\land\ CN \neq \{\}$

ASSUME $ValueAssms \triangleq \exists\, v \in Value : \text{TRUE}$

Figure 8. Declarations, definitions, and assumptions of the Bracha RBC TLA$^+$ specification.

3.4 Behaviors

The system is specified by a set of possible *behaviors* – a sequence of states, where the state is an assignment of values to variables, representing a correct execution of the system [Lam02]. The Bracha RBC specification has behaviors (see Figure 9) based on the steps presented in section 3.1.

As the structure of defined behaviors is similar, we'll analyze one of them – *RecvE-*

cho(eq). In order for behavior to take place, there should exist a correct node n and value v, which has received $2t + 1$ ECHO messages, and has not sent a READY message yet. If this condition is satisfied, the READY message is sent to all nodes. The last line of behavior lists the variables that have not been changed during the behavior.

Behaviors

$Broadcast \triangleq$
$\quad \wedge\ bcNode \in CN$ We only care on the behaviour of the correct nodes.
$\quad \wedge\ \neg HaveProposeMsg(bcNode,\ Value)$
$\quad \wedge\ msgs' = msgs \cup \{[t \mapsto \text{“PROPOSE”},\ src \mapsto bcNode,\ v \mapsto bcValue]\}$
$\quad \wedge\ \text{UNCHANGED}\ \langle bcNode,\ bcValue,\ predicate,\ output\rangle$

$UpdatePredicate \triangleq$
$\quad \exists\, n \in CN,\ p \in \text{BOOLEAN}\ :$
$\quad\quad \wedge\ predicate[n] \Rightarrow p$ Only monotonic updates are make the algorithm to terminate.
$\quad\quad \wedge\ predicate' = [predicate\ \text{EXCEPT}\ ![n] = p]$
$\quad\quad \wedge\ \text{UNCHANGED}\ \langle bcNode,\ bcValue,\ output,\ msgs\rangle$

$RecvPropose(pm) \triangleq$
$\quad \exists\, n \in CN :$
$\quad\quad \wedge\ predicate[n]$
$\quad\quad \wedge\ HaveProposeMsg(bcNode,\ \{pm.v\})$
$\quad\quad \wedge\ \neg HaveEchoMsg(n,\ Value)$
$\quad\quad \wedge\ msgs' = msgs \cup \{[t \mapsto \text{“ECHO”},\ src \mapsto n,\ v \mapsto pm.v]\}$
$\quad\quad \wedge\ \text{UNCHANGED}\ \langle bcNode,\ bcValue,\ predicate,\ output\rangle$

$RecvEcho(eq) \triangleq$
$\quad \exists\, n \in CN,\ v \in Value :$
$\quad\quad \wedge\ eq \in QXF$
$\quad\quad \wedge\ \forall\, qn \in eq : HaveEchoMsg(qn,\ \{v\})$
$\quad\quad \wedge\ \neg HaveReadyMsg(n,\ Value)$
$\quad\quad \wedge\ msgs' = msgs \cup \{[t \mapsto \text{“READY”},\ src \mapsto n,\ v \mapsto v]\}$
$\quad\quad \wedge\ \text{UNCHANGED}\ \langle bcNode,\ bcValue,\ predicate,\ output\rangle$

$RecvReadySupport(rq) \triangleq$
$\quad \exists\, n \in CN,\ v \in Value :$
$\quad\quad \wedge\ rq \in Q1F$
$\quad\quad \wedge\ \forall\, qn \in rq : HaveReadyMsg(qn,\ \{v\})$
$\quad\quad \wedge\ \neg HaveReadyMsg(n,\ Value)$
$\quad\quad \wedge\ msgs' = msgs \cup \{[t \mapsto \text{“READY”},\ src \mapsto n,\ v \mapsto v]\}$
$\quad\quad \wedge\ \text{UNCHANGED}\ \langle bcNode,\ bcValue,\ predicate,\ output\rangle$

$RecvReadyOutput(rq) \triangleq$
$\quad \exists\, n \in CN,\ v \in Value :$
$\quad\quad \wedge\ rq \in Q2F$
$\quad\quad \wedge\ \forall\, qn \in rq : HaveReadyMsg(qn,\ \{v\})$
$\quad\quad \wedge\ output[n] = NotValue$
$\quad\quad \wedge\ output' = [output\ \text{EXCEPT}\ ![n] = v]$
$\quad\quad \wedge\ \text{UNCHANGED}\ \langle bcNode,\ bcValue,\ predicate,\ msgs\rangle$

Figure 9. Behaviors of the Bracha RBC algorithm.

3.5 Behavioral Properties

Behavioral properties are the properties that specify what a system is supposed to do [Lam02]. They are specified in the same way as the definitions (see section 3.3).

In the Bracha RVC specification, five behavioral properties must be satisfied:

- Agreement – all honest nodes should have the same output;
- Validity – if the broadcaster is honest, all honest nodes eventually output the message M;
- Totality – if an honest node outputs a message, then every honest node eventually outputs a message;
- Single value from peer per message type – one single message from the honest node is enough;
- Propose message exists – verifies that PROPOSE message was sent.

Behavioral properties

$$HaveOutput(n) \triangleq output[n] \neq NotValue$$

Agreement. If an honest node outputs a message M' and another honest node outputs a message M", then $M' = M$".

$$\begin{aligned}&Agreement \triangleq \\ &\quad \forall\, n1,\, n2 \in CN : \\ &\qquad HaveOutput(n1) \land HaveOutput(n2) \Rightarrow output[n1] = output[n2]\end{aligned}$$

Validity. If the broadcaster is honest, all honest nodes eventually output the message M.

$$\begin{aligned}&Validity \triangleq \\ &\quad bcNode \in CN \leadsto \Box \forall\, n \in CN : output[n] = bcValue\end{aligned}$$

Totality. If an honest node outputs a message, then every honest node eventually outputs a message.

$$\begin{aligned}&Totality \triangleq \\ &\quad \forall\, v \in Value : \\ &\qquad (\exists\, n \in CN : output[n] = v) \leadsto \Box \forall\, n \in CN : output[n] = v\end{aligned}$$

Additionally: We can only receive a single message of a particular type from a correct peer. Thus, we can ignore the following messages and prevent an adversary from sending us a lot of messages to fill our memory.

$$\begin{aligned}&SingleValueFromPeerPerMsgType \triangleq \\ &\quad \forall\, m1,\, m2 \in msgs : (\\ &\qquad \land\, m1.src \in CN \\ &\qquad \land\, m1.src = m2.src \\ &\qquad \land\, m1.t = m2.t \\ &\quad) \Rightarrow m1.v = m2.v\end{aligned}$$

$$\begin{aligned}&ProposeMessageExists \triangleq \\ &\quad \lor \Diamond(\exists\, m \in msgs : m.t = \text{“PROPOSE”} \land m.src = bcNode \land m.v = bcValue) \\ &\quad \lor \{bcNode\} \subseteq FN\end{aligned}$$

Figure 10. Behavioral properties of the Bracha RBC algorithm.

3.6 Specification

Once all of the system's behaviors are described, the actual specification is described. The specification of the Bracha RBC algorithm is presented in Figure 11.

It begins with the *initial* predicate that specifies the possible initial values of earlier described variables [Lam02]. In the initial state of Bracha RBC specification, the broadcaster

node and broadcasted value are selected, the predicate is set to the boolean value, the output is set to *NotValue* for all nodes, and messages from faulty nodes are added for modeling purposes.

The *next-state* action specifies how these variables can change in any step. It is a formula expressing the relation between the values of variables in the old state and the new state (adorned with the *primed* operator ”′”). In the Bracha RBC algorithm, the *next-state* action defines that in every step, the executed behavior is a step from Figure 1.

After that, the *Liveness* predicate is defined. It uses *weak wairness* operator *WF*, which means that if an action can happen, then it will *eventually* happen. In this case, if *Next* is possible, it will eventually happen, guaranteeing the liveness.

The specification is defined by the temporal formula *Spec*, which is an assertion about the system's behaviors. In the Bracha RBC algorithm, with the initial state *Init*, the *next-state* action is always *true* the (denoted by temporal operator □) and satisfies the *Liveness* predicate.

Lastly, the *theorem* is defined, which asserts that the predicate can be proved from the definitions and assumptions of the current module. The `PROOF OMITTED` command means that the theorem will be checked by model checking.

The specification.

$Init \triangleq$
$\land\ bcNode \in AN$
$\land\ \lor\ bcNode \in CN \land bcValue \in Value$
$\quad \lor\ bcNode \in FN \land bcValue = NotValue$
$\land\ predicate\ \in [CN \to \text{BOOLEAN}\ ]$
$\land\ output = [n \in CN \mapsto NotValue]$
$\land\ msgs = [t : \{\text{“PROPOSE”}, \text{“ECHO”}, \text{“READY”}\}, src : FN, v : Value]$

$Next \triangleq$
$\lor\ Broadcast$
$\lor\ UpdatePredicate$
$\lor \exists\, pm \in msgs : RecvPropose(pm)$
$\lor \exists\, eq\ \in QXF\ : RecvEcho(eq)$
$\lor \exists\, rq\ \in Q1F\ : RecvReadySupport(rq)$
$\lor \exists\, rq\ \in Q2F : RecvReadyOutput(rq)$

$Liveness \triangleq$
$\land \text{WF}_{vars}(Broadcast)$
$\land \text{WF}_{vars}(UpdatePredicate)$
$\land \text{WF}_{vars}(\exists\, pm \in msgs : pm.src \in CN \land RecvPropose(pm))$
$\land \text{WF}_{vars}(\exists\, eq\ \in QXF\ : eq \subseteq CN \land RecvEcho(eq))$
$\land \text{WF}_{vars}(\exists\, rq\ \in Q1F\ : rq\ \subseteq CN \land RecvReadySupport(rq))$
$\land \text{WF}_{vars}(\exists\, rq\ \in Q2F : rq\ \subseteq CN \land RecvReadyOutput(rq))$

$Spec \triangleq Init \land \Box[Next]_{vars} \land Liveness$

THEOREM $Spec \Rightarrow$
$\land \Box TypeOK$
$\land \Box Agreement$
$\land \Box SingleValueFromPeerPerMsgType$
$\land Validity$
$\land Totality$
$\land ProposeMessageExists$

PROOF OMITTED Checked by the *TLC*.

Figure 11. Specification of the Bracha RBC algorithm.

