SANTA CATARINA STATE UNIVERSITY – UDESC
CENTER OF TECHNOLOGICAL SCIENCES – CCT
GRADUATE PROGRAM IN APPLIED COMPUTING – PPGCA

GABRIELA MOREIRA

TEST GENERATION FROM TLA$^+$ SPECIFICATIONS

JOINVILLE
2022

GABRIELA MOREIRA

TEST GENERATION FROM TLA$^+$ SPECIFICATIONS

Master thesis submitted to the Computer Science Department at the College of Technological Science of Santa Catarina State University in fulfillment of the partial requirement for the Master's degree in Applied Computing.

Supervisor: Janine Kniess
Co-supervisor: Cristiano Damiani Vasconcellos

JOINVILLE
2022

Para gerar a ficha catalográfica de teses e
dissertações acessar o link:
https://www.udesc.br/bu/manuais/ficha

Moreira, Gabriela
TEST GENERATION FROM TLA$^+$ SPECIFICATIONS / Gabriela Moreira. - Joinville, 2022.
50 p. : il. ; 30 cm.

Supervisor: Janine Kniess.
Co-supervisor: Cristiano Damiani Vasconcellos.
Master's dissertation - Santa Catarina State Univeristy, College of Technological Science, Master's in Applied Computing, Joinville, 2022.

1. Test Generation. 2. Formal Specification. 3. Temporal Logic. 4. Model Checking. 5. TLA$^+$. 6. Elixir. I. Kniess, Janine. II. Vasconcellos, Cristiano Damiani . III. Santa Catarina State Univeristy, College of Technological Science, Master's in Applied Computing. IV. Título.

GABRIELA MOREIRA

TEST GENERATION FROM TLA$^+$ SPECIFICATIONS

Master thesis submitted to the Computer Science Department at the College of Technological Science of Santa Catarina State University in fulfillment of the partial requirement for the Master's degree in Applied Computing.

Supervisor: Janine Kniess
Co-supervisor: Cristiano Damiani Vasconcellos

THESIS COMITTEE:

Janine Kniess, P.h.D.
Santa Catarina State University

Members:

Rodrigo Geraldo Ribeiro, P.h.D
Federal University of Ouro Preto

Adolfo Gustavo Serra Seca Neto, P.h.D
Federal University of Technology - Paraná

Joinville, August 23th 2022

ACKNOWLEDGEMENTS

I am very passionate about this work, and even with all the struggle that came with it, I always found a lot of joy with every new idea I had and every concept I managed to turn into reality. However, I could not have done this only with my joy. My advisors, Janine and Cristiano, were the ones that motivated me to actually get this done in time. Together with my research group Função, they convinced me that I should do this and actually made me commit to a master's program. I would not have finished this without their incentives and high standards.

Still, commitment is not always enough to get something done. I would not have been sane enough to do this work if André had not saved me when I lost my ground. He pulled me up and was by my side every step of the way. His company and my cat's, who is always lying down between me and my keyboard, were the best support I could ask for.

“Vivo em tela viva, tela de cara e coragem
Solta esse seu muro e põe os pés nessa viagem
Meu sonho feliz era chegar e já cair no mar”
(Marina Sena, 2021)

ABSTRACT

Specifying software with formal methods provides the benefit of automatically verifying the specification against desired properties, helping to prevent unwanted behaviors. TLA$^+$ (Temporal Logic of Actions$^+$) is a formal specification language focused on concurrent systems, where behaviors can be hard to predict without a model checker - a tool that checks a given property on all possible behaviors of a specified system. Checked properties, however, can be falsified if any mistakes are made while implementing an executable version of the specified system. This work proposes a tool that generates automated unit and integration tests from the model used by the model checker, making assertions over allowed and unallowed behaviors on the implementation, and thus mitigating its deviance from verified properties. It also designs a more decentralized environment for generated code execution and its tests, based on an oracle abstraction.

Keywords: Test Generation. Formal Specification. Temporal Logic. Model Checking. TLA$^+$. Elixir.

RESUMO

Especificar *software* com métodos formais traz o benefício de verificar automaticamente propriedades esperadas da especificação, ajudando a prevenir comportamentos indesejados. TLA$^+$ (*Temporal Logic of Actions*$^+$ - Lógica Temporal de Ações$^+$) é uma linguagem de especificação formal focada em sistemas concorrentes, onde comportamentos podem ser difíceis de prever sem um *model checker* - uma ferramenta que verifica uma propriedade em todos os possíveis comportamentos de um sistema especificado. Este trabalho propõe uma ferramenta que gera testes automatizados unitários e de integração a partir do modelo usado pelo *model checker*, fazendo asserções sobre comportamentos permitidos e não permitidos na implementação, e assim mitigando possíveis violações das propriedades verificadas. O trabalho também propõe um ambiente mais decentralizado a execução do código e testes gerados, baseando-se em uma abstração de oráculo.

Palavras-chave: Geração de Testes. Especificação Formal. Lógica Temporal. Model Checking. TLA$^+$. Elixir.

CONTENTS

1 INTRODUCTION

Formally specifying software is a valuable practice to increase the trustability of software, providing mechanisms to verify properties of behaviors at the software's design stage. The importance of verifications becomes imperative when designing concurrent systems, where the number of reachable states and behaviors does not fit the designer's mind alone. Therefore, specification methods arise with a focus on concurrent systems, such as Petri net's (PETRI, 1962) and pi-calculus (MILNER; PARROW; WALKER, 1992).

Following these works, Lamport proposes TLA$^+$ in (LAMPORT, 2008) based on the TLA (Temporal Logic of Actions) logic, which extends temporal logic terms allowing predicates over pairs of states, called actions. TLA$^+$ provides a declarative and mathematical syntax to define actions that together will dictate whether a behavior - an infinite sequence of states - is accepted or not by the specification. The state space for a specification can be explored with a model checker, TLC (LAMPORT, 2002) or Apalache (KONNOV; KUKOVEC; TRAN, 2019), to verify the properties' satisfiability, defined as the state or temporal formulas.

Besides design validation, writing a formal specification can also benefit existing running systems. For example, it can help to find bugs and inconsistencies and to enable aggressive optimizations by changing the specification and checking whether it still satisfies the set of properties. These benefits were reported by Amazon Web Services (NEWCOMBE et al., 2015) after using TLA$^+$ in 10 (ten) of its complex systems and, for each one of them, finding bugs or understanding that led to optimizations.

Written formal specifications, however, lack any correspondence with the actual implementation besides the programmer's understanding. Some available tools to mitigate this problem are trace checking, code generation, and test case generation. TLA$^+$, however, still lacks most of this tooling, with no published materials on this matter as of the writing of this dissertation. A previous work (MAFRA, 2019) proposes a code generator, which takes a specification written in TLA$^+$ and generates Elixir (Elixir Core Team,) code, with the main goals of maintaining a cognitive correspondence between actions and functions, prioritizing readability, and facilitating and implantation in distributed and concurrent scenarios. With this cognitive correspondence, it is presumable that the programmer makes changes to the generated code to make it simpler or more performatic. By doing this, the programmer can introduce unallowed behaviors or remove allowed ones. As for test case generation, there are two separate initiatives on model-based testing from TLA$^+$ models: exploring TLC state space (DORMINEY, 2020) and reproducing witness from Apalache's counterexamples or simulation (KUPRIANOV, 2020), with results still to be published.

Automated testing is a well-established method in the software engineering field to mitigate the introduction of errors in working software. In this context, writing automated tests based on a formal specification would not only enhance confidence in making code changes but also help validate the initially generated code itself. Moreover, while writing tests is a manual

task subjected to human-introduced false negatives, it is possible to automate this process by generating test scenarios directly from the specification itself.

A formal specification written in TLA$^+$ contains different information that can be used to generate test scenarios. All the information about states and transitions can be obtained by interpreting the defined actions, similar to the process of generating code, which can be used to assemble some white-box testing - tests where each internal state of the behavior is checked, not only some final result. If any properties are defined, they can also be an input to test generation by running the executable code and checking that the property holds when it is supposed to. The third kind of information is not present in the specification itself but can be obtained by running the specification through a model checker. Model checkers take a specification and a set of temporal formulas and, for each formula, return success if it is true for all behaviors or false if there is at least one behavior that falsifies it - informing which behavior is that, also known as a counterexample - that can be used as a test scenario. TLA$^+$ model checkers can also generate model-compliant traces without properties as input by running simulation.

The set of techniques to generate tests from model information is known as model-based testing, and methods to obtain tests from properties are called property-based testing. While there is extensive research on these two topics, their implementation for TLA$^+$ specifications is a fresh subject being also approached by other researchers, with no published works at the time of this writing. The main challenges regarding minimizing the test suite size and its execution time while meeting a coverage criterion often face the state-explosion problem.

This work proposes a test generation tool using some information from a TLA$^+$ specification, to be used mainly alongside the code generation tool previously proposed, to improve confidence in spec correspondence for both the original generated code and any manual modifications made with no intention of modifying the set of allowed behaviors.

When a bug is found by model checking, it is often not trivial how to 1. Reproduce that bug in production software, and 2. Ensure the bug is fixed after an update to the implementation. This is especially hard when the person writing the specification is not the same as the one implementing the software. By providing pluggable interfaces in the generated code and integration tests with the capability of reproducing traces, the tool described in this work aims to reduce the gap between model and implementation with this problem as a focus.

This work's evaluation is done through case studies, attempting to make it as realistic as possible. For that purpose, some environmental changes in both code generation and the first version of test generation were made, allowing completely decentralized models to be executed without a central unit, relying only on a message-passing abstraction. This improvement has significant relevance to the resulting artifact of this work and, therefore, is considered an initially unplanned contribution.

Motivation for this work relies on improving tooling around TLA$^+$ to incentivize its usage. The problem being addressed is the gap between specification and production systems since no executable artifact is derived from the model or the model checker outputs. The proposed

solution includes improvements to the previously proposed code generator (MAFRA, 2019), which provides executable artifacts from models; and two new approaches to generate tests that can both improve the reliability of the generated code (even if a programmer modifies it) and provide artifacts from model checker results, where outputted counterexamples can be reproduced in the generated code.

This text is organized as follows: Chapter 2 presents the necessary concepts for understanding the test generation process; Chapter 3 reviews other model-based testing works, pointing out information that can be relevant to this research; Chapter 4 presents the proposed strategy for the test generation including which information to use and how to connect it with the implementation; Chapter 5 describes the problematic of decentralization and the solutions to code and test generation; Chapter 6 develops two case studies, the first on a decentralized termination detection algorithm and the second on a real blockchain bug; and Chapter 7 collects final considerations for this stage of the work and presents a plan for next steps.

The complete implementation, including code and test generation and the mentioned examples in this work, is available publicly in the opensource repository at https://github.com/bugarela/tla-transmutation.

2 CONCEPTS

TLA$^+$ is a specification language combining the TLA (Temporal Logic of Actions) logic, and the Zermelo-Fraenkel set theory (ZFC) (MERZ, 2003). TLA was proposed by Lamport (LAMPORT, 1994) and is based on transition systems - specifically Kripke structures - where a set of allowed transitions is modeled as one or more actions. A specification written in TLA$^+$ can be used as the input to a model checker that can check it against properties also defined in TLA$^+$ to whether prove the satisfiability of that property or falsify it with a counterexample. This chapter presents concepts for transition systems, model checking and TLA$^+$ itself.

2.1 TRANSITION SYSTEMS

Transition systems are abstractions to describe the behavior of systems in a mathematically precise and unambiguous manner. They can be understood as directed graphs with states as nodes and model transitions as edges. A state describes some system information at a specific time, and a transition describes how a system can change from one state to another (BAIER; KATOEN, 2008).

A transition system is a tuple $(S, \rightarrow, I)$ where

- S is a set of states,
- $\rightarrow \subseteq S \times S$ is a transition relation, and
- $I \subseteq S$ is a set of initial states.

The transition system is said to be finite if and only if S is finite. In finite transition systems, a behavior given by a sequence of states can still be infinite.

The set of successors of a state s is defined as $Post(s) = \{s' \in S \mid s \rightarrow s'\}$. A transition system is called deterministic if $|I| \leq 1 \wedge \forall s \in S : |Post(s)| \leq 1$. In this work, when it's said that non-determinism happens at a state s, it is meant that $|Post(s)| > 1$.

2.1.1 Kripke structures

Kripke structures are types of transition systems with one additional restriction: $\rightarrow$ must be left total, that is, $\forall s \in S, \exists s' \in S : s \rightarrow s'$. This restriction implies that all behaviors are infinite. In Kripke structures, termination can be expressed as a state with only one self-transition.

A simple example of transition systems is the traffic semaphore, which has three possible states (green, yellow and red) and three possible transitions (green $\rightarrow$ yellow, yellow $\rightarrow$ red, and red $\rightarrow$ green). A traffic semaphore starting at any color ($I = S$) can be represented with the Kripke structure in Figure 1.

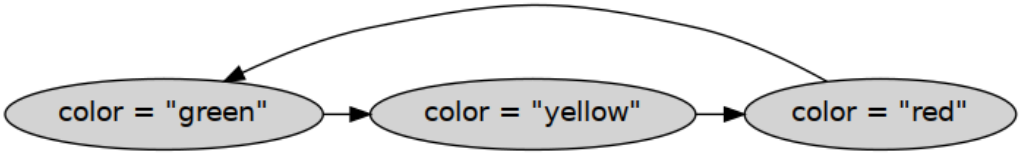

Figure 1 – Kripke structure for a traffic semaphore. Gray states are initial states.

2.2 MODEL CHECKING

From finite transition systems, usually Kripke structures, defining possible behaviors with precision, the system's states can be explored with different techniques such as model checking, where all states are explored exhaustively; simulation, where some paths are chosen to be run; and automated testing, which will try to execute a behavior with a real implementation.

Model checking is a verification technique based on exploring all possible system transitions to prove that a given property is satisfied on all of them or finding a sequence that falsifies it. A model checker works with two different inputs (AMMANN; BLACK; MAJURSKI, 1998):

1. A state machine defined in terms of variables, initial values for the variables and transitions that change these variables when some condition over these variables is met.

2. Invariants and/or temporal formulas expressed as logic properties over the set of the transition system's possible paths.

The model checker conceptually explores all the state space for the defined state machine, checking whether the properties are satisfied. It uses heuristics to navigate all reachable states, where strategies for exploration are already well established. If a property is unsatisfied, the model checker tries to output a counterexample: a sequence of states for which the property does not hold. However, obtaining a counterexample may not be possible for some temporal formulas.

Model checkers present significant scalability issues due to the growth of state space originating from variable addition. For N variables with a domain of k, there can be up to k^N states to explore, which is exponential growth. This is known as the state explosion problem.

When the state space is too large to be explored exhaustively, alternative techniques may be applied to explore implicit regularities in the structure of the model (BAIER; KATOEN, 2008). Symbolic model checkers are alternatives that represent the state space symbolically, commonly using binary decision diagrams, that can then be solved by other tools such as SAT solvers.

2.2.1 Model checkers for TLA$^+$

TLC is the first TLA$^+$ model checker. It is an explicit-state model checker and works by enumerating states produced by all behaviors allowed by a TLA$^+$ specification. It keeps a directed graph with states and transitions, called the reachability graph; and a queue of states for it to check. By starting with evaluating the set of initial states and adding unvisited states

reachable by them to the queue, it can navigate all distinct states checking properties for each of them. However, as with any explicit-state model checker, it is susceptible to the state-explosion problem.

Apalache is a symbolic model checker for TLA$^+$ that translates the specification to a set of quantifier-free logical constraints that can be solved by an SMT (Satisfiability-Modulo-Theory) solver (KONNOV; KUKOVEC; TRAN, 2019). Apalache can only analyze finite executions of bounded length - although it can be an issue for verifying properties, it is not a problem for finding counterexamples to generate tests.

Both model checkers are limited and cannot handle all TLA$^+$ operators. In fact, Lamport states that "no model checker can handle all the specifications that we can write in a language as expressive as TLA$^+$ " (LAMPORT, 2002). These model checkers also assume that:

- All specification parameters are fixed: although specifications can be parameterized, the parameters' values have to be specified in order to run the model checker - they will not check all possible values for the parameters.
- All states are finite structures: reachable states and parameter values are finite structures. All sets and records are finite, functions have finite domains, and so on.

2.3 TEMPORAL LOGIC OF ACTIONS+

TLA$^+$ is a language to specify software focusing on distributed systems based on actions. An action is a predicate over a pair of states called a step. If an action is true for a $s \rightarrow s'$ step, then a transition from s to s' is part of the transition set of the model. Modeling a system with TLA$^+$ means specifying a set of behaviors - sequences of states - possible in that system. This is done by specifying a temporal formula that is true for all the steps in those behaviors and false otherwise. A system's behavior can be checked against other temporal and non-temporal formulas.

In a TLA$^+$ spec, a system can be defined with two formulas: a predicate over the initial state, defining a set of allowed initial states; and an action (often called $Next$) over the system steps, defining a set of allowed transitions. An action can be interpreted as a conjunction of predicates over a current state and predicates over the next state. In this work, predicates over the current state will be called an action's condition - meaning they represent a condition for the transition to happen from that state.

As an example of basic TLA$^+$ usage, consider the same traffic semaphore from Section 2.1.1. This transition system could be written as the $TrafficSemaphore$ TLA$^+$ spec presented in Figure 2. In this spec, $color$ evaluates the value for the variable color in the first state of a step, while $color'$ represents the value for the second state. The $Next$ definition lists three possible actions, and each one of them has a condition (predicate over the current color) and a transition (which color to turn to). Although the TLA$^+$ tools allow the definition of a spec with

MODULE $TrafficSemaphore$

VARIABLE $color$

$TurnRed \triangleq color = \text{“red”} \land color' = \text{“green”}$
$TurnYellow \triangleq color = \text{“green”} \land color' = \text{“yellow”}$
$TurnGreen \triangleq color = \text{“yellow”} \land color' = \text{“red”}$

$Init \triangleq color \in \{\text{“red”}, \text{“yellow”}, \text{“green”}\}$

$Next \triangleq TurnRed \lor TurnYellow \lor TurnGreen$

Figure 2 – TLA$^+$ Specification for the traffic semaphore

the identification of an $Init$ and a $Next$ definitions, the same spec could also be defined with the temporal formula:

$$Spec \triangleq Init \land [Next]_{\langle color \rangle}$$

TLA$^+$ syntax borrows elements from first-order logic and set theory. The logical operators $\land$ and $\lor$ can be used both prefixed and infixed such that:

$$\begin{array}{l} \land\ e_1 \\ \vdots \\ \land\ e_n \end{array} \equiv e_1 \land \cdots \land e_n \qquad and \qquad \begin{array}{l} \lor\ e_1 \\ \vdots \\ \lor\ e_n \end{array} \equiv e_1 \lor \cdots \lor e_n$$

As a temporal logic, TLA$^+$ includes the $\Box$ and $\Diamond$ operators, standing respectively to “always” and “eventually”. $\Box F$ for a formula F is true for a behavior (that is, an execution) if and only if F is true for every step of that behavior; $\Diamond F$ for a formula F is true for a behavior if and only if F is true for at least one step in that behavior.

2.3.1 Stuttering Steps

The $Next$ definition in the presented temporal formula is written as $[Next]_{\langle color \rangle}$ as a way to specify that $Spec$ allows stuttering steps over the variable $color$. A stuttering step represents self-transition, such that an action stating $color = color'$ is always allowed. Stuttering steps are essential to represent loops and crash states. Considering the traffic semaphore example, allowing stuttering steps over $color$ means the semaphore could get stuck forever in any possible colors.

2.3.2 Properties

With a system modeled as a TLA$^+$ spec, desired properties of this system can be verified. For TLA$^+$, properties are temporal formulas or invariants over specified actions. A property is satisfied if and only if it is true for all the specified behaviors.