3.7 Formal Specification Refinement

Stepwise refinement is a specification construction technique for developing a specification through a sequence of refinement steps – starting from the abstract model, each subsequent model refines the previous one. This is controlled through several proof obligations, which guarantee the correctness of the development. Such proof obligations are proved by automatic proof procedures supported by a proof engine. The essence of the refinement relationship is that it preserves system properties. Moreover, proofs of properties of these models help to convince that the system is correct since they demonstrate the behavior of the last, and the most concrete model respects the behavior of the first, most abstract model [CGM07].

The refinement may be helpful for model-checking generated specifications as they might be more abstract than manually written ones.

4 Overview of PlusCal Syntax

PlusCal is an algorithm language for writing and debugging algorithms, which is compiled to TLA$^+$. Leslie Lamport developed it in 2009 to make TLA$^+$ more accessible to programmers [Way18].

An algorithm written in PlusCal is debugged using the TLA$^+$ tools – mainly the TLC model checker. Correctness of the algorithm can also be proved with the TLAPS proof system, but that requires a lot of hard work and is seldom done [Lam].

Writing implementations in TLA$^+$ can be a barrier starting out, so PlusCal has additional constructs that make TLA$^+$ easier to learn and use. For example, the `:=` assignment syntax and the labels, give us a pseudocode-like structure on top of TLA$^+$. We describe the "P syntax" of PlusCal, which is closer to the Elixir programming language, but there is also a "C syntax" [Lam21].

A PlusCal algorithm appears inside a comment within a TLA$^+$ module. Its' structure is presented in Listing 10.

```
--algorithm name {
  (* declaration of global variables *)
  (* operator definitions *)
  (* macro definitions *)
  (* procedures *)
  (* algorithm body or process declarations *)
end algorithm; }
```

Listing 10: Top-level structure of PlusCal algorithm

Variable declarations, operator and macro definitions, and procedures are optional. There must either be an algorithm body (for a sequential algorithm) or process declarations (for a concurrent algorithm).

4.1 Declaration of Variables

Variables declaration is presented in Listing 11. There may be at most one `variables` clause, but that it may declare any number of variables. This example declares three variables `x`, `y`, and `z`. The variable `x` is not initialized, `y` is initialized to `0`, and may take either `1`, `2` or `3` as initial values. During model checking, TLC will assign `x` a default value that is different from all ordinary values and explore all three initial values for `z` [Way18].

```
variables x, y = 0, z \in {1,2,3};
```

Listing 11: Variables declaration in PlusCal algorithm

4.2 Operator Definitions

As in TLA$^+$, operators represent utility functions for describing the algorithm [Way18].

```
define
  Cons(x,s) == <<x>> \o s
  Front(s)  == [i \in 1 .. Len(s) - 1 |-> s[i]]
end define;
```

Listing 12: Operators declaration in PlusCal algorithm

For example, Listing 12 defines a declaration of *Cons* operator that prepends an element x to a sequence s and an operator *Front* that returns the subsequence of a non-empty sequence without the last element. As with the variables, there can be a single *define* clause, but it may contain any number of operator definitions.

4.3 Macro Definitions

A macro represents several PlusCal statements that can be inserted into an algorithm, and it may have parameters. In contrast to a defined operator, a macro need not be purely functional but may have side effects. In contrast to a procedure, it cannot be recursive and may not contain labels or complex statements such as loops or procedure calls, or return statements [Way18].

```
macro rcv(p, var)
  (* body expressions *)
end macro;
```

Listing 13: Macros declaration in PlusCal algorithm

4.4 Procedures

A procedure declaration is similar to that of a macro, but it may also contain the declaration of local variables. The procedure body may contain arbitrary statements, including procedure calls and even recursive calls [Way18].

```
procedure p(x,y)
variables ...  \* procedure-local variable
begin
    (* body expressions *)
    return;
end procedure;
```

Listing 14: Procedure declaration in PlusCal algorithm

Procedure parameters are treated as variables and may be assigned to them. Any control flow in a procedure should reach a `return` statement (it does not return any value to the calling process, it simply ends the procedure). Procedures are invoked with keyword `call` and must be followed by the label [Lam21].

Since the PlusCal translator introduces a stack for handling procedures, a module that has a PlusCal algorithm using procedures must extend the standard module `Sequences`.

4.5 Process Declarations

A PlusCal algorithm may declare one or more process templates, each of which can have several instances. Processes define the different actions of the algorithm that may be executed concurrently. The process, like a procedure, can contain its' local variables and call other procedures or macros [Way18].

A process template is presented in Listing 15.

```
process name [= | \in] e
variables ...   \* process-local variables
begin
   (* body expressions *)
end process;
```

Listing 15: Process declaration in PlusCal algorithm

4.6 Labels

In PlusCal, the labels are used to describe concurrent systems accurately. They determine the grain of atomicity of the specification. TLC executes everything inside the label in a single step or an action. Then it checks all invariants and looks for the next label to execute (action to take) [Way18].

The labels should be placed with the following rules [Way18]:

- There should be a label at the beginning of each `process` and before every *while* loop;
- Label cannot be placed inside a `macro` or `with` statement;
- Label must be placed after every `goto`;
- If `either` or `if` statement is used and any possible branch has a label inside it, the label must be placed after the end of the control structure;
- It is impossible to assign the same variable twice in a label.

4.7 Translation to TLA$^+$

The PlusCal translator *SANY*, which comes with *tla2tools* executable, embeds the TLA$^+$ specification corresponding to the PlusCal algorithm in the module within which the algorithm appears, immediately following the algorithm [Lam21].

The delimitation of translation is presented in Listing 16. Users should not touch this area, as it will be overwritten whenever the PlusCal translator is invoked. However, TLA$^+$ definitions may appear either before the PlusCal algorithm or after the generated TLA$^+$ specification. In particular, operators defined before the PlusCal algorithm may be used in the algorithm. Correctness properties are defined below the generated specification because they refer to the variables declared by the translator.

```
\* BEGIN TRANSLATION
...
\* END TRANSLATION
```

Listing 16: Delimitation of TLA$^+$ translation

5 TLA$^+$ Transmutation

TLA$^+$ Transmutation [Maf19] is a transformation tool that works in the opposite direction of our approach – it generates Elixir code from TLA$^+$ specifications. The tool is responsible for parsing internal structures from the `.tla` specification file and then transforming these structures into the Elixir code by using translation rules, which are presented in Section 5.1.

5.1 Translations

The process of transforming a specification into executable code is a translation, which can be expressed with translation rules that map TLA$^+$ specifications (with elements from Figure 12) to Elixir code (combining elements from Figure 13).

The translation rules are layered according to the recursion of the definitions so that the complete specification is translated into the code by the $\vdash$ rules. This means that the tool begins with applying *Top-Level translation* (analyzed in Section 5.1.2) and applies more specific translations recursively.

Identifiers	I, C	Values	v
Parameters	p		
Specification	$Spec$	$::=$	$Module\ M$ CONSTANTS $C_o,\ \ldots, C_n$ VARIABLES $V_o,\ \ldots, V_n$ $D_0,\ \ldots, D_n$
Definition	D	$::=$	$Action(p_0, \ldots, p_n) \triangleq \mathcal{A}$
Action	$\mathcal{A}$	$::=$	$A \mid P \mid \mathcal{A} \wedge \mathcal{A} \mid \mathcal{A} \vee \mathcal{A}$
Condition	P, Q	$::=$	$\neg P \mid P \wedge P \mid P \vee P \mid v_1 \in v_2$
		$\mid$	$v_1 = v_2 \mid v_1 \neq v_2 \mid$ ENABLED $\mathcal{A}$
Transition	T	$::=$	$I' = v \mid$ UNCHANGED $\langle I_0, \ldots, I_n \rangle$
Set	S	$::=$	$v \mid S_a \cup S_b$
Record	R	$::=$	$[k \mapsto v] \mid [I$ EXCEPT $![k] = v]$

Figure 12. TLA$^+$ syntax in transformation rules [Maf19].

```
Atoms  i, k     Values  x, y     Parameters  p

State          t   ::=  action(variables, p̄)
                    |   variables | Map.merge(a, a)
                    |   %{ i_o : x_o, ..., i_n : x_n }
Condition      c   ::=  condition(variables, p̄)
                    |   not c | c and c | c or c
Definition     d   ::=  def action(variables, p̄) do
                          ...
                        end
                    |   def decide([info])
Set            s   ::=  MapSet.new([x])
                    |   MapSet.union(s_a, s_b)
Record         r   ::=  %{ k:  x } | Map.put(i, k, x)
Information  info  ::=  %{ action: "Name", condition: c, state: a }
                    |   Enum.map(x, f (i) -> [info] end)
```

Figure 13. Elixir syntax in transformation rules [Maf19].

5.1.1 Structure

At the beginning of the translation, we can see a signature for $\vdash$ operator

$$\Gamma \vdash_{name} TLA\_Element \mapsto Elixir\_Element,$$

where Γ is the context of TLA$^+$ specification (definitions, parameters, variables, constants, etc.), *name* is a name of the $\vdash$ operator, *TLA_Element* is an element of manually-written TLA$^+$ specification, and the *Elixir_Element* is a generated Elixir code. The arrow $\rightarrowtail$ means *"is translated as"*.

After the translation's signature is defined, the set of translation rules for implementing the translation is presented.

Every translation rule is formed according to the *rule of inference*, where a set of upper statements are premises, and the lower statement is the conclusion. A translation name is also present next to the rule.

5.1.2 Translation Rules

The TLA$^+$ Transmutation tool provides the following groups of translation rules:

- Top-level translation – the utmost level of the translation process, extracting metadata like constants, variables, and definitions;
- Constant translation – defines the translation rules for constant values;
- Definition translation – defines the translation rules for TLA$^+$ definitions;
- Action translation – defines the translation rules for TLA$^+$ actions (e.g. function calls, if statements, etc.);

- Predicate translation – defines the translation rules for TLA$^+$ predicates (e.g. *equals*, *not equals*, *not*, etc.);
- Transition translation – defines the translation rules for variable value change during the state transition (e.g. *primed* operation, *unchanged* operation);
- Value translation – the lowest layer of the translation rules, translating values like *sets*, *records*, *variables*, etc.;
- Initial state predicate translation – defines the translation rules for the initial state of the program;
- Next state action translation – defines the translation for *main* function, which defines the system's state transitions and is the entry point of the program.

For simplicity, we'll analyze only one group – *Action* translations (Figure 14). During the translation, the action is broken down into statements for translation purposes, and each statement is classified as a condition or a transition. Conditions are predicates over the current state, while transition statements also consider the succeeding state of the step.

Every definition yields a pair of two elements – conditions and transitions from a given action. Conditions of action are translated with the operator $\vdash_p$ (Figure 14), resulting in code for a boolean expression, representing whether that action is enabled given a current state. Transitions are translated with the operator $\vdash_t$ (Figure 14), resulting in code that defines the new state based on the current one.

Aggregating transitions by a conjunction ($\wedge$) is delegated to the *Top-level* translations, where the resulting states are merged as if the aggregated transitions were composed. Aggregating transitions by disjunction ($\vee$) is complex, as it can represent non-determinism – possibly more than one transition can be executed. This is solved by delegating this non-determinism to a `decide()` function, described in rule `OR`.

If the definition contains actions based on condition, `IF` (Figure 14) transformation rule is used. It is necessary to evaluate the conditions before applying actions. These actions are split into two sets – conditions and actions. If conditions are not met, the action block is not executed, and if they are met, the action is called and mutates the program's state.

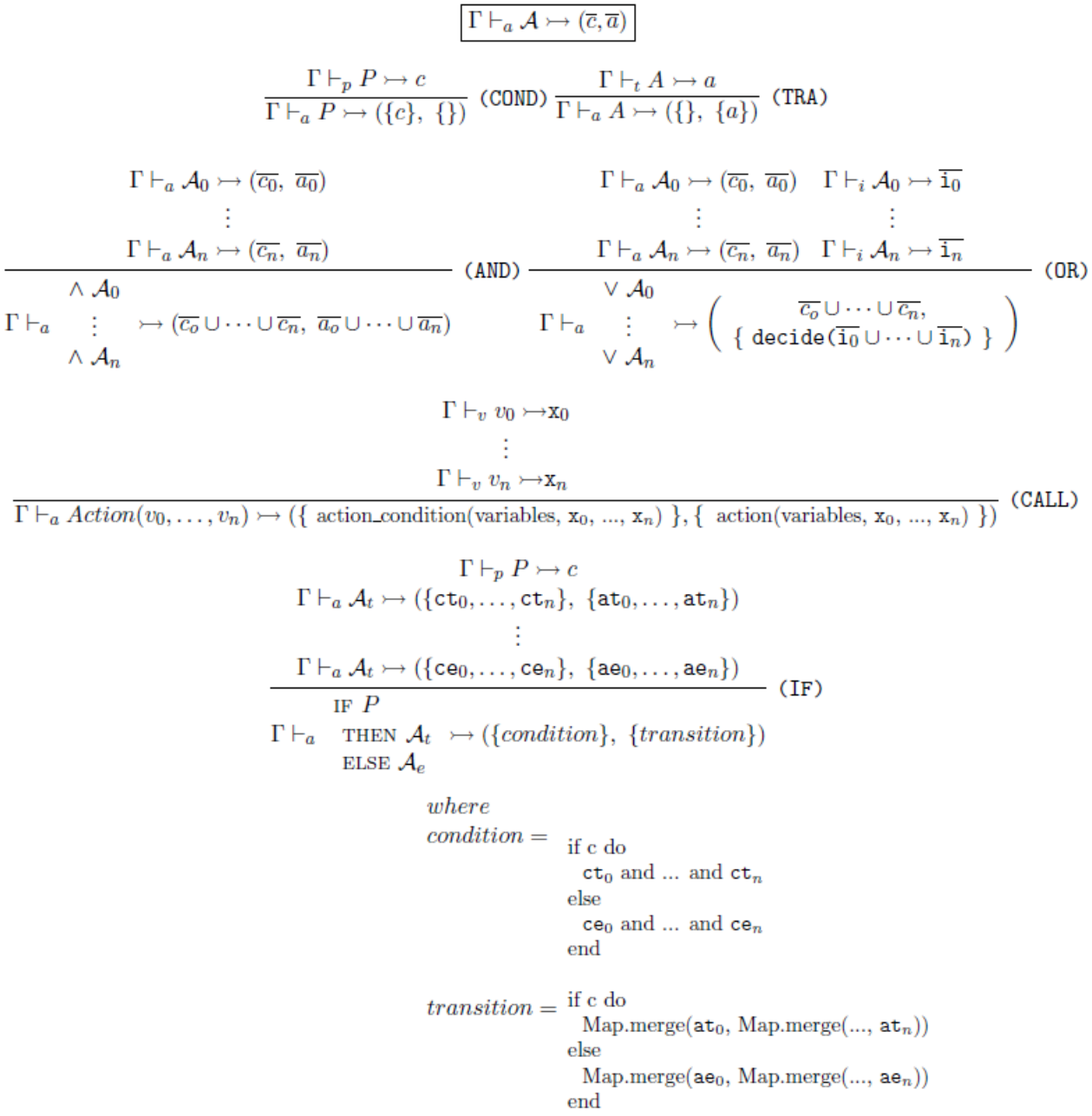

Figure 14. Action translation rules [Maf19].

6 Principal Solution

This section presents the principal solution of the research work. In the beginning, the Elixir to TLA$^+$ translation rules are proposed and the translation method is described. After that, an experiment that uses the translation method is presented. The correctness of generated TLA$^+$ specifications is validated by model checking and refinement. In order to verify that the application of the translation rules does not lose any important information, the Elixir code is generated from the generated TLA$^+$ specification and tested with unit tests. In the end, the advantages, disadvantages, and limitations of the current solution are listed.

6.1 Translation Rules

This section provides the translation rules for Elixir to PlusCal conversion.
PlusCal was selected because of the following reasons:

- Is more similar to the Elixir programming language. Thus, the translation rules are less complex;
- Has additional constructs on top of TLA$^+$, which also makes the translation rules less complex;
- Is used for modeling concurrent algorithms and builds a state machine during the PlusCal to TLA$^+$ conversion process. Thus, there's no need to build the state machine manually;
- PlusCal is not whitespace sensitive. In TLA$^+$, the complex conditionals are whitespace sensitive, and it may not be trivial to generate the same level of indentation with a generator;
- Can handle the recursive calls and generate TLA$^+$ for it;
- Is widely used in TLA$^+$ community [Way18] and is easier to understand.

The provided translation rules are applied recursively, applying more and more specific rules until any of them cannot be applied anymore. In this work, we have created only the rules needed for the translation of the analyzed distributed algorithm, which is presented in section 6.3. The translation rules for non-implemented Elixir expressions should be defined and implemented in the same way as those provided in this section.

The translation rules should be read in this way:

- Above the line, the input for a translation rule is provided – the abstract syntax tree node or a variable of standard Elixir type (e.g., atom, boolean, number, string, map, tuple);

- The pattern matching and additional rules are used to match the correct translation rule (additional conditions for the argument's input are joined with $\wedge$ operator);
- Elements before $\vdash$ defines the pre-condition for the translation rule, based on the context properties;
- The variables in the input are written in *italics* font;
- _ in the input of the translation rule means that any value can be provided;
- Below the line, the output of a translation rule is provided – the PlusCal expression;
- Next to the line, the name of a translation rule is provided.

6.1.1 Additional Context Properties

Some of the translation rules use additional context. The following properties are available as context during the translation:

- *is_single_line* – denotes whether the AST node is the whole line in the source Elixir program. This is useful when applying *type* translation rules to see whether the AST node was used as a return functionality in Elixir (meaning that it is the only thing in the line) or the type was used in another Elixir expression (e.g., inside *if* statement);
- *counter* – defines a number of the particular expression (e.g. *if* statement) in the function. This is useful when translation rules are using labels, that need to be unique in the scope of the whole specification.

6.1.2 Types

This section provides translation rules for the standard types of Elixir. The translation rules that translate Elixir atom (rule ATOM), string (rule STRING), and number (rule NUMBER) type variables leverage standard Elixir functions `is_atom`, `is_binary`, and `is_number` to check the variable type and do the correct value conversion.

$$\frac{\vdash \text{is\_atom}(a)}{\vdash \text{"}a\text{"}} \text{ (ATOM)}$$

$$\frac{\vdash \text{is\_binary}(a)}{\vdash \text{"}a\text{"}} \text{ (STRING)}$$

$$\frac{\vdash \text{is\_number}(a)}{\vdash a} \text{ (NUM)}$$

Translation rules for boolean values (rules BOOL-TRUE and BOOL-FALSE) also use standard Elixir functions `is_boolean`, with an additional check for the argument value to do a proper conversion.

$$\frac{\vdash \text{is\_boolean}(a) \land a == \texttt{true}}{\vdash \texttt{TRUE}} \text{ (BOOL-TRUE)}$$

$$\frac{\vdash \text{is\_boolean}(a) \land a == \texttt{false}}{\vdash \texttt{FALSE}} \text{ (BOOL-FALSE)}$$

For the conversion of Elixir tuple type values, two translation rules exist. The first rule (TUPLE-1) tries to match :{} atom as AST marker element, and the second rule (TUPLE-2) checks whether provided arguments AST is a list of other values. Both translation rules use context property `is_single_line` to check whether the argument is not the only expression in the Elixir code's line (see Section 6.1.1). If this condition is not satisfied, a different translation rule is applied. The Elixir tuple is converted to PlusCal tuple type, which also allows access values by index – the only difference is that indexes start from 1 instead of 0.

$$\frac{!\mathit{is\_single\_line} \vdash \{:\{\}, \_, a\}}{\vdash << a >>} \text{ (TUPLE-1)}$$

$$\frac{!\mathit{is\_single\_line} \vdash \text{is\_list}(a)}{\vdash << a >>} \text{ (TUPLE-2)}$$

The Elixir variable is translated using VAR rule. It checks whether the value of the children element of the AST node is equal to `nil`.

$$\frac{\vdash \{a, \_, \text{nil}\}}{\vdash a} \text{ (VAR)}$$

The Elixir `nil` is translated by rule NIL. It translates to NULL, which is exposed as a module constant.

$$\frac{\vdash \text{nil}}{\vdash \text{NULL}} \text{ (NIL)}$$

6.1.3 Return Value

This section defines translation rules for return values. PlusCal's processes and procedures do not have a concept of a value return. Still, the value could be assigned to a variable and used immediately where needed (e.g., in other procedures). This is represented in value return translation rules below. A new variable `result` is defined, and then the value is generated from the AST. To distinguish the actual return of a value, `is_single_line` context property is used to check whether the value is the only thing in Elixir's source code line (see Section 6.1.1).

Here are defined value return translation rules for variables of tuple type, but translation rules for values of another type should be described in the same way.

$$\frac{\mathit{is\_single\_line} \vdash \{:\{\}, \_, a\}}{\vdash \mathit{result} :=<< a >>} \text{ (RET-TUPLE-1)}$$

$$\frac{\mathit{is\_single\_line} \vdash \text{is\_list}(a)}{\vdash \mathit{result} :=<< a >>} \text{ (RET-TUPLE-2)}$$

6.1.4 Empty Value Assignment

This section provides translation rules for empty value assignments. If the empty value assignment expression is encountered during the translation, the variable is assigned to itself. This is done due to the reason that PlusCal does not allow empty value assignments for some types (e.g., structure). Thus, the default value should be assigned when defining a local variable in PlusCal.