Safety properties define what a system can do, and a property of this type is violated, it is violated in a specific behavior step (LAMPORT, 2002). They are defined as invariants, where an

invariant is a predicate P that should be true for all the steps of all allowed behaviors of $Spec$, such that the property can be verified by proving

$$Spec \implies \Box P$$

Liveness properties define what a system should do, violating these properties in a behavior. For example, a traffic semaphore must eventually be red. If the system allows a behavior where the semaphore switches only between yellow and green, this liveness property would be violated by that behavior.

Liveness properties can be weak fairness properties, or strong fairness properties (LAMPORT, 2002). Weak fairness for a state formula f and an action A is defined as $WF_f(A)$. It is true for a behavior if and only if a non-stuttering A step is infinitely not enabled or infinite non-stuttering A steps occur. A non-stuttering step can be defined as $A \wedge (f' \neq f)$, which can also be expressed as $\langle A \rangle_f$. Therefore, $WF_f(A)$ is defined as

$$\Box(\text{ENABLED } A \implies \Diamond\langle A \rangle_f)$$

However, if an action is repeatedly enabled and not enabled throughout a behavior, weak fairness makes no guarantee that this action will ever occur in that behavior since it requires that the action is continually enabled to ensure its occurrence. This stronger guarantee is provided by strong fairness. Strong fairness for a state formula f and an action A is defined as $SF_f(A)$. It states that if A is always eventually enabled, then a $\langle A \rangle_f$ step should eventually occur and can be expressed as

$$\Box\Diamond\text{ENABLED } A \implies \Box\Diamond\langle A \rangle_f$$

The model checker is also able to check for deadlocks. A deadlock is a state where no transitions can be made, and for some specifications, it can represent termination. The property asserting that a model never reaches a deadlock is a temporal formula ensuring the following state formula $Next$ is always enabled, granting that a $Next$ step can always be performed.

$$\Box(\text{ENABLED } Next)$$

2.4 TEST GENERATION BY MODEL CHECKING

Automated tests for software are implementations that define an expected result for some code evaluation, run a specific code instruction and compare its output with the expected result, reporting an error when they do not match. Although any code piece with any parameters can be tested, to minimize a test suite (the set of all defined tests) code size and time to run, it's desirable that different tests capture different aspects of the code. A test setup is referred to as a test scenario, including what code is being tested, which values are given as parameters, and

what is the expected outcome. This section describes how relevant test scenarios can be obtained from model checkers.

A counterexample produced by the model checker can be used as a test scenario for the specified system. With this intent, a system's desired properties can be negated to find counterexamples that hold for the original property, which can then be used as a scenario. The negation is necessary since a well-specified system will satisfy all the desired properties and therefore generate no counterexamples.

A property used with the intent of generating a counterexample is called a trap property, and the generated counterexample is called a witness of that property. A trap property is considered feasible if the model checker successfully finds a counterexample for it, and unfeasible if it finishes exploring the state without finding any violations. Unfeasible trap properties cannot be used for generating tests as it is not known how to test the property. There is also the possibility that the model checker cannot explore the entire state space in feasible time because of the state explosion problem. If no counterexample is found and the exploration did not finish, it is unknown whether the trap property is feasible (ARCAINI; GARGANTINI; RICCOBENE, 2015).

Another concept in test generation is test harness, which refers to the implementation that allows for the tests to be run with the actual code. Mapping counterexamples to tests and implementation states to specification states are examples of test harness implementation. Some test generation tools require harness definition from the programmer in order to work.

Chapter 3 gathers more information about test generation by model checking in model-based testing, and Chapter 4 elaborates on the specific techniques used for this work.

3 RELATED WORK

This section describes some existing material on model-based testing, property-based testing and TLA$^+$ specific initiatives. The exploration of this material was the theorical basis for the test generation methodologies proposed in this work.

The work in (HONG et al., 2002) uses a symbolic model checker such as SMV (Symbolic Model Verifier) to obtain states, controlling the state explosion problem. Models are defined as EFSMs (Extended Finite State Machines), and properties are written on CTL (Computation Tree Logic), a branching-time temporal logic, to use the model checker to obtain counterexamples to be used as test scenarios. This work also defines formally different coverage criteria that can be used to measure the quality of a test suite: state coverage, transition coverage and data flow-oriented coverage, along with heuristics to apply them. Since a single CTL property can have multiple witnesses, choosing a witness for each property becomes an optimization problem where the suite length and duration should be minimized while the coverage constraints are met. The authors prove this optimization problem to be NP-Hard.

A different approach in (VISSER; PĂSĂREANU; KHURSHID, 2004) uses the Java Pathfinder, a model checker for Java using the JVM, to generate test inputs for complex data manipulation. To efficiently generate tests while keeping high coverage, they use symbolic execution as a form of abstraction, where a set of constraints is calculated over the inputs for each possible executable path. The constraints are then solved to generate a minimal test suite covering all those paths. The authors present a case study on red-black trees where an invariant asserting validity of the tree, accordingly to red-black tree constraints, is checked across behaviors.

The applied work in (XIA; VITO; nOZ, 2005) generates tests from C code and requirements in the form of properties. The C program is abstracted using predicate abstraction, and any unfeasible paths found later by the model checker can help to refine this abstraction. The model checker receives the abstraction and the requirements as trap properties and outputs a counterexample in the form of an execution path. The path is considered unfeasible if it is not allowed by the original program, and the abstraction is refined by new predicates. If the path is feasible, its inputs and outputs result in a test. The authors apply these techniques to avionics systems, showing an example from a Boeing 737 autopilot simulator.

Another application of model checking-based testing in (MOHALIK et al., 2014) to safety-critical systems is done in General Motors, where model checker-generated inputs are compared to random inputs and guided simulation in a set of real-life case studies at the company. The authors show that these other existing techniques were not enough to provide high test coverage and proof of unreachability, while combining them with model checking-based tools achieved the best result in terms of coverage.

The industrial evaluation in (ENOIU et al., 2016) proposes a test generation technique using a model checker for the Function Block Diagram language, widely used in industrial safety-critical components. The generated tests are meant to complement requirement-based

tests with a structural perspective, and they can be generated using different coverage goals. The proposed tool was used to generate tests and analyze coverage for 157 different programs.

The work in (KOO; MISHRA, 2009) proposes property and design decomposition procedures to make the test generation process more efficient in order to apply it to formally verified microprocessors. The generation technique uses decompositional model checking, and both the model and the properties are decomposed to avoid the state explosion problem. The used model checker is also bound (BMC - Bounded Model Checking), which is an approach to limit the number of transitions made by the model checker. With SAT-based BMC, used in this work, after the BMC reaches the transition limit, the model is converted to a satisfiability problem and sent to a SAT solver to obtain a counterexample.

Another decompositional technique in (ARCAINI; GARGANTINI; RICCOBENE, 2015) decomposes the system model into subsystems by exploiting the variables' dependency in a way where a subsystem's output is used as part of another subsystem's input. These subsystems are then sent to the model checker to obtain counterexamples which are later combined to obtain tests for the system as a whole. The proposed technique reduces test generation time by avoiding the state explosion problem by checking smaller subsystems instead of the system itself.

A model checker is used in (ROBINSON-MALLETT et al., 2006) to generate characterization sets used in black-box testing. Black-box testing is a type of test that inputs values to a system and check its outputs without being able to see the internal state. Characterization sets are useful in black-box testing since each input sequence on the set produces different outputs for each different initial state.

The reviewed works showed different approaches to the test generation process using model checking, where some of them focused on improving efficiency, others on coverage criteria, and many had objectives that were a combination of both, showing how they relate. The test generation works used a similar set of strategies with variations mainly involving efficiency improvements and the area or language of the application. The work proposed here also uses some of the observed strategies and concepts, including the usage of negated versions of satisfied properties to obtain counterexamples and differentiation between black-box and white-box testing. This work's primary variation is the application to the TLA$^+$ language, since similar works in TLA$^+$ are not present in the literature.

3.1 RELATED WORKS IN TLA$^+$

There are no published works generating tests from TLA$^+$ specification at the time of the writing of this dissertation. However, some works in progress were presented at the 2020 TLA$^+$ Conference, such as a model-based test generator based on TLC called Kayfabe (DORMINEY, 2020). This approach uses the model itself that TLC enumerates to make its exploration, checking that every step of the program is a refinement of the model's state. In order to explore the entire state space efficiently, the method would need to find a state tour that minimizes revisited states,

which, as the author shows, reduces to the NP-Hard traveling salesman problem. It is also shown that test harness can have some limitations on how to map TLA$^+$ values to concrete ones and on considering some program side effects that are not present on the model.

Another work by Informal Systems, the company behind the Apalache symbolic model checker for TLA$^+$, obtains tests from counterexamples produced by Apalache itself (KUPRIANOV, 2020). The test generation technique is part of the verification-driven development process (VDD) developed at the company, which begins with an English specification, that is manually translated into a TLA$^+$ specification and later to code. The model-based testing reduces the gap between specifications and implementations in this flow.

4 TEST GENERATION

The proposed work aims to produce test scenarios automatically and test the code generated by (MAFRA, 2019) using the same TLA$^+$ specification. While the code generator uses the model specification alone, a test generator can benefit from the usage of other information such as specified properties and interactions with a model checker. This additional information can be used to obtain valid behaviors accordingly to the specification, which can then be used to test the generated code since a valid behavior for the specification must be valid in the implementation. If any behavior allowed by the specification is not allowed in the implementation, there is an error in the translation process, or some error was introduced by modifying the generated code.

Generated tests run against Elixir code (Elixir Core Team,) that is generated by the same tool. Elixir is a functional programming language heavily based on Erlang (ARMSTRONG; VIRDING; WILLIAMS, 1991) process communication and Ruby (COMMUNITY, 2021) syntax. The motivation for generating Elixir code instead of any other language 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 functional paradigm is declarative over operational, similar to the TLA$^+$ syntax. This helps to keep correspondence between definitions in each language.

3. Elixir is focused on legibility and maintainability, borrowing a lot from Ruby's syntax, which is desirable to provide an artifact that can actually be adapted and upgraded, making it more suitable to run in a real-life environment.

4. The BEAM provides platform transparency, making the generated code suitable for a more significant number of environments.

5. Elixir is open source under the Apache 2.0 license, which is crucial to ensure further correspondence between code and spec on all levels.

After obtaining test scenarios, the generator must be able to couple them with the code in order to run them. The scenario's initial state has to become an input, and the program's execution behavior must be comparable with the sequence of states from the rest of the scenario.

From behaviors obtained either from the state exploration graph or a counterexample, executing tests means checking that each behavior should be executable in the implementation. Since this work focuses on generating tests for code generated by (MAFRA, 2019), the proposed harness is restricted to these code interfaces, but it is possible to adapt everything to other implementations with similar interfaces.

Generated tests are static and only need to be re-generated if the model or the properties change. There is no coupling with the model checker or the test generator after the tests are generated. The test suite should run independently within the Elixir project.

This chapter describes two different test approaches: white-box and black-box tests. White-box tests check internal states, while black-box tests are restricted to checking output. Black-box testing is implemented in this context by checking only stages with non-determinism, which on generated code turns into interaction with external factors through the oracle. The oracle can be used to choose an action accordingly to the behavior being tested, and the test should fail if the choice is not possible. These black-box tests act like integration tests in software engineering terminology and can also be used as regression tests. When a bug is found in an application being developed, a test is often written to reproduce the bug scenario and expect the correct outcome, so further updates after the bug fix cannot re-introduce it without breaking a test. Model checking specifications of software can also find bugs, and this sort of test generation provides an automated form of verifying the bug scenario's outcome in the implementation.

In contrast, white-box tests act like unit tests, where all implementation states, including internal ones, are checked against model information. An internal state is any state that is not directly influenced by the user or exposed to them. In some software testing applications, internal states are not accessible, and thus, white-box testing is impossible. In this work's scenario, however, both model and implementation internal states are available since the specification is a requirement to test generation, and the implementation is generated to the `next` function that can be called repeatedly to obtain any state.

To exemplify some techniques, the didactic TLA$^+$ specification for multiple traffic semaphores is given in Figure 3. This specification consider a set of semaphores under the following requirements:

1. The sequence of colors for each semaphore should always be "red", "green", "yellow" and back to "red";
2. Two different semaphores should never be "green" at the same time;
3. Opening of semaphores (turning "green") should alternate.

In this spec, alternation is controlled by the $next\_to\_open$ variable, which cycles through the list of semaphores, and a specific semaphore can only turn green if all semaphores are red, and it is the next to open. The following properties are defined for the specification based on the requirements:

$$no\_collision \triangleq Cardinality(\{s \in SEMAPHORES : colors[s] = \text{“green”}\}) \leq 1$$
$$never\_stuck \triangleq \mathrm{WF}_{\langle colors \rangle}(Next) \implies \forall\, s \in SEMAPHORES : \Diamond(colors[s] = \text{“green”})$$

$no\_collision$ is an invariant establishing that at every state, the number of green semaphores should be less than or equal to 1. $never\_stuck$ is a temporal formula asserting that if

MODULE *TrafficSemaphores*

EXTENDS $Integers, FiniteSets$
VARIABLE $colors, next\_to\_open$
CONSTANT $SEMAPHORES$

$$
\begin{aligned}
TurnGreen(s) \triangleq\ & \land \forall\, s2 \in SEMAPHORES : colors[s2] = \text{“red”} \\
& \land colors' = [colors \text{ EXCEPT } ![s] = \text{“green”}] \\
& \land next\_to\_open = s \\
& \land next\_to\_open' = (s+1)\%Cardinality(SEMAPHORES)
\end{aligned}
$$

$$
\begin{aligned}
TurnYellow(s) \triangleq\ & \land colors[s] = \text{“green”} \\
& \land\ colors' = [colors \text{ EXCEPT } ![s] = \text{“yellow”}] \\
& \land \text{ UNCHANGED } \langle next\_to\_open \rangle
\end{aligned}
$$

$$
\begin{aligned}
TurnRed(s) \triangleq\ & \land colors[s] = \text{“yellow”} \\
& \land\ colors' = [colors \text{ EXCEPT } ![s] = \text{“red”}] \\
& \land \text{ UNCHANGED } \langle next\_to\_open \rangle
\end{aligned}
$$

$$
\begin{aligned}
Init \triangleq\ & \land colors = [s \in SEMAPHORES \mapsto \text{“red”}] \\
& \land next\_to\_open = 0
\end{aligned}
$$

$$Next \triangleq \exists\, s \in SEMAPHORES : TurnGreen(s) \lor TurnYellow(s) \lor TurnRed(s)$$

$$Spec \triangleq Init \land \Box[Next]_{\langle colors,\, next\_to\_open \rangle}$$

Figure 3 – TLA$^+$ Specification for multiple traffic semaphores working together

weak fairness of the $Next$ state action for the variable $colors$ holds, then all semaphores should eventually be green. The weak fairness condition for this property is necessary since it would never hold when stuttering steps over $colors$ occur forever.

4.1 WHITE-BOX TESTING

White-box tests are granular unit tests that look inside the implementation, checking that every transition is correct. It does so by comparing the transitions from every state to a graph of states explored by the TLC model checker.

4.1.1 Specification information to obtain scenarios

Obtaining scenarios directly from the TLA$^+$ spec would be susceptible to errors in the translation process between TLA$^+$ structures and test structures. The transition system could instead be obtained directly from the TLC model checker, which exhaustively explores all states and can output which states were visited. This information is provided exclusively by the model checker TLC. TLC is an explicit state model checker and therefore enumerates all states and transitions while model checking a specification. The produced information about states and transitions is the main source of information for generating white-box tests. This information is unavailable in the Apalache model checker as it does not enumerate all states.

When running TLC, it is possible to ask it to dump explored states into a file. It generates a `.dot` file composing the states in a graph that can be rendered with GraphViz to obtain a visual representation such as the one in Figure 4 generated from the Traffic Semaphores spec with $SEMAPHORES = 0, 1$. Any tour on this graph beginning on a valid initial state should be possible in the implementation - if not, then the implementation is incorrect.

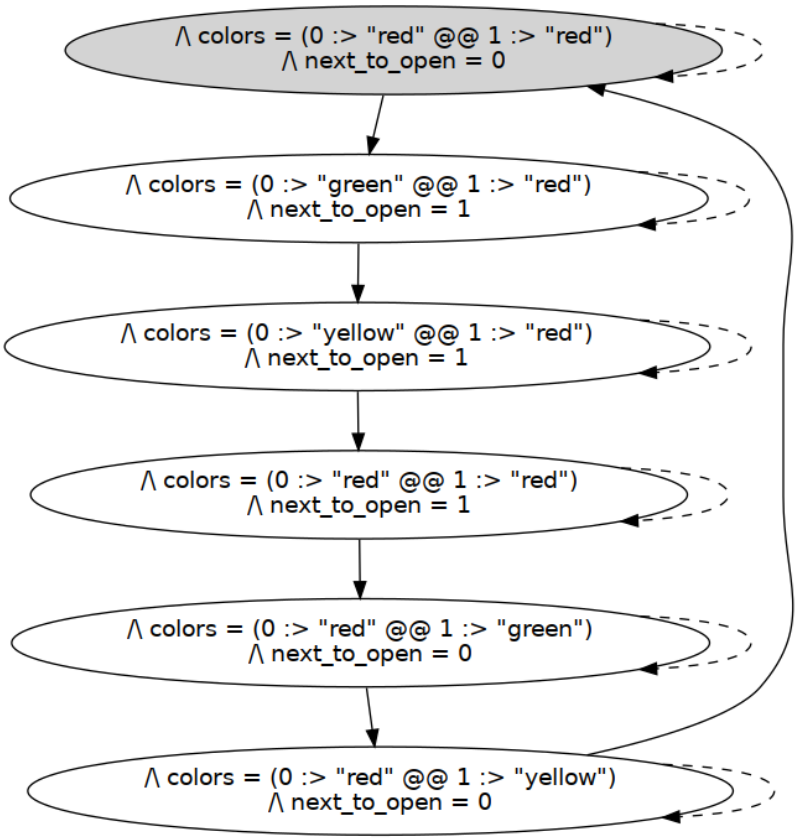

Figure 4 – State graph for the traffic semaphore spec generated by TLC. Gray states are initial states.

A limitation of this graph generation feature is that it is not able to inform which actions are used to generate each transition. The only information about the transition is the two states it is connecting. Therefore, in order to apply these transitions to the implementation, there are two options:

1. Derive which action would make the transition possible from the TLA$^+$ model and choose it, or

2. Inspect the internal state that would result from each possible action and choose the one that results in the transition result state.

Option 2 was shown to be a good interface solution as it was also useful for handling counterexample information.

In order to generate a complete graph, TLC has to enumerate all states. Therefore, when running TLC to generate the state graph, the property being checked should be satisfied. If it is not, TLC can find a counterexample before exploring the entire state space and output a partial state graph. For that, a simple property as $MyProperty \triangleq TRUE$ can be used.

The output format is visualization-driven, mainly containing graphical information like colors and coordinates. A simple script was written to clean up this output into a normalized and filtered `.json` file that can be easily parsed by the tool. In this script, the `dot` CLI tool is used to convert the `.dot` file to JSON format that is then processed with `jq` to filter out noise information.

4.1.2 Test Harness

The following state action is the main definition of the system's state transitions. Upon informing the translation tool that an action $Action \triangleq A$ is the next state action, its name will be translated as `main`, since it should be invoked at the start of the program's execution. In its original implementation, the function receives the state, represented as a hash, as its parameter and invokes itself recursively with the result of the application of its transition over the received state.

Since the `main` function itself does the loop control, there is no external way of stopping to make checks at each state, which would be necessary for testing. Therefore, this implementation had to be slightly changed in order to keep the loop control outside this function, in a way where there is a function that can be called to execute a single transition. This function is called `next` and takes a state as an argument and returns a list of possible next states. The `main` function is kept as a one-point interface called at the beginning of execution, which calls `next` repeatedly and uses the oracle to chose transitions from that given list when it has more than one possible transition.