For the experiment, only the empty value for *map* type variable was needed (rule EMPTY-MAP), but empty value translation rules for other types would look and behave the same way.

$$\frac{\vdash \{:=, \_, [\{a, \_, \text{nil}\}, \{:\%\{\}, \_, []\}]\}}{\vdash a := a;} \text{ (EMPTY-MAP)}$$

6.1.5 Mathematical Operators

This section provides translation rules for standard mathematical operators. Translation rules for mathematical operators look very similar – the only different thing is a marker of the Elixir AST node. The more specific translation rules are applied for *left* and *right* variables until the whole PlusCal expression is built. Also, the braces are added around the translated expression in order to avoid problems with the order of operators if several math operators are used (e.g., `((a + b) * c)`).

Translation rules for the following mathematical operators were defined:

- *Equality check* operator (rule MATH-EQ);
- *Less than* operator (rule MATH-LT);
- *Less than or equal* operator (rule MATH-LTEQ);
- *Greater than* operator (rule MATH-GT);
- *Greater than or equal* operator (rule MATH-GTEQ);
- *Plus* operator (rule MATH-PLUS);
- *Minus* operator (rule MATH-MINUS);
- *Multiplication* operator (rule MATH-MULT);
- *Division* operator (rule MATH-DIV);
- *And* operator (rule MATH-AND);
- *Or* operator (rule MATH-OR);

$$\frac{\vdash \{:==, \_, [left, right]\}}{\vdash (left = right)} \text{ (MATH-EQ)}$$

$$\frac{\vdash \{:<, \_, [left, right]\}}{\vdash (left < right)} \text{ (MATH-LT)}$$

$$\frac{\vdash \{:<=, \_, [left, right]\}}{\vdash (left <= right)} \text{ (MATH-LTEQ)}$$

$$\frac{\vdash \{:>, \_, [left, right]\}}{\vdash (left > right)} \text{ (MATH-GT)}$$

$$\frac{\{:>=, \_, [left, right]\}}{\vdash (left >= right)} \text{ (MATH-GTEQ)}$$

$$\frac{\vdash \{:+, \_, [left, right]\}}{\vdash (left + right)} \text{ (MATH-PLUS)}$$

$$\frac{\vdash \{:-, \_, [left, right]\}}{\vdash (left - right)} \text{ (MATH-MINUS)}$$

$$\frac{\vdash \{:*, \_, [left, right]\}}{\vdash (left * right)} \text{ (MATH-MULT)}$$

$$\frac{\vdash \{:/, \_, [left, right]\}}{\vdash (left \texttt{\textbackslash div} right)} \text{ (MATH-DIV)}$$

$$\frac{\vdash \{:and, \_, [left, right]\}}{\vdash (left \texttt{/\textbackslash} right)} \text{ (MATH-AND)}$$

$$\frac{\vdash \{:or, \_, [left, right]\}}{\vdash (left \texttt{\textbackslash/} right)} \text{ (MATH-OR)}$$

Not operator is quite different from the mathematical operators described above. It takes only one argument's abstract syntax tree, which is translated using more specific rules and then negated afterward.

$$\frac{\vdash \{:\text{not}, \_, [a]\}}{\vdash \sim \text{a}} \text{ (NOT)}$$

6.1.6 Conditionals

This section presents the translation rules for conditional statements of Elixir. One of the most widely used conditional statements in Elixir is *if* statement. PlusCal has `if/else` construct, which is similar to Elixir's, and that simplifies the translation rule. Like in the Elixir, PlusCal's statement has *do* and *else* blocks, which contain other expressions. In addition, PlusCal's *if* statement should have a label at the beginning of the inner expressions of the block containing a label. For simplicity, we add a label every time.

The translation rules for *if* statements with and without *else* block can be seen in translation rules IF and IF-ELSE. They are identical, except that IF-ELSE rule has an additional ELSE block.

$$\frac{\vdash \begin{array}{l}\{:\text{if}, \_, [condition, [\\ \quad \text{do: } \{:\_\_\text{block}\_\_\_, [], do\_block\}]]\}\end{array}}{\begin{array}{l}\text{if } condition \text{ then}\\ \quad counter \vdash \begin{array}{l}\text{if\_}counter\text{:}\\ \quad do\_block\end{array}\\ \text{end if;}\end{array}} \text{(IF)}$$

$$\frac{\begin{array}{l}\{:\text{if}, \_, [condition, [\\ \vdash \text{do: } \{:\_\_\text{block}\_\_\_, [], do\_block\},\\ \quad \text{else: } \{:\_\_\text{block}\_\_\_, [], else\_block\}]]\}\end{array}}{\begin{array}{l}\text{if } condition \text{ then}\\ \quad \text{if\_}counter\text{:}\\ \quad\quad do\_block\\ counter \vdash \text{else}\\ \quad\quad \text{else\_}counter\text{:}\\ \quad\quad\quad else\_block\\ \quad \text{end if;}\end{array}} \text{(IF-ELSE)}$$

Another commonly used conditional statement in Elixir is *cond*. It executes the first expression, for which the condition is *true*. The construct with the same behavior exists in PlusCal – it allows executing an action based on the fulfilled condition and then assigns the result to the variable. The translation rule is defined in COND. The transformation rule would be more complicated if PlusCal did not have such a construct.

$$\frac{\vdash \begin{array}{l}\{:=, \_, [\{variable, \_, \text{nil}\},\\ \quad \{:\text{cond}, \_, [[\text{do: } [condition_1, \ldots, condition_n]]]\}\}\end{array}}{\begin{array}{l}variable \coloneqq \text{CASE}\\ \quad \vdash \begin{array}{l}condition_1 \rightarrow action_1\\ \quad \vdots\\ \quad [] \ condition_n \rightarrow action_n;\end{array}\end{array}} \text{(COND)}$$

Pin operator in Elixir is used when pattern matching to the value of the variable is needed. If the value is not matched, an exception is thrown, and the execution stops. The same behavior can be achieved in PlusCal with the labels functionality. If the values of the left and right operands are not the same, the process execution jumps to the automatically defined *Done* label and finishes the process this way. Also, another label must be placed after every *goto* call.

The PIN operator that matches a value of a variable with the value of the right operand can be found in the rule PIN-VAR. For the experiment, only this rule was needed, but values of other type variables could be matched in the same way.

$$\frac{\vdash \{:=, \_, [\{:\wedge, \_, [variable, , nil, right\_operand]\}}{\begin{array}{l}\text{if } variable \neq right\_operand \text{ then} \\ \quad \text{goto } Done; \\ \text{end if;} \\ \text{after\_pin\_}counter:\end{array}} \text{ (PIN-VAR)}$$
$$counter \vdash$$

The *Arguments Condition* is a condition defined by pattern matching the specific values of the Elixir function's input (e.g., `def handle_message(rbc, :PROPOSE, from, value = _msg)`). This means that the clause of a function must be executed only when the first value of tuple `msg` is equal to `:PROPOSE`. The described behavior is the same as for the *pin* operator. Thus, the exact translation rule applies.

6.1.7 Enumerable

This section provides translation rules for Elixir's enumerable. The `Enum` module provides a huge range of functions to transform, sort, group, filter, and retrieve items from enumerables. It is one of the modules developers use frequently in their Elixir code.

Currently, only one translation rule is implemented – ENUM-INTO. It inserts the given enumerable into the collectable (e.g., `msgs = Enum.into(peers, %{}, fn peer_id -> {peer_id, [{:ECHO, me, value}]} end)`). What it does is for every available value of the iterator from the enumerable, the assignment is executed. The PlusCal structure type can achieve such behavior, where every *key* has an assigned *value*. The variable must have a default value set; otherwise, model checking would fail due to the invalid assignment operation.

Translation rules for other enumerable operations should be defined similarly.

$$\frac{\begin{array}{l}\{:=, \_, [\\ \quad \{variable, \_, \text{nil}\},\\ \quad \{\{:., \_, [\{:\_\_\_aliases\_\_\_, \_, [:Enum]\}, :into]\}, \_,\\ \quad\ [\\ \vdash \quad\ \{enumerable, \_, \text{nil}\},\\ \quad\ \{:\%\{\}, \_, []\},\\ \quad\ \{:fn, \_, [\{:->, \_, [[iterator], assignment]\}]\}\\ \quad\ ]\}\\ \quad]\}\end{array}}{\vdash variable := [iterator\ \texttt{\textbackslash in}\ enumerable\ |->\ assignment]}\ \text{(ENUM-INTO)}$$

6.1.8 Map

This section provides translation rules for Elixir's maps. Maps are the key-value data structure in Elixir.

The rule MAP-DESTR defines the translation for Map destructuring into separate variables (e.g., `%RBC{me:me, n:n, f:f} = rbc`). The assignments are expressed as key-value pairs in the AST. Thus, we iterate through every available key, create a variable of the key name and then assign the value of the source variable from the same key.

$$\frac{\begin{array}{l}\{:=, \_, [\{:\%, \_, [\{:\_\_\_aliases\_\_\_, \_, \_\},\\ \vdash\ \{:\%\{\}, \_, [assgn\_map]\}]\},\\ \quad \{src\_variable, \_, nil\}]\}\end{array}}{\begin{array}{l}assgn\_map[0].key := src\_variable[assgn\_map[0].key].value;\\ \vdash \vdots\\ assgn\_map[n].key := src\_variable[assgn\_map[n].key].value;\end{array}}\ \text{(MAP-DESTR)}$$

The rule MAP-GET defines the translation for the operation that returns the value from the map based on the provided key (e.g., `value = Map.get(map_var, key)`). As the structure type has a similar behavior to a map, a simple value retrieval with square parentheses is used.

$$\frac{\begin{array}{l}\{:=, \_, [\\ \vdash \quad \{variable, \_, \text{nil}\},\\ \quad \{\{:., \_, [\{:\_\_\_aliases\_\_\_, \_, [:Map]\}, :get]\}, \_,\\ \quad\ [\{map\_variable, \_, \text{nil}\}, key, \_]\}]\}\end{array}}{\vdash variable := map\_variable[key];}\ \text{(MAP-GET)}$$

The translation of value updates of a map (e.g., `rbc = %RBC{rbc | echo_sent:true, msg_recv:msg_recv}`) is described by the rule MAP-UPD. Such

expression in Elixir returns a new variable. The same behavior should be achieved in the generated specification. This is done by assigning the current map value to a new variable and then mutating it. After this assignment, the label should be put so that simultaneous assignment does not happen as it might cause race conditions. These steps could be skipped if the new value is assigned to the old variable. Then, a value is assigned for every key from the updates map. If there is more than one update, they should be joined by PlusCal operator `||`, which executes all updates simultaneously.