Using this new `next` interface, white-box tests are generated from the JSON state graph to output a unit test file where each test calls `next` for a state, represented by a node in the graph, and asserts that it returns a list matching the transitions from that state, represented by outgoing edges in the graph. To run tests, Elixir provides a built-in testing library called ExUnit (TEAM, 2021) that is used to execute behaviors and make assertions.

4.1.3 Example

From the state graph in Figure 4, a test file with 118 lines is generated with 6 unit tests (one per state/node). One of them is presented below.

```
test "fromState 7" do
  variables = %{
    colors: %{0 => "red", 1 => "red"},
    next_to_open: 0
```

```
  }

  expectedStates = [
    %{
      colors: %{0 => "green", 1 => "red"},
      next_to_open: 1
    }
  ]

  actions = TrafficSemaphores.next(variables)
  states = Enum.map(actions, fn action -> action[:state] end)

  assert Enum.sort(Enum.uniq(states)) == Enum.sort(Enum.uniq(expectedStates))
end
```

The unit tests can be run with the Mix build tool, which includes ExUnit support, by running `mix test`. When there is a failure, the output shows the two-state lists for comparison, and it is easy to spot the difference between them (which is also colored in the terminal emulator). A failure example for the presented unit test is given below.

```
1) test fromState 7 (TrafficSemaphoresTest)
   test/generated_code/traffic_semaphores_test.exs:4
   Assertion with == failed
   code:  assert Enum.sort(Enum.uniq(states)) ==
          Enum.sort(Enum.uniq(expectedStates))
   left:  [
            %{colors: %{0 => "green", 1 => "red"}, next_to_open: 0},
            %{colors: %{0 => "red", 1 => "green"}, next_to_open: 0}
          ]
   right: [%{colors: %{0 => "green", 1 => "red"}, next_to_open: 1}]
   stacktrace:
     test/generated_code/traffic_semaphores_test.exs:18: (test)
```

4.2 BLACK-BOX TESTING

Black-box tests make assertions and guide executions using only the public interfaces of the implementation, that is, the `main()` function and the oracle. A black-box test tries to reproduce a trace in the implementation.

4.2.1 Specification information to obtain scenarios

The specified model contains enough information about the transition system to generate tests for every state or every transition. However, using only the model information can be inefficient since the problem of finding routes minimizing revisited states is a NP-Hard problem, as shown in (DORMINEY, 2020).

Besides the model, a specification should contain properties to be verified. Section 3 mentions work related to model-based and property-based testing. A property can be translated directly to tests, where an invariant can be an assertion on each state of any execution, and temporal properties can check states accordingly to the timeline it imposes. Properties can also be used to generate tests in conjunction with the model checker by obtaining counterexamples. When asking a model checker to check properties that are not satisfied, it will try to produce a counterexample for that property. This counterexample is a behavior (sequence of states) that is allowed by the model and does not satisfy the property. This work focuses on model-based testing using traces produced by model checking to obtain scenarios rather than using the properties alone.

Both model checkers (TLC and Apalache) can be used to obtain counterexamples, as their output format is similar. Therefore, one can benefit from trying both tools and using the most performant one for a spec to generate counterexample-based tests. The two of them also provide simulation functionality, which can be used to obtain samples of traces that respect the model without a property dependency. The simulation functionality is a viable alternative for witness-based test generation when there are few or no good properties specified.

A usual specification, however, will not have any unsatisfied properties ready to be used. To obtain unsatisfied properties that can be used to generate counterexamples, the provided satisfied properties have to be negated and turned into trap properties. A property can also be decomposed into smaller properties to generate different (and often faster) counterexamples. One simple way of altering the property to obtain different witnesses is to negate different parts of it.

Taking the $never\_stuck$ property, for example, it is an implication $A \implies B$ and thus can be rewritten as $\neg A \vee B$. Negating the whole property would result in $A \wedge \neg B$, so any state that satisfies B or doesn't satisfy A would be part of a counterexample - and this is too general, often the initial state will be a valid counterexample for itself, which is not an interesting behavior to check. Generally, a counterexample will be better if it is a longer sequence since it would test more transitions in the implementation. In this case, another possible negation would be $A \implies \neg B$. A witness for this trap property would be a sequence where A holds but B does not, which is more restricted and, therefore, will take more transitions to be falsified and thus can generate a better counterexample. Therefore, the trap property for $never_s tuck$ is built with the negation of the implication result, ensuring that the witness occurs in a behavior where the weak fairness holds:

$$trap \triangleq \mathrm{WF}_{\langle colors \rangle}(Next) \implies \neg \forall\, s \in SEMAPHORES : \Diamond(colors[s] = \text{“green”})$$

If a sequence of actions is produced as a valid counterexample for a model, then that model's implementation should allow this exact sequence of actions. Therefore an automated test can be made by providing to the implementation the necessary inputs that will lead to this behavior and checking if the behavior can be executed.

In the strategy for deriving trap properties, elaborating heuristics is not necessary to improve property negation to obtain better counterexamples. Instead, the model checker can check different negations for the same property. Then, the test generator tool can choose between feasible properties, which counterexamples to use by a simple criterion such as sequence size. However, this is a step of the test generation with low manual effort and a significant impact on the results, so it is left for the tool user to decide which properties and their counterexamples or simulation results are more relevant for testing, and automation of this selection can be explored in a future work.

A similar trace can be obtained by running the model checker in simulation mode, where an arbitrary number of traces of arbitrary length can be generated randomly, without the need for a property to be contradicted. However, testing random simulation can be less interesting than testing a trace related to a property since properties are usually important desired behaviors specified by the specification designers.

Model checkers also provide options to generate multiple traces in the same run, such as the maximum number of counterexamples to generate and a view. The view is a user-defined state formula used by the model checker to determine if two counterexamples are different by evaluating the view for each state in the counterexamples. The states are considered different if and only if their views are different. This is useful to obtain all relevant scenarios in a single run, i.e., one where a variable is negative, another where its value is zero, and a third with a positive value.

A specification defines a set of behaviors, and the implementation should be able to execute that same set. A trace can either be or not be part of this set: if it is, then it is expected that the implementation can execute it; otherwise, it is expected that it's impossible to reproduce it in the implementation. The strategies listed up to this point described how to obtain valid traces, but it is also possible to have black-box tests asserting that invalid traces are not reproducible. One source of invalid testing traces worth testing can be old specification versions containing bugs that led to undesired behaviors. If it is undesired in the model, it is also undesired in the implementation.

4.2.2 Test Harness

The oracle is the other interface that comes to action when some non-determinism needs solving. Non-determinism on TLA$^+$ occurs in a state where the $Next$ formula is true for more than one step starting on that state - that is $\exists\ s1, s2 \in states \mid Next(s \to s1) \wedge Next(s \to s2)$. There are different ways to go about it: the execution could be divided into branches running on different parallel processes, determined by a random factor, or prompt the user for a choice. This

will depend heavily on the scenario in which the generated code is running.

The chosen approach to handle such non-determinism is to isolate it with a concept called oracle. Every generated code has an oracle process, and the main process sends a message to this oracle every time it needs to decide something. The programmer can implement the oracle decision behavior and has a simple interface: it receives a message with a list of possible transitions to take and responds with the chosen action.

From a concurrency point of view, the oracle serializes messages so that the generated code only processes one thing simultaneously. This is consistent with the definition that all non-determinism is isolated to the oracle since concurrency is a kind of non-determinism. Deterministic segments of the spec - sequences of transitions where only one action is enabled at the time - can be viewed as critical zones or atomic operations, which are crucial to handling concurrent systems consistently.

Algorithm 1 Algorithm for the TraceCheckerOracle

```
1:  procedure START(trace, expected_result, step, model)
2:      if step  trace length then                          ▷ All steps were reproduced
3:          exit with: SUCCESS if expected_result was true, else FAILURE
4:      current_variables ← trace at step
5:      message ← wait for message
6:      if message is a choice request then
7:          respond with the action with state equal to trace at step + 1, or with "cancel"if not
    found
8:          start(trace, expected_result, step, model)
9:      else                                          ▷ message is a new state notification
10:         if new state is equal to trace at step + 1 then
11:             start(trace, expected_result, step + 1, model)
12:         else
13:             exit with: FAILURE if expected_result was true, else SUCCESS
```

The only needed change at the oracle interface for test generation was an additional message, where the `main` function notifies the oracle of a transition it does. This is necessary so the oracle can keep track of a trace's execution even in steps fully deterministic, where it would not receive any information in the original implementation. With this update, a new test-specialized oracle was implemented with the capacity to enforce a trace's behavior upon the generated code and, if it is correct, reproduce that trace. The algorithm for this new oracle is presented at Algorithm 1 and consists of keeping a step counter and reacting to the two possible messages: request choices, where it receives a list of actions and responds with the one that matches the next desired state on the trace; and notifications, where it checks whether the transition was correctly made and updates its step counter.

4.2.3 Example

The perhaps most interesting example traces to be reproduced traces that once represented a bug scenario. It is a typical software engineering practice to reproduce bugs with automated tests when fixing them in order to prevent future changes from re-introducing the same bug - this is called regression testing, tests that avoid regressing to the state where the bug happens.

As an example, consider a previous version of the traffic semaphore specification previously given in Figure 3, where variable $next\_to\_open$ is never updated and, therefore, there is no alternation control. In this version of the spec, it is possible for one of the semaphores to repeatedly open and close while others stay red forever, which is undesired. The property $never\_stuck$ defines alternation since it states that all semaphores should eventually be green as long as the colors variable is changing, and verifying this property results in the faulty behavior where only one semaphore repeatedly opens (in this case, semaphore 1 changes color where semaphore 0 stays red forever):

```
State 1: <Initial predicate>
/\ colors = (0 :> "red" @@ 1 :> "red")
/\ next_to_open = 0

State 2: <TurnGreen>
/\ colors = (0 :> "red" @@ 1 :> "green")
/\ next_to_open = 0

State 3: <TurnYellow>
/\ colors = (0 :> "red" @@ 1 :> "yellow")
/\ next_to_open = 0

Back to state 1: <TurnRed>
```

By providing this trace (named `NoAlternation`) to the black-box test generator and specifying that the expected result is a failure, the following Elixir task is generated:

```
defmodule Mix.Tasks.NoAlternation do
  use Mix.Task

  @impl Mix.Task
  def run(_) do
    trace = [
      %{
        colors: %{0 => "red", 1 => "red"},
        next_to_open: 0
```

```
      },
      %{
        colors: %{0 => "red", 1 => "green"},
        next_to_open: 0
      },
      %{
        colors: %{0 => "red", 1 => "yellow"},
        next_to_open: 0
      },
      %{
        colors: %{0 => "red", 1 => "red"},
        next_to_open: 0
      }
    ]

    model = spawn(TrafficSemaphores, :main, [self(), Enum.at(trace, 0), 0])
    TraceCheckerOracle.start(trace, false, 0, model)
  end
end
```

Using the build tool Mix, this test can be run with the command `mix no_alternation`, which returns whether the test succeeded or not. In this code, variable `trace` is a direct translation from the given trace, the model is run in the background, and the `TraceCheckerOracle` is the oracle instance that will try to enforce the trace and produce the test output.

With the faulty version of the specification, where $next\_to\_open$ is never updated, the generated code can reproduce this trace and, therefore, the test fails (as the expected result was that the trace should not be reproducible, see the second parameter to `TraceCheckerOracle`). The test succeeds when the specification is fixed to make $next\_to\_open$ be updated correctly as the final spec in Figure 3, and the code is re-generated.

4.3 ADAPTIONS TO THE CODE GENERATOR

Along with the interface improvement from only `main` to `next` and `main`, the ability to process non-determinism in any action was removed. That is, the new interface requires that all non-deterministic choices happen at the `next` level only, meaning it is the only action function that is allowed to return a list of transitions instead of a single one. In order to allow that for all actions, some form of flattening procedure would need to be implemented, as done by the model checkers (KONNOV; KUKOVEC; TRAN, 2019). Since that is not in the scope of this work, the input spec is assumed to be flattened - that is, all non-determinism occurs on the $Next$ action. Any spec can be rewritten to satisfy this constraint.

Regarding parsing of specifications, the original tool implements a prototype of a TLA parser, which evidenced how complex it is to parse pure TLA. Instead, the main parsing procedure in the final version takes a `.json` version of the specification produced by the Apalache parser (which uses SANY, the TLC parser, under the hood). This format has a better-defined structure and is much easier to parse. In addition, the resulting parser can handle a broader set of TLA$^+$ specifications and takes less effort to extend with new language constructs.

Along with minor bug fixes, a relevant improvement was made to allow support of different kinds of definitions. In the original version, there was an assumption that all named definitions referred to an action. However, using the tool with different inputs showed that some specifications define named actions for auxiliary functions and values, which were not appropriately translated as the tool assumed they were actions. Fixing this problem is not trivial since it requires inferring kinds of TLA$^+$ expressions (action or value) across the hole specification. Lamport defines that TLA$^+$ expressions have different levels, which can be inferred (LAMPORT, 2002), but this inference is outside this work's scope. The intermediate solution applied requires the user to define which names of definitions refer to actions in the config file. That is enough to distinguish between values and actions in the conversion from parsed JSON to the tool's internal specification representation.

The infrastructure of how the generated code is initialized along with the oracle is also a change worth noting. While the original tool generated the model and its initializer on the same file, defining the oracle process ID as a constant loaded at initialization which broke some build tool commands; the current version properly generates the model as an elixir module and its initializer as a mix task that can be invoked directly as a build tool subcommand. The task is then responsible for spawning the oracle and providing its process ID to the model.

Finally, some significant changes were made to make the tool more suitable for the decentralized systems' environment. Chapter 5 gives a description of these changes and their motivation.

5 RUNNING AND TESTING DECENTRALIZED ALGORITHMS

Obtaining executable correspondence of declarative formulas is not trivial. Considering distributed systems the target environment, two concepts need to be present in the execution context: non-determinism and decentralization. Non-determinism is addressed by the oracle, which establishes a straightforward interface around pieces of code where non-determinism can occur. However, the oracle alone is insufficient to make the generated code decentralized.

In a centralized distributed system, with one central node and multiple distributed parts, the generated code runs actions from the central part of the model, while distributed nodes can be interfaces communicating through the oracle. For example, in an IoT (Internet of Things) scenario, a central controller can run the model and be responsible for making decisions based on values read from many sensors. In contrast, the sensors have no knowledge of the whole model and just send messages to the oracle with values as they read them.

However, where does the model run when the specified system is entirely decentralized? There is no specific node that is aware of every action. If the specification defines a producer and a consumer, for example, the executable correspondent of it should be divided into two parts: one responsible for the producer's actions and the other for the consumer's. This division, however, cannot be derived from the specification alone, as TLA$^+$ only defines actions and not their actors. Therefore, the generation tool provides a configuration interface that allows users to define groups of actions corresponding to different nodes in a decentralized system.

Centralized models can also describe an abstracted centralization that is not actually centralized in the existing system. Abstractions like these are necessary to reduce the scope of a model and maintain it within a feasible state space. For example, when modeling some application that runs on top of a blockchain, the inner works of the blockchain do not need to be modeled in the exact specification. Supposing another model already defines a blockchain and its properties, the application specification can model the blockchain as a centralized object with those properties. In this case, the code generated by the original tool would be fit to simulate a blockchain, where the application would send its commands to the oracle to be computed on this centralized node representing an abstracted blockchain - which is not very useful beyond simple simulation. The application code following the specified protocols and restrictions is a most useful executable artifact for this specification. The programmer can unplug this from the abstracted blockchain and point it to an actual blockchain. With the implemented improvements to the tool, the user can specify in the configuration file what actions are part of the application, and each one is part of the blockchain, so the generated code for the application can be run against a real blockchain (given the connection code is inserted after generation).

5.1 CONFIGURATION FILE

With the goal of automating as much of the code and test generation as possible, and considering that there is no way of inferring the process distribution from the model alone, the

new version of the tool requires a configuration file that describes how actions are distributed between processes and which variables should be shared between those processes. Once the tool requires a configuration file, it can also be used for metadata previously defined as command line arguments or required manual updates of the generated files (as was the case for constant values).

In the updated tool implementation, the configuration file is a JSON formatted file, and it is the only parameter for code, white-box test, and black-box test generation commands, holding the value for the following parameters:

- Processes: a list of objects with the process ID and the actions (defined as TLA$^+$ strings) it is responsible for;
- Shared variables: a list of strings with the shared variables' names
- Constants: a list of objects with name and value (defined as a TLA$^+$ string) for each constant;
- Init: the name of the initial state definition in the specification;
- Next: the name of the next state definition in the specification;
- Input format: either JSON or TLA;
- Input file: the JSON or TLA file name, accordingly to the input format;
- State graph: the JSON file name with the formatted state graph, used for white-box tests;
- Black-box tests: a list of objects with the test name and its trace file name;

5.2 CODE GENERATION

The oracle's role in a decentralized scenario is more complex. It acts as an abstraction for inter-process communication by storing some user-defined shared variables and providing reads and updates, controlled by locks, to nodes in order for them to execute their actions. Each node starts independently since it can be run at any time and place. Then, it tries to obtain the lock from the oracle and read the current state by combining its private variables with the shared ones obtained from the oracle. It then either performs an action or stays at the same state, informing the oracle about the resulting state, so it can update the shared variables' values and release the lock. If a node's actions have non-determinism between them, the node can have its private oracle to handle this non-determinism without having access to any shared information at all. Algorithm 2 describes how the decentralized processes communicate with the oracle.

As for the oracle implementation, it maintains the variables value for the current state, the lock-related variables, and can have other variables related to its specific implementation

Algorithm 2 Algorithm for a processes' main function

```
1:  procedure MAIN(oracle)
2:      variables ← wait lock from oracle
3:      possible_actions ← next(variables)
4:      if size of possible_actions is 1 then                    ▷ Deterministic
5:          notify oracle of transition to the single action in possible_actions
6:      else                                                     ▷ Non-deterministic
7:          send a choice request with possible_actions to the oracle
8:          response ← oracle response
9:          if response is cancel then
10:             new_variables ← variables
11:         else
12:             new_variables ← response
13:         notify oracle of transition to new_variables
14:     return main(oracle)
```

(i.e., the `TraceCheckerOracle` maintains a trace and the step index). Other than that, it works by reacting to incoming messages as follows:

- Upon receiving a lock request, it checks its lock-related variables and responds with either
  - Already lock; or
  - Lock acquired + shared variable's value, updating the lock-related variables and starting to monitor the process which obtained the oracle;
- Upon receiving a choice request along with a list of actions to choose from, it performs its choice implementation and responds with the chosen action (i.e., the `RandomOracle` will choose a random action from the provided list)
- Upon receiving a notification with the new values for the variables, it updates its internal value for the shared variables. Notifications are not responded.
- Upon receiving a termination message from the monitored process, that is, the process that holds the lock frees it. Therefore, this message is not responded.

5.3 WHITE-BOX TEST GENERATION

Testing this decentralized generated code is also more complex. For white-box tests, instead of matching transitions from a state to a call of `next()` from that state, the test actually matches the transitions to a combination of `next()` calls from all the generated nodes. The result is one white-box test file per specification testing all the generated files. These tests also act as a way to ensure that the user-defined distribution of actions per node is complete since if any

action is unassigned, its transitions will be missing, and the test will fail. This testing process is illustrated in Figure 5.