$$\frac{\begin{array}{l}\{:=, \_, [\\ \quad \{variable, \_, \text{nil}\},\\ \quad \{:\%, \_, [\\ \vdash \quad\quad \{:\_\_\text{aliases}\_\_\_, \_, \_\},\\ \quad\quad \{:\%\{\}, \_, [\{:|, \_, [\{src\_variable, \_, \text{nil}\}, updates\_map]\}]\}\\ \quad]\}\\ ]\}\end{array}}{\begin{array}{l}variable := src\_variable;\\ map\_update:\\ \vdash variable[updates\_map[0].key] := updates\_map[0].value\\ \quad \vdots\\ \quad || \; variable[updates\_map[n].key] := updates\_map[n].value;\end{array}} \text{(MAP-UPD)}$$

Putting the element to the map (e.g., `msg_recv = Map.put(msg_recv, {from, :PROPOSE}, true)`) is translated by MAP-PUT rule. It works in the same way as MAP-UPD rule, except that only one key in the variable is updated with a new value.

$$\frac{\begin{array}{l}\{:=, \_, [\\ \vdash \quad \{variable, \_, \text{nil}\},\\ \quad \{\{:., \_, [\{:\_\_\text{aliases}\_\_\_, \_, [:\text{Map}]\}, :\text{put}]\}, \_,\\ \quad\quad [\{src\_variable, \_, \text{nil}\}, key, value]\}]\}\end{array}}{\begin{array}{l}variable := src\_variable;\\ counter \vdash map\_put\_counter:\\ \quad\quad variable[key] := value;\end{array}} \text{(MAP-PUT)}$$

In order to calculate the size of a map (e.g., `size = map_size(value_recv)`), the MAP-SIZE translation rule is used. It generates an operator definition `map_size` in PlusCal and then uses it to get the count of keys in the structure. As the structure may be initialized with a default value of `NULL`, we filter out such keys.

$$\frac{\vdash \{:=, \_, [\{variable, \_, \text{nil}\}, \{:\text{map\_size}, \_, [\{argument, \_, \text{nil}\}]\}]\}}{\begin{array}{l}\text{define} \\ \quad map\_size(structure) == \\ \quad\quad \text{Cardinality}(\{k \texttt{ \textbackslash in } (\text{DOMAIN } structure) : structure[k] \neq \text{NULL}\}) \\ \vdash \text{end define;} \\ \vdots \\ variable \coloneqq \text{map\_size}(argument);\end{array}} \text{(MAP-SIZE)}$$

6.1.9 Function Calls

This section defines translation rules for function calls.

The anonymous function call (e.g., `predicate.(value)`) is translated with FN-ANON-NORES rule. In this translation rule, the anonymous function should be implemented in the specification's operator definitions section and used like any other operator. This translation rule does not assign the result to any value. Thus, the returned value should be used immediately.

Translation rules for other function calls were not defined yet. Still, their structure would be similar, where the function would be implemented as a PlusCal operator definition, procedure, or macro.

$$\frac{\vdash \{\{:., \_, [\{function\_name, \_, \text{nil}\}]\}, \_, args\}}{\begin{array}{l}\text{define} \\ \quad function\_name\_fn(args) == \ldots \\ \vdash \text{end define;} \\ \vdots \\ function\_name\_fn(args)\end{array}} \text{(FN-ANON-NORES)}$$

6.2 Translation Method

The Elixir to TLA$^+$ translation method, which can be seen in Figure 15 has 5 main steps, which will be analyzed in depth in the corresponding sections:

1. Creating a configuration for Elixir module specification generation.
2. Creating a model checking configuration.
3. Translating Elixir code to PlusCal specification.
4. Generating TLA$^+$ specification from PlusCal specification.
5. Running model checking for generated specifications.

In the research, we aim to extract the specifications only from the sequential part of the Elixir source code. Thus, steps 3 to 5 are repeated for every function defined in step 1.

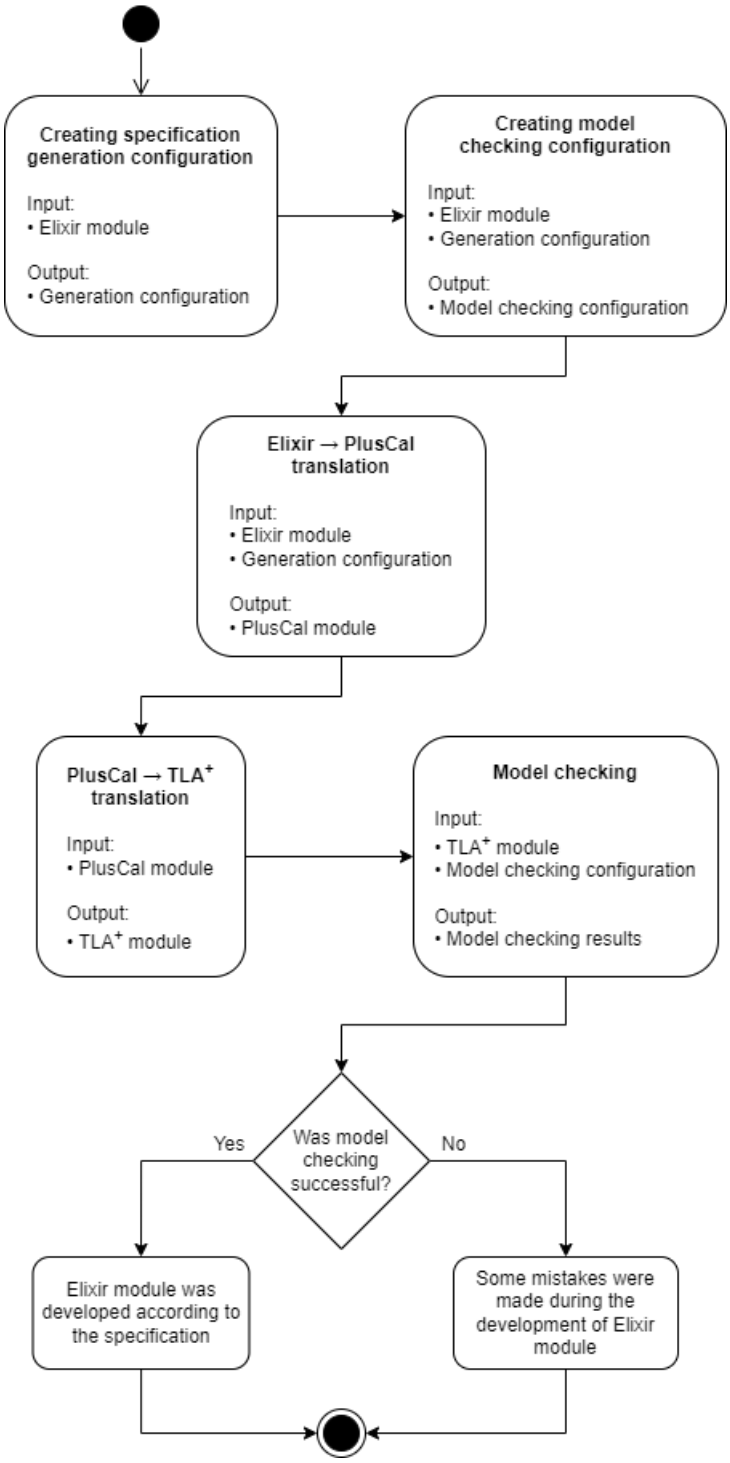

Figure 15. Elixir to TLA$^+$ translation method

6.2.1 Creating a Specification Generation Configuration

Firstly, the configuration for the translated Elixir module should be defined. The configuration holds various essential data, which is needed for specification, but couldn't be extracted from the code in order to have fully functional and correctly working specification.

The model, which can be found in Figure 16 was defined to save and manipulate configuration in a structured way.

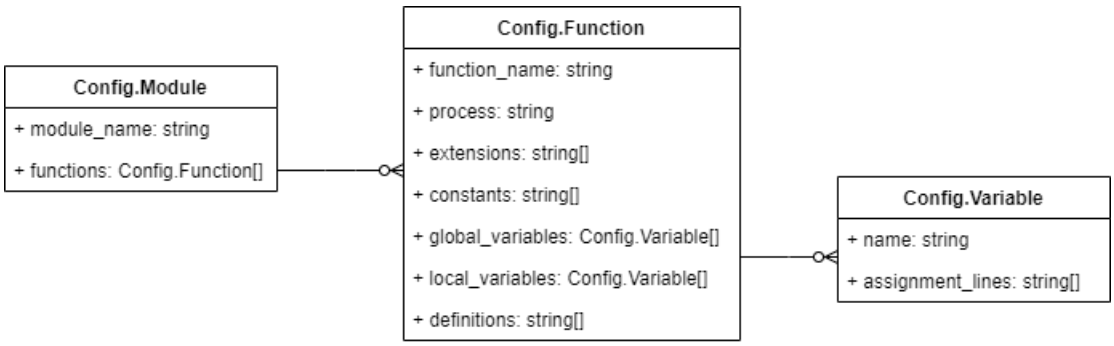

Figure 16. Specification generation configuration model

The module configuration is used for defining for what functions the separate specifications will be extracted.

The function configuration is used to define the specific data needed for the generated specification. The following data is defined:

- `function_name` – is used for defining the specific name of the function (e.g., *handle_echo_message*), as in Elixir, the function may have multiple clauses, which represent different predicates in TLA$^+$ specification. Also, the value of this parameter is used to map the configuration with the translated Elixir function or function clause – the function must be annotated with `@tlagen_function` and function name (e.g., `@tlagen_function :handle_echo_message`);
- `process` – is used for defining a set of PlusCal processes (e.g., `\in` *AllNodes*);
- `extensions` – is used for defining what standard TLA$^+$ modules specification should extend (e.g. *Naturals*). These could be generated from the code, but for the proof of concept, it was chosen to define them in the configuration;
- `constants` – is used to define the specification's constants. It was chosen to store the constants in the configuration, as Elixir does not have a context about which values should be modeled during the model checking. Thus, they should be defined manually;
- `global_variables` – is used to define and initialize specification scope variables. This is needed because it is impossible to run model checking of the specification with some variables not having a value or having a value of incorrect type. Also, the Elixir does not have a context of what default values the variables should have for the model checking;
- `local_variables` – is used to define and initialize process scope variables that need to have a default value (for the same reason which was described in the previous point);
- `definitions` – is used to define the specification scope definitions, which may be used for variable initialization or inside the process.

Some values, like constants or variables, could be defined as annotations in the Elixir module and reused for all of the module's specifications. Still, for the proof of concept, they are defined inside the configuration file.

The configuration file should be named after the translated Elixir file, adding the `.tlagen.json` extension. The configuration file should be saved in the same directory as the Elixir file.

6.2.2 Creating a Model Checking Configuration

Secondly, the configuration for TLA$^+$ model checking is created. It contains essential data used during the model checking, like constants, properties, invariants, etc. This part is not extracted from the code as a human being should decide how the generated specification should be verified.

```
SPECIFICATION Spec
CONSTANTS
    CN = {n_1, n_2, n_3}
    FN = {n_4}
    Value = {v_1, v_2}
    NotValue = NotValue
    NULL = NULL
CHECK_DEADLOCK FALSE
PROPERTIES
    AbsStepSpec
    Liveness
INVARIANTS
    TypeOK
```

Listing 17: Model checking configuration.

6.2.3 Translating Elixir Code to PlusCal Specification

After the configuration is defined, the process continues to the actual translation.

Firstly, the Abstract Syntax Tree of the Elixir module is extracted, and the AST expressions of the function we're interested in are collected based on the `@tlagen_function` and `function_name` from the configuration.

Then, the translation of the Elixir function to the PlusCal process is executed based on the rules described in Section 6.1.

Once we have a generated PlusCal process, we create a `.tla` file based on the configuration and generated process. You can find the structure of the generated file with the PlusCal algorithm in Listing 18. Extensions, constants, global variables, and a process set are generated only based on the configuration. The process is generated purely from the Elixir function AST. The Definitions and local process variables are generated based on the configuration and generated process.

Variables that do not have a default value specified are assigned a value of `NULL`, which is a module's constant.

```
---------------------- MODULE function_name ----------------------

EXTENDS (* extensions from the configuration *)

CONSTANTS (* constants from the configuration *)

(*--algorithm function_name
    variables
        (* global variables from the configuration *)

    define
        (* definitions from the configuration *)
        (* generated definitions *)
    end define;

    fair process function_name (* process from the configuration *)
    variables
        (* local variables from the configuration *)
        (* generated local variables *)
    begin
        (* generated process *)
    end process;
end algorithm;
=============================================================================
```

Listing 18: Generated PlusCal module.

6.2.4 Generating TLA$^+$ Specification from PlusCal Specification

Once the PlusCal specification is acquired, it is translated to TLA$^+$ using the *SANY* parser (see Section 4.7). After this generation, the PlusCal specification remains commented out, and generated TLA$^+$ can be used like any other TLA$^+$ specification. The *SANY* parser is invoked via the command line – `tla2tools` with `pcal.trans` command is used to execute the conversion between the PlusCal and TLA$^+$.

6.2.5 Model Checking for Generated Specifications

Once the TLA$^+$ specification is acquired, the model checking and refinements can be executed for the generated specification. Based on the model checking results, it can be verified whether the code does what it needs to do. The TLC model checker is invoked via the command line – `tla2tools` with `tlc2.TLC` command is used to execute the model checking.

Figure 17 provides an illustration of the mapping between states in a PlusCal specification and states in a TLA$^+$ specification. Instead of defining a predicate for the initial

states in the generated specification, we use the states for which the refinement mapping implies the invariant on the abstract specification.

The process of model checking, which involves exhaustively exploring the state space of a system to verify properties, is also affected by this mapping. The intermediate steps modeled in the generated TLA$^+$ specification increase the number of states. As these steps are not present in the abstract specification, the model checking is slower for the generated specifications.

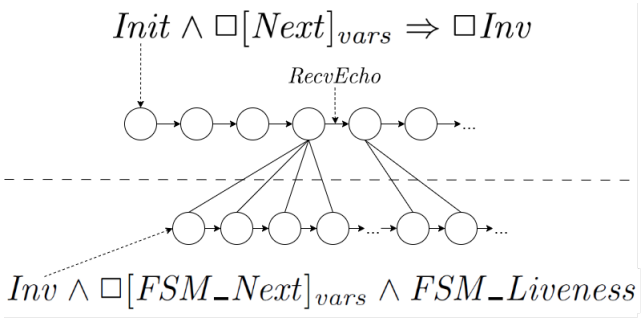

Figure 17. Mapping states of abstract TLA$^+$ and PlusCal specifications.

6.2.6 Including Translation Method Into the Build Pipeline

The described process can be included in the build pipeline. The process could look as presented in Algorithm 2. It uses the Elixir to TLA$^+$ translation library and other standard tools, provided by Elixir and TLA$^+$ toolbox.

Algorithm 2 Translation method's inclusion into the build pipeline.

1: *// executed once by software engineer*
2: **add** reference to the library, which translates Elixir to TLA$^+$.
3: **define** configuration for TLA$^+$ generation.
4: *// executed automatically during build process*
5: **translate** Elixir to PlusCal.
6: **translate** PlusCal to TLA$^+$ using `tla2tools`.
7: **execute** model checking using `tla2tools`.
8: **fail** the build if model checking was not successful.

6.3 Experiment

To show how translation rules and the translation method work, an experiment was carried out. In the experiment, the specifications for Bracha reliable broadcast (RBC) protocol's (see section 3.1) functions were generated, and refinement mapping to the original TLA$^+$ specification was defined, which later was model checked. Even though specifications were generated and the model checked for all algorithm's functions, for simplicity, we will analyze only one of them.

The source code of the experiment can be found in the author's GitHub repository [4].

[4] `https://github.com/DeividasBrazenas/tla-generator`

We have used the TLA$^+$ specification of the protocol, which was analyzed in the section 3. Even though all steps of the algorithm were translated in the experiment, we will analyze only one of them – *Receive Echo*. Its TLA$^+$ specification can be seen in Figure 18.

```
> 9: upon receiving 2t + 1 ⟨ECHO, M⟩ messages
>      and not having sent a READY message do
> 10:    send ⟨READY, M⟩ to all
```

$$
\begin{aligned}
&RecvEcho(eq) \triangleq \\
&\quad \exists\, n \in CN,\, v \in Value : \\
&\qquad \land\, eq \in QXF \\
&\qquad \land\, \forall\, qn \in eq : HaveEchoMsg(qn,\, \{v\}) \\
&\qquad \land\, \neg HaveReadyMsg(n,\, Value) \\
&\qquad \land\, msgs' = msgs \cup \{[t \mapsto \text{“READY”},\, src \mapsto n,\, v \mapsto v]\} \\
&\qquad \land\, \text{UNCHANGED}\ \langle bcNode,\, bcValue,\, predicate,\, output\rangle
\end{aligned}
$$

Figure 18. *Receive Echo* step in TLA$^+$.

6.3.1 Protocol's Implementation in Elixir

The protocol's implementation in Elixir can be found in the experiment's repository [5]. It was ported to Elixir based on the Go implementation from Wasp repository [6]. The protocol's implementation is based on the *Wasper* library [7]

In Figure 19 you can find the implementation of *Receive Echo* step. It uses common Elixir language expressions to implement the behavior of steps 9 and 10. The annotation `@tlagen_function :handle_echo_message` denotes that the TLA$^+$ specification will be generated for the function.

The distinct expressions of the Elixir code can be found marked on the right side of Figure 19. Later they will be used to show the mapping between the Elixir source code and the generated PlusCal specification.

[5] https://github.com/DeividasBrazenas/tla-generator/blob/main/test/apps/bracha/lib/rbc_bracha.ex

[6] https://github.com/iotaledger/wasp/blob/develop/packages/gpa/rbc/bracha/bracha.go

[7] https://github.com/DeividasBrazenas/tla-generator/blob/main/test/apps/bracha/lib/wasper_gen_pure_alg.ex

```elixir
# 09: upon receiving 2t + 1 (ECHO, M) messages and not having sent a READY message do
# 10:     send (READY, M) to all
@tlagen_function :handle_echo_message
def handle_message(rbc, {:ECHO, from, value} = _msg) do                                    } 1
  %RBC{me: me, n: n, f: f, peers: peers, msg_recv: msg_recv,                               }
       echo_recv: echo_recv, ready_sent: ready_sent, output: output} = rbc                 } 2

  existing_recv = Map.get(echo_recv, value, %{})                                           } 3
  value_recv = Map.put(existing_recv, from, true)                                          } 4
  echo_recv = Map.put(echo_recv, value, value_recv)                                        } 5

  rbc = %RBC{rbc | echo_recv: echo_recv}                                                   } 6

  count = map_size(value_recv)                                                             } 7

  if not ready_sent and count > (n + f) / 2 do                                             } 8
    msg_recv = Map.put(msg_recv, {from, :ECHO}, true)                                      } 9
    rbc = %RBC{rbc | ready_sent: true, msg_recv: msg_recv}                                 } 10

    msgs = Enum.into(peers, %{}, fn peer_id -> {peer_id, [{:READY, me, value}]} end)       } 11
    {:ok, rbc, msgs, output}                                                               } 12
  else
    msgs = %{}                                                                             } 13
    {:ok, rbc, msgs, output}                                                               } 14
  end
end
```

Figure 19. *Receive Echo* step in Elixir

6.3.2 Generated Specifications

The generated specification can be found next to the Elixir source code, in the directory, based on the module and generated function name [8].

Figure 20 presents a part of generated PlusCal specification for the Elixir function shown in Figure 19. For simplicity, only the main part – the process – is provided, but the whole generated specification can be found in the repository. The corresponding expressions in Elixir and PlusCal are labeled with the same numbers.

[8] https://github.com/DeividasBrazenas/tla-generator/blob/main/test/apps/bracha/lib/rbc_bracha_specs

```
handle_echo_message:
  if (_msg[1] /= "ECHO") then                                                    } 1
    goto Done;
  end if;
  after_condition:

  echo_recv := rbc.echo_recv;                                                    } 2
  f := rbc.f;
  me := rbc.me;
  msg_recv := rbc.msg_recv;
  n := rbc.n;
  output := rbc.output;
  peers := rbc.peers;
  ready_sent := rbc.ready_sent;

  existing_recv := echo_recv[value];                                             } 3

  value_recv := existing_recv;                                                   } 4
  map_put_0:
  value_recv[from] := TRUE;

  echo_recv[value] := value_recv;                                                } 5

  rbc.echo_recv := echo_recv;                                                    } 6

  count := map_size(value_recv);                                                 } 7

  if (~ready_sent /\ (count > ((n + f) \div 2))) then                            } 8
    if_0:
      msg_recv[from]["ECHO"] := TRUE;                                            } 9

      rbc.msg_recv := msg_recv                                                   } 10
      || rbc.ready_sent := TRUE;

      msgs := [peer_id \in peers |-> msgs[peer_id] \cup {<<"READY", me, value>>}]; } 11

      result := <<"ok", rbc, msgs, output>>;                                     } 12

  else
    else_0:
      msgs := msgs;                                                              } 13

      result := <<"ok", rbc, msgs, output>>;                                     } 14

  end if;
```

Figure 20. *Receive Echo* step in PlusCal

6.4 Correctness and Completeness

This section analyzes the correctness and completeness of the solution.