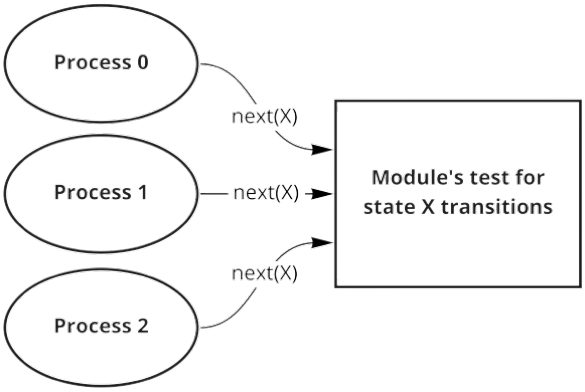

Figure 5 – White-box tests for decentralized processes

5.4 BLACK-BOX TEST GENERATION

Black-box tests run with an oracle implementation, so a single oracle is defined to work both as the node's private oracle and the communication oracle. This oracle is more active than other ones, which just wait for decision or lock requests, as it needs to start all nodes and control the execution order to follow the tested trace's path. It releases the lock to one node at a time and waits for the resulting state: if it matches the next step on the trace, the transition is accepted, and shared variables are updated; otherwise, it responds to the node with a rejection, and queries the next one by releasing the lock for it. If the oracle is queried for a non-determinism resolution, it behaves the same (see Algorithm 1) by choosing only transactions that result in the desired state when present. If in a same step of the trace, none of the nodes can perform a transition that results in the next state of the trace, the test fails.

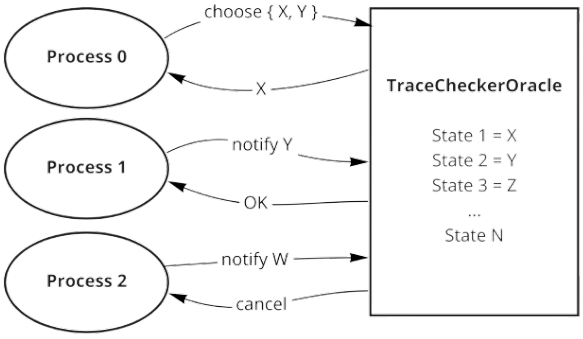

Figure 6 – Example of interactions with the oracle in black-box testing

A black-box test representation is given at Figure 6. First, process 0 needs help from the oracle to solve the non-determinism between X and Y, and the oracle looks at the trace it is checking and the current step counter, responding with X since it is the first state. Later, process

1 performs a transition to state Y and notifies the oracle of it. As the transition matches the trace's next state, the oracle accepts it and responds with OK. Then, process 2 notifies a transition to W, which is not expected and therefore refused with a "cancel" response - which does not fail the test until all other processes also have their actions canceled.

6 CASE STUDIES

This section describes two case studies where decentralized code and tests are generated from specifications of a termination detection algorithm and a blockchain protocol.

6.1 CASE STUDY: DIJKSTRA'S TERMINATION DETECTION

Dijkstra's algorithm for termination detection in distributed computations, referenced as EWD840 (DIJKSTRA; FEIJEN; van Gasteren, 1983), is explored as a case study for the code and test generation tool. In this algorithm, nodes are disposed of in a ring, and the first node can initiate probes by sending messages to the last node to discover whether there are active nodes in the ring. The message is a token that can be black or white. Each node has its color, which can also be black or white. A node can be active or passive, and only active nodes can wake up other passive nodes - therefore, once all nodes are passive, they will stay passive forever, and this is the termination that should be detected by a probe initiated by the first node. This information is encoded in four TLA$^+$ variables:

VARIABLES $active$, $color$, $tpos$, $tcolor$

$Nodes \triangleq \{0, 1, 2\}$
$Init \triangleq$
$\quad \wedge\, active = [n \in Nodes \mapsto \text{TRUE}]$
$\quad \wedge\, color = [n \in Nodes \mapsto \text{“white”}]$
$\quad \wedge\, tpos = 0$
$\quad \wedge\, tcolor = \text{“black”}$

$active$ and $color$ are functions that map the active/passive status and color of each node. $tpos$ is the token position and, $tcolor$, its color. According to the algorithm specification, the only requirement at the initial state is that the token color is black, and all other values in this $Init$ definition are assigned arbitrary type-correct values.

In a ring with N nodes, node 0 sends a black token to node $N-1$, which then propagate the token through the ring until it reaches node 0 again. The token changes its color to black or white depending on the emitting node's color. If a white token reaches node 0, the node's color is also white and not active, then termination is detected for all nodes in the ring. As follows, the $InitiateProbe$ action defines the probe start, which can only happen when the token is in node 0, and either black or the node color is black. Then, the node and the token become black and the token position switches to $N-1$. Formula $terminationDetected$ defines conditions for a state to be considered a termination detection.

$InitiateProbe \triangleq$
$\quad \wedge\, tpos = 0$
$\quad \wedge\, tcolor = \text{“black”} \vee color[0] = \text{“black”}$

$\land\, tpos' = N - 1$
$\land\, tcolor' = \text{“white”}$
$\land\, active' = active$
$\land\, color' = [color \text{ EXCEPT } ![0] = \text{“white”}]$

$terminationDetected \triangleq$
$\land\, tpos = 0 \land tcolor = \text{“white”}$
$\land\, color[0] = \text{“white”} \land \neg active[0]$

When the token is in any other node i, it can be passed to node $i-1$ as defined by action $PassToken(i)$. If it is not active, the token is white, and its color is white, it propagates the token. If it is active, the token is white, and its color is white, it will not propagate any token as this action will not be enabled. Otherwise, it sends a black token. This action also paints the node white. The fact that this action is not applied for node 0 is ensured by the $Next$ action, which is defined later.

$PassToken(i) \triangleq$
$\land\, tpos = i$
$\land\, \neg active[i] \lor color[i] = \text{“black”} \lor tcolor = \text{“black”}$
$\land\, tpos' = i - 1$
$\land\, tcolor' = \text{IF } color[i] = \text{“black” THEN “black” ELSE } tcolor$
$\land\, active' = active$
$\land\, color' = [color \text{ EXCEPT } ![i] = \text{“white”}]$

Active nodes may also wake up passive nodes at any time. This action is described by $SendMsg(i, j)$, where an active node i wakes up a possibly passive node j. If the activation goes against the probe order in the ring, that is, ij, then i 's color turns black.

$SendMsg(i, j) \triangleq$
$\land\, active[i]$
$\land\, active' = [active \text{ EXCEPT } ![j] = \text{TRUE}]$
$\land\, color' = [color \text{ EXCEPT } ![i] = \text{IF } j > i \text{ THEN “black” ELSE } @]$

Lastly, any active node i can become passive at any given time, which is represented by action $Deactivate(i)$.

$Deactivate(i) \triangleq$
$\land\, active[i]$
$\land\, active' = [active \text{ EXCEPT } ![i] = \text{FALSE}]$
$\land\, \text{UNCHANGED } \langle color, tpos, tcolor \rangle$

The next state formula $Next$ combines all these actions to form a valid transition system for this algorithm: all nodes except 0 can propagate the token, and all nodes (with no exception) can send activation messages to nodes different then themselves or deactivate themselves.

$$
\begin{aligned}
Next \triangleq\ & \lor InitiateProbe \\
& \lor \exists\, i \in Nodes \setminus \{0\} : PassToken(i) \\
& \lor \exists\, i \in Nodes : \lor \exists\, j \in Nodes \setminus \{i\} : SendMsg(i, j) \\
& \qquad \lor Deactivate(i)
\end{aligned}
$$

The main safety property $TerminationDetection$ states that if the termination was detected in a state (as defined with $terminationDetected$ at the beginning of this chapter), then all nodes should not be active in that state. As for liveness, a temporal property is defined, stating that if all nodes are passive in a state, termination has to be eventually detected in a future state. That is, all nodes being passive leads to termination detection.

$$
\begin{aligned}
& TerminationDetection \triangleq \\
& \quad terminationDetected \implies \forall\, i \in Nodes : \neg active[i]
\end{aligned}
$$

$$
\begin{aligned}
& Liveness \triangleq \\
& \quad (\forall\, i \in Nodes : \neg active[i]) \leadsto terminationDetected
\end{aligned}
$$

This is a decentralized system, so the generator tool should be configured to generate different files for each node. In the TLA$^+$ specification, actions for different nodes are parameterized by the node's index, and the distribution configuration can use this parametrization: Node 0 is responsible for actions referring to initiating the probe, waking up other nodes, or deactivating itself; Other nodes cannot initiate a probe but can pass the token to the preceding node, besides waking up other nodes and self-deactivating.

6.1.1 Generated code

Generating code for the specification of this algorithm instanced with 3 nodes results in the elixir modules:

1. A base module containing all the translated definitions and some shared auxiliary functions, but no `next()`. This is an internal module that should not be used directly.

2. Three modules, one per node, containing the `next()` function for a node. This function calls other functions defined in the base module and should result in all possible transitions on a given state considering the actions under the node's responsibility. That is, node 0 `next()` should list transitions for initiating probes while others shouldn't.

3. One task to start and monitor the oracle - a random oracle is used for quickest simulation - which starts with the translated initial state.

4. Three tasks to start each one of the nodes, also with the translated initial state and with the oracle's PID (Process Identifier) that is found by name from the BEAM's global registry. These tasks are generated separately because, in a realistic scenario, different nodes can be started at different times and locations.

The four different tasks (one for the oracle and one for each of the three nodes) can be run from different machines as long as they are on the same network. The oracle's address (BEAM's `sname`) is provided as a command-line parameter to the nodes' tasks so it can be found on the network. With the oracle in place being the random oracle, once all tasks are started, a random simulation occurs with messages being exchanged between the oracle and the nodes and transitions being computed.