6.4.1 Correctness

The correctness of generated specifications was assessed by translation validation (see 1.4) and specification refinement (see 3.7). As input, the specifications that were generated

during the experiment are used for verification.

6.4.1.1 Assessment by Specification Refinement

A refinement for *Receive Echo* step is presented in Figure 21. As *BrachaRBC* specification is more abstract than generated *handle_echo_message*, the refinement mapping was defined. As it was not implemented in the code, the *predicate* value for all correct nodes is set to `TRUE`. The *output* value is set to *NotValue* for all correct nodes like in the abstract specification, as this step of a specification does not do anything with this variable. The *msgs* variable is populated with `READY` messages from all correct nodes once they have been sent to them.

The specification of *Receive Echo* step functionality from the original specification is defined with *AbsStepSpec* property is checked during the model checking.

Another temporal property that is checked during the model checking is *Liveness*, which checks whether the specification does what is expected. In this case, one of the following conditions must be true:

- If there are enough `ECHO` messages received, the *READY* message is sent once;
- If there are not enough `ECHO` messages received, the *READY* message is not sent.

The check whether the node has sent a `READY` message is implemented in *IsReadySent(res, node)* predicate. It uses the `result` variable from the generated spec (see expressions 12 and 14 in Figure 20) to check whether the required values have been set. In this case, the value for the *ready_sent* property should be `TRUE`. The result should contain a `READY` message sent for every peer, or the message was sent earlier, and there are no messages in the result (see expression 8 in Figure 20).

Also, the *TypeOK* invariant is checked during the model checking, ensuring that all variables have the correct type of values.

```
MODULE handle_echo_message_REF
EXTENDS handle_echo_message

1. Result should exist for the node.
2. ready_sent should be set to TRUE for the node.
3.1. If ready message was sent earlier, then there should be no messages in result.
3.2. If ready message was not sent earlier, then ready message should exist for every peer in the result.
IsReadySent(res, node) ≜
    ∧ res[node] ≠ NULL
    ∧ res[node][2].ready_sent = TRUE
    ∧ ∨ ∧ readySent
        ∧ res[node][3] = [node_id ∈ AN ↦ {}]
      ∨ ∧ ¬readySent
        ∧ ∀ peer ∈ AN : {⟨"READY", node, bcValue⟩} ⊆ res[node][3][peer]

EnoughEchoMsgs(cn) ≜ EchosCount(rbcs[cn].echo_recv) ≥ ((N + F) ÷ 2)

org_spec ≜ INSTANCE BrachaRBC WITH
    bcNode ← bcNode,
    bcValue ← bcValue,
    predicate ← [p ∈ CN ↦ TRUE],
    output ← [o ∈ CN ↦ NotValue],
    msgs ← IF ∀ cn ∈ CN : IsReadySent(result, cn)
           THEN [t : {"READY"}, src : CN, v : {bcValue}]
           ELSE {}

QXF ≜ {q ∈ SUBSET AN : Cardinality(q) = ((N + F) ÷ 2) + 1}   Intersection is F + 1.

AbsStepNext ≜ ∃ eq ∈ QXF : org_spec!RecvEcho(eq)
AbsStepSpec ≜ □[AbsStepNext]⟨⟩

For every correct node:
1.1. If there are enough echo messages received, the ready message should be sent.
1.2. If there are not enough echo messages received, the ready message should not be sent.
Liveness ≜
    ◇(∀ cn ∈ CN :
      ∨ ∧ EnoughEchoMsgs(cn)
        ∧ IsReadySent(result, cn)
      ∨ ∧ ¬EnoughEchoMsgs(cn)
        ∧ ¬IsReadySent(result, cn))

TypeOK ≜ org_spec!TypeOK

THEOREM Spec ⇒
    ∧ □TypeOK
    ∧ AbsStepSpec
    ∧ Liveness
PROOF OMITTED   Checked by the TLC.
```

Figure 21. *Receive Echo* step refinement in TLA$^+$.

6.4.1.2 Assessment by Translation Validation

In order to evaluate the correctness of the defined translation rules (see Section 6.1) and ensure the preservation of critical information, we have developed an analyzer that takes

the PlusCal algorithm as input and generates the Elixir code[9]. The diagram of our used translation validation method can be seen in Figure 22.

We have chosen the generated PlusCal algorithm as an input as it allows us to test whether our defined translation rules (see Section 6.1) accurately capture all the essential details and properties of the original algorithm. We did not choose TLA$^+$ module as an input because the compilation of the PlusCal algorithm to TLA$^+$ is done by the standard tools. Given the extensive usage of these tools and being present for more than a decade, we consider them reliable and correct.

The analyzer incorporates the utilization of regular expressions to establish a mapping between PlusCal constructs and Elixir code. It is important to note that this tool was purpose-built specifically for this experiment, and thus, no generic translation rules were defined for this particular translation process.

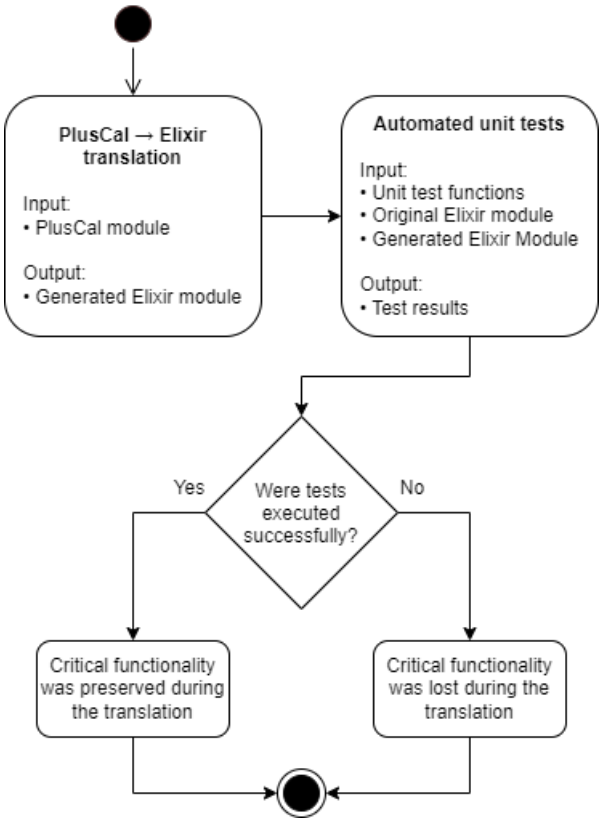

Figure 22. Method of translation validation

To ensure the preservation of code behavior, we have used automated unit tests on the initial code base. These tests involved passing references of both the original and generated Elixir functions to the test method, thereby verifying that their respective outputs were identical. The example for one of the tests is presented in Listing 19. This unit test asserts the functional requirement of the Bracha algorithm, whether the node sends `READY` message once enough `ECHO` messages are received from other nodes. The test function invokes a method passed by the reference and then asserts whether all of the required details were set in the result.

[9] `https://github.com/DeividasBrazenas/tla-generator/tree/main/lib/verifier`

With the algorithm attaining a comprehensive unit test coverage of 100% and the successful execution of all tests, we can claim that the translation process preserved the critical behavior of the algorithm without any loss.

```
test "ready message is sent (original)",
  do: test_ready_message_is_sent(&Wasper.HBBFT.RBC.Bracha.handle_message/2)

test "ready message is sent (generated)",
  do: test_ready_message_is_sent(&Generated.HandleEchoMessage.handle_message/2)

defp test_ready_message_is_sent(handle_message) do
  # Arrange
  bc_value = "value"
  msg = {:ECHO, :n_2, bc_value}
  me = :n_1
  echo_recv = %{bc_value => %{n_3: true, n_4: true}}
  peers = [:n_1, :n_2, :n_3, :n_4]

  # Act
  {:ok, rbc, msgs, output} =
    handle_message.(
      %Wasper.HBBFT.RBC.Bracha{
        me: me,
        broadcaster: :n_3,
        peers: peers,
        n: 4,
        f: 1,
        ready_sent: false,
        echo_recv: echo_recv,
        predicate: fn _ -> true end,
        max_msg_size: 1000,
        output: nil
      },
      msg
    )

  # Assert whether new message was added to echo_recv collection
  assert rbc.echo_recv == %{bc_value => %{n_2: true, n_3: true, n_4: true}}

  # Assert whether new message was added to msg_recv collection
  assert Map.get(rbc.msg_recv, {:n_2, :ECHO}) == true

  # Assert whether ready_sent flag was set to true
  assert rbc.ready_sent == true

  # Assert whether READY message was sent to all peers
  Enum.each(peers, fn peer ->
     peer_msgs = Map.get(msgs, peer)
```

```
    assert Enum.any?(peer_msgs, fn {type, from, value} ->
       type == :READY and from == me and value == bc_value
    end)
  end)

  # Assert whether the output was not set
  assert output == nil
end
```

Listing 19: Unit test for verifying that `READY` message was sent.

6.4.2 Completeness

The completeness of translation rules, covered in section 6.1, was calculated in two ways. In a first way, we have divided the total count of translation rules by the count of the total count of possible AST constructions in the newest Elixir version (v1.14.3)[10]. The covered AST nodes percentage is 47%, which shows that translation rules cover almost half of all possible AST constructions.

Table 1. Coverage of Elixir's AST constructions

Count of Elixir's AST constructions in the standard Elixir library (v1.14.3)	66
Count of Elixir's AST constructions covered in translation rules	31
A percentage of covered Elixir's AST constructions	∼47%

Even though the first way of completeness calculation shows the coverage of AST constructions, the coverage of the whole Elixir language remains unknown. That's why we have decided to calculate the completeness in a second way – by diving total count of translation rules from the count of all available functions in the latest version Elixir standard library (v1.14.3)[11].

Table 2. Coverage of Elixir's language

Count of functions in the standard Elixir library (v1.14.3)	1174
Count of Elixir functions covered in translation rules	40
A percentage of covered standard Elixir library	∼3.4%

The approximate percentage of translated standard Elixir functions is 3.4% (see 2). The percentage is rather low due to the fact that Elixir is a pretty extensive language and not only distributed algorithms could be developed. Thus, it has a lot of different functions for various use cases. Only a very small portion of the language's features was used when developing a distributed Bracha reliable broadcast algorithm, which is presented in the experiment section (see 6.3).

[10] https://hexdocs.pm/elixir/1.14.3/syntax-reference.html#the-elixir-ast
[11] https://hexdocs.pm/elixir/1.14.3

6.5 Discussion

This section provides the current solution's advantages, disadvantages, and limitations. Advantages of the implemented translation method:

- We leverage the power of additional constructs of PlusCal. Thus, the user of the generator library does not need to know the TLA$^+$ in depth in order to extend the library with new translation rules – PlusCal resembles an imperative programming language;
- As the solution was implemented as a direct parser from Elixir AST to PlusCal, it is easy to understand and debug it;
- PlusCal is a mature language with great tools developed and tested for over 13 years. Thus, the translation to TLA$^+$ is instant and does not cause any performance issues;
- PlusCal is specifically designed for writing and debugging multi-process algorithms. As we saw in our experiment, a PlusCal process resembled the actual process of a peer.

Disadvantages of the implemented translation method:

- It is hard to prove the TLA$^+$ specifications generated from PlusCal with TLA Proof System (TLAPS) [Lam];
- TLC model checker's performance is a bit poorer for TLA$^+$ specifications generated from PlusCal [Lam].

Limitations of the current solution:

- Only a limited amount of translation rules is currently supported by the tool. For simplicity, only the translation rules that were needed for the experiment were defined and developed;
- The function with one clause can be translated. For simplicity, the translation of only one function clause was developed;
- Expression inside the expression is currently not supported, as it would need a more complicated mechanism to keep track of function calls and insertion of results.

Based on the experiment and observations above, we can claim that our experiment was carried out successfully. With defined Elixir to PlusCal translation rules, that were inspired by the TLA$^+$ Transmutation tool, we have managed to translate the code to algorithmic language. This was beneficial as it made the translation tool simpler and more scalable. Using the standard TLA$^+$ tools, the generated PlusCal was compiled to TLA$^+$ and model-checked. The correctness of the generated TLA$^+$ specification was checked with specification refinement, checking that generated specification implements the abstract one. In order to

show that defined translation rules do not lose any important information, we have generated the Elixir code from the generated TLA$^+$ specification and tested the generated Elixir code with the unit tests. As the verification of the correctness was successful, we can claim that the translation tool is working correctly. Currently, the main limitations of the implemented solution are caused by a lack of features, but it is possible to solve these issues by spending more time on the tool. However, the successful experiment shows that it is possible to generate the TLA$^+$ specifications out of the sequential parts of Elixir code.

Results and Conclusions

Results:

1. A set of generic transformation rules for extracting the TLA$^+$ specifications via the PlusCal algorithm from the code of Elixir programming language was described.
2. A program that extracts TLA$^+$ specifications from the sequential part of Elixir code was developed.
3. A program that extracts Elixir code from the PlusCal algorithm for translation validation was developed.

Conclusions:

1. It is possible to extract the TLA$^+$ specifications from the sequential parts of Elixir code. The correctness of generated specifications can be verified through model checking, specification refinement, and translation validation.
2. The utilization of an intermediate PlusCal language has certain drawbacks, such as slower model checking and increased complexity in proving specifications using the TLA$^+$ proof system (TLAPS). However, the similarity to imperative programming languages simplifies the creation of new translation rules and makes the translation tool easier to extend.
3. Given that the translation tool is implemented as an Elixir library and standard tools are accessed via the command line interface, it is feasible to integrate the TLA$^+$ specification generation and model-checking process into the build process. This integration allows the identification of discrepancies or unintended modifications in the algorithm at an earlier stage, thereby reducing the risk of encountering issues in the production environment.

References

[AT20] AdaCore and Thales. *Implementation Guidance for the Adoption of SPARK*. AdaCore and Thales, 2020.

[BCD+14] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56:1012–1032, 2014.

[Bez05] J. Bezivin. On the unification power of models. *Softw. Syst. Model*, 4:171–188, 2005.

[Bez08] J. Bezivin. ATL: A model transformation tool. *Science of Computer Programming*, 72:31–39, 2008.

[BL02] B. Batson and L. Lamport. High-Level Specifications: Lessons from Industry. In *Formal Methods for Components and Objects*, pp. 242–262. Springer, 2002.

[BP23] D. Bražėnas and K. Petrauskas. TLA$^+$ specifikacijų išskyrimas iš Elixir programos. In *Lietuvos magistrantų informatikos ir IT tyrimai*, pp. 5–14, 2023. URL: `https://www.zurnalai.vu.lt/open-series/article/view/32213`.

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

[CC90] E. J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: a taxonomy. *IEE Software*, 7:13–17, 1990.

[CDH+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering*, pp. 439–448, 2000.

[CGM07] D. Cansell, J. P. Gibson, and D. Mery. Refinement: A Constructive Approach to Formal Software Design for a Secure e-voting Interface. *Electronic Notes in Theoretical Computer Science*, 183:39–55, 2007.

[CHV+18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*. Springer, 2018.

[DeL04] R. DeLine. Spec# Home Page. `https://www.microsoft.com/en-us/research/project/spec/`, 2004. accessed 2021-11-20.

[Dev] Devboost. EMFText. `https://devboost.github.io/EMFText/`. accessed 2022-03-10.

[DXR21] S. Das, Z. Xiang, and L. Ren. Asynchronous Data Dissemination and Its Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2705–2721, 2021.

[Ecla] Eclipse. Acceleo Home Page. `https://www.eclipse.org/acceleo/`. accessed 2022-04-07.

[Eclb] Eclipse. Xtext. https://www.eclipse.org/Xtext. accessed 2022-03-10.

[Fav04] J. M. Favre. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, 2004.

[FBT+02] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothi. Columbus - Reverse engineering tool and schema for C++. In *Conference on Software Maintenance*, pp. 172–181, 2002.

[FPM+19] F. Ferrarotti, J. Pichler, M. Moser, and G. Buchgeher. Extracting High-Level System Specifications from Source Code via Abstract State Machines. In *Model and Data Engineering: 9th International Conference*, pp. 267–286. Springer, 2019.

[Groa] Object Management Group. Abstract Syntax Tree Metamodel. https://www.omg.org/spec/ASTM. accessed 2022-03-10.

[Grob] Object Management Group. Architecture-Driven Modernization. https://www.omg.org/adm/. accessed 2022-03-10.

[Groc] Object Management Group. Knowledge Discovery Metamodel. https://www.omg.org/spec/KDM/1.4/About-KDM/. accessed 2022-03-10.

[HT06] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45:451–461, 2006.

[IM12] J.C. Izquierdo and J. Molina. Extracting Models from Source Code in Software Modernization. *Software & Systems Modeling*, 13, 2012.

[Lam] Leslie Lamport. PlusCal Tutorial. https://lamport.azurewebsites.net/tla/tutorial/intro.html. accessed 2023-01-01.

[Lam02] L. Lamport. *Specifying Systems. The TLA+ Language and Tools for Hardware and Software Engineers*. Microsoft Research, 2002.

[Lam05] L. Lamport. Fast Paxos, 2005.

[Lam06] L. Lamport. Checking a Multithreaded Algorithm with +CAL. In *In Distributed Computing: 20th International Conference*, pp. 151–163. Springer, 2006.

[Lam21] Leslie Lamport. A PlusCal Users Manual. P Syntax. https://lamport.azurewebsites.net/tla/p-manual.pdf, 2021. accessed 2022-12-12.

[LH08] E. I. Leonard and C. L. Heitmeyer. Automatic Program Generation from Formal Specifications using APTS. In *Automatic Program Development*, pp. 93–113. Springer, 2008.

[LMW11] T. Lu, S. Merz, and C. Weidenbach. Towards Verification of the Pastry Protocol Using TLA+. In *Formal Techniques for Distributed Systems: 13th International Conference*, pp. 244–258. Springer, 2011.

[LST+01] L. Lamport, M. Sharma, M. Tuttle, and Y. Yu. The Wildfire Challenge Problem, 2001. URL: https://www.microsoft.com/en-us/research/publication/wildfire-challenge-problem/.

[Maf19] Gabriela Moreira Mafra. Tradução automática de especificação formal modelada em TLA+ para linguagem de programação, 2019.

[McC15] C. McCord. *Metaprogramming Elixir. Write Less Code, Get More Done (and Have Fun!)* The Pragmatic Bookshelf, 2015.

[Met] Metacase. MetEdit+ Home Page. https://www.metacase.com/mep/. accessed 2022-04-07.

[MLH+06] A. Methni, M. Lemerre, B.B. Hedia, S. Haddad, and K. Barkaoui. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Generative Programming and Component Engineering*, pp. 249–254, 2006.

[MLH+14] A. Methni, M. Lemerre, B.B. Hedia, S. Haddad, and K. Barkaoui. Specifying and Verifying Concurrent C Programs with TLA+. In vol. 476, pp. 206–222, 2014.

[New14] C. Newcombe. Why Amazon Chose TLA+. In *Abstract State Machines Alloy, B, TLA, VDM, and Z: 4th International Conference*, pp. 25–38. Springer, 2014.

[NRZ+15] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58:66–73, 2015.

[NSE] NSERC. TXL. http://www.txl.ca/. accessed 2022-03-10.

[PA19] D. Patterson and A. Ahmed. The Next 700 Compiler Correctness Theorems (Functional Pearl). In *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 85–114, 2019.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 151–166, 1998.

[RFZ17] C. Raibulet, F. Arcelli Fontana, and M. Zanoni. Model-Driven Reverse Engineering Approaches: A Systematic Literature Review. *IEEE Access*, 5:14516–14542, 2017.

[Sch08] M. Scheidgen. Textual Modelling Embedded into Graphical Modelling. In *European Conference on Model Driven Architecture Foundations and Applications*, pp. 153–168, 2008.

[Sch19] Arjan Scherpenisse. The Elixir AST explained using the AST Ninja. https://www.botsquad.com/2019/04/11/the-ast-explained/, 2019. accessed 2022-04-07.

[Sel03] B. Selic. The Pragmatics of Model-driven Development. *IEEE Software*, 20:19–25, 2003.

[Str] Stratego. StrategoXT. http://strategoxt.org/. accessed 2022-03-10.

[Tea] Elixir Team. Elixir Home Page. https://elixir-lang.org/. accessed 2021-11-20.

[Tea11] AWS Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/, 2011. accessed 2021-11-20.

[Tec] Moose Technology. Moose Home Page. https://moosetechnology.org/. accessed 2022-04-07.

[Way18] H. Wayne. *Practical TLA+: Planning Driven Development*. Apress Media, 2018.

[YML99] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications, 1999.