6.1.2 Generated tests

The white-box testing strategy generates 110 unit tests for this specification with 3 nodes, accounting for all 110 different states and 467 transitions. As an experiment, node 1's code is updated to perform no transition at all, and that causes a total of 68 tests to fail with missing transitions. These guarantees full state coverage and full transition coverage for the model. One example of a state with only one transition is presented below.

```
test "fromState 107" do
  variables = %{
    tpos: 1,
    active: %{0 => false, 1 => false, 2 => false},
    tcolor: "white",
    color: %{0 => "black", 1 => "black", 2 => "white"}
  }

  expectedStates = [
    %{
      tpos: 0,
      active: %{0 => false, 1 => false, 2 => false},
      tcolor: "black",
      color: %{0 => "black", 1 => "white", 2 => "white"}
    }
  ]

  actions =
    List.flatten([
      APAEWD840_node0.next(variables),
      APAEWD840_node1.next(variables),
      APAEWD840_node2.next(variables)
    ])

  states = Enum.map(actions, fn action -> action[:state] end)
```

```
  assert Enum.sort(Enum.uniq(states)) == Enum.sort(Enum.uniq(expectedStates))
end
```

As for the black-box tests, the most interesting witness to be tested in a termination detection algorithm is probably one that detects a termination. The property for termination detection is the following.

$$
\begin{aligned}
&terminationDetected \triangleq \\
&\quad \land tpos = 0 \land tcolor = \text{“white”} \\
&\quad \land color[0] = \text{“white”} \land \neg active[0]
\end{aligned}
$$

$$
\begin{aligned}
&TerminationDetection \triangleq \\
&\quad terminationDetected \implies \forall\, i \in Nodes : \neg active[i]
\end{aligned}
$$

Negating the whole property results in a violation in the initial state since the left side of the implication ($terminationDetected$) is not satisfied by it, making the implication true and, its negation, false. Therefore, negating only the right side of the implication is more useful, resulting in a formula that asserts that if termination is detected, there is at least one active node. Running the model checker with this formula will result in a 7-state counterexample where termination is detected, and there are no active nodes, which is a witness to termination that can be used to test the code.

```
State 1: <Initial predicate>
/\ tpos = 0
/\ active = (0 :> TRUE @@ 1 :> TRUE @@ 2 :> TRUE)
/\ tcolor = "black"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")

State 2: <InitiateProbe>
/\ tpos = 2
/\ active = (0 :> TRUE @@ 1 :> TRUE @@ 2 :> TRUE)
/\ tcolor = "white"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")

State 3: <Deactivate>
/\ tpos = 2
/\ active = (0 :> FALSE @@ 1 :> TRUE @@ 2 :> TRUE)
/\ tcolor = "white"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")
```

```
State 4: <Deactivate>
/\ tpos = 2
/\ active = (0 :> FALSE @@ 1 :> FALSE @@ 2 :> TRUE)
/\ tcolor = "white"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")

State 5: <Deactivate>
/\ tpos = 2
/\ active = (0 :> FALSE @@ 1 :> FALSE @@ 2 :> FALSE)
/\ tcolor = "white"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")

State 6: <PassToken>
/\ tpos = 1
/\ active = (0 :> FALSE @@ 1 :> FALSE @@ 2 :> FALSE)
/\ tcolor = "white"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")

State 7: <PassToken>
/\ tpos = 0
/\ active = (0 :> FALSE @@ 1 :> FALSE @@ 2 :> FALSE)
/\ tcolor = "white"
/\ color = (0 :> "white" @@ 1 :> "white" @@ 2 :> "white")
```

The test task starts a `TraceCheckerOracle` with this witness as trace and the three nodes. The oracle restricts the nodes' transitions and choices, always respecting the oracle's interface to follow the desired trace. After all 7 states from the trace have been reached, the oracle returns success, and all processes are killed. The termination task, named after the trace name provided by the user, is presented below and can be run with the build tool by executing `mix termination`.

```
defmodule Mix.Tasks.Termination do
  use Mix.Task

  @impl Mix.Task
  def run(_) do
    trace = [
      # The translated trace, omitted for simplicity
    ]
```

```
    modules = [
      APAEWD840_node0,
      APAEWD840_node1,
      APAEWD840_node2
    ]

    pids = Enum.map(
      modules,
      fn m -> spawn(m, :main, [self(), Enum.at(trace, 0), 0]) end
    )
    TraceCheckerOracle.start(trace, true, 0, nil, pids)
  end
end
```

6.2 CASE STUDY: ATTACK VECTOR ON A BLOCKCHAIN API

This second case study considers a production application in the blockchain environment. A document published in (VLADIMIROV, 2016) refers to an API definition for token transfer in the Ethereum blockchain and exposes a vulnerability in the protocol. This faulty API version is later modeled with a TLA$^+$ specification, and the attack trace can be found by model checking a specification property.

This section will not discuss details about the protocol and the attack as this model is significantly more complex than previous examples. Instead, this case study aims to demonstrate how the proposed tool works and can be useful in a real production implementation.

The model for this API considers only three methods, and even so, the number of states escalates too fast with the number of steps. TLC, which tries to enumerate all states at each step, takes more than a few minutes to compute traces of length 7 and beyond, where about 40 million distinct states are found in runs up to that length. This model defined an infinite number of states since there is a transaction counter that monotonically increases as transitions are made. Therefore, generating white-box tests for the entire state space is not possible. It should be possible to obtain partial white-box tests from a partial state graph, but this work did not explore strategies like this.

Instead, a black-box test is generated from Apalache's trace, representing the attack vector. Apalache's strategy for model checking performs better than TLC in finding a counterexample for this model, resulting in a 9-step trace (that is, a sequence of ten states).

Code generation for this API is configured to output different files for agents and the abstracted blockchain, with agents responsible for running client-side requests and the abstracted blockchain committing the transactions. From the counterexample found by Apalache, a black-box test is generated. By running the test, processes for three different agents are started, along

with the blockchain-representing process, and the `TraceCheckerOracle` containing the 10-state sequence configures the attack. The oracle can manipulate the agents into running the attack sequence, which the blockchain process accepts as the test succeeds and the trace is successfully reproduced. The ability to reproduce an attack or bug trace like this reduces the gap between specification artifacts and production systems by allowing the agent code to take simple modifications and run the methods in a production blockchain, demonstrating that the found attack or bug is reproducible in production. Besides that, the ability to navigate through a simulation of a trace in an executable code representing the entire model enables a better understanding of the faulty behavior and the ability to explore other possibilities as trying to change the code to prevent that specific sequence of steps to happen. It is reasonable to conjecture that the playground available in this generated scenario allows more exploration than existing tooling around TLA$^+$ model checkers.

Lastly, if the model is updated to represent a new version of the API that fixes this vulnerability, this test would break the code generated from that new model. The ideal lifecycle of this test would be to update the expected result from successful reproduction to failure, turning it into a regression test, stating that the described faulty behavior should not be executable in new versions of the implementation from now on.

7 CONCLUSION

Although TLA$^+$ specifications are purely declarative, they are meant to be used to describe algorithmic behavior, which should be simulatable, executable, and testable. The main challenge is to define clear interfaces for the high-level abstractions that TLA$^+$ enables. This work transforms deep study in understanding TLA$^+$ concepts and how they work in an executable context into a tool that translates all of these concepts (conditions, transitions, non-determinism, decentralization, state graphs, and counterexamples) into artifacts more familiar to engineers: functions, tasks, inter-process communication, unit and integration tests.

The result is a full development environment generated from a single specification that has the potential to help with learning TLA$^+$, by working as a playground for people writing their first models, and mainly by reducing the gap between specification and implementation, by providing simple interfaces to connect generated code from a specification with real-world systems, validating integrations and reproducing bugs (and their fixes, as regression tests).

The ability to run the specified model contributes to the process of both learning and debugging TLA$^+$ specifications. Pursuing this goal, TLC included a REPL (Read Eval Print Loop) capable of executing TLA$^+$ expressions on a console. Generating code and tests is a relevant additional step towards this objective, enabling the user not only to run the model, but also to visualize how it can be represented in a real programming language, how the states behave (with white-box tests), and easily manipulate the execution to follow a specific math either automatically (with black-box tests and the `TraceCheckerOracle`) or manually defining each step (by using an I/O oracle).

Besides qualities that improve the developers' workflow in learning and debugging models, which is a way of reducing the cost of modeling software, the tool has aspects that are relevant to industry usage, which can be able to incentivize TLA$^+$ usage in the industry by significantly increasing the benefits of modeling software. These benefits include a low-cost tested prototype obtained directly from the model and automated reproduction of bugs (either found by model checking or created manually).

The fact that generated tests are testing generated code takes the test harness burden off. Again, clear interfaces make for easy testing, where test files are self-containing (no external data needs to be read), a single function is called, and the results are matched, resulting in high readability and low indirection. This is an environmental advantage when compared to generic model-based testing.

Future development of this work should focus on black-box testing and overall orchestration, as these can be useful features in multiple contexts. Black-box testing infrastructure can bootstrap the system to a certain point and open it up for user-guided simulation. That would be a great debugging tool as this is not possible with any model checker as of this writing. There are also improvements to be made to the current implementation, including mainly user experience-related features and user documentation which were not prioritized in the first validation-focused

implementation.

Although TLA$^+$ is untyped, Apalache implements a type system for it with comment-based type annotations. This work's tool implements untyped code generation from untyped TLA$^+$, which evidently has no correctness benefits from a type system. Considering the possibility of typed TLA$^+$ specifications, as any specification that can be model checked by Apalache, it would be possible and beneficial to generate typed code and tests. As the current implementation is not deeply coupled with Elixir, the code could be generated in another statically typed programming language. The main necessary change would be to include types in parsing and the tool's internal representation.

Lastly, a relevant consideration is that the tool was developed for validation under some experiment scenarios and has much to be improved in both the scope of specification constructs accepted as input and the quality of output code. Realistically, qualifying it for industrial usage would require a team of people to develop and maintain it. Nonetheless, the initially proposed ideas for this work could be transformed into concrete implementation with few adjustments, and adaptions to the project have gotten easier to make at each iteration, which evidenced a well-designed project with potential for extension.

REFERENCES

AMMANN, P; BLACK, Paul; MAJURSKI, William. Using model checking to generate tests from specifications. In: . [S.l.]: IEEE International Conference on Formal Engineering Methods, Brisbane, AS, 1998. Cited at page 13.

ARCAINI, Paolo; GARGANTINI, Angelo; RICCOBENE, Elvinia. Improving model-based test generation by model decomposition. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 119–130. ISBN 9781450336758. Disponível em: <https://doi.org.ez74.periodicos.capes.gov.br/10.1145/2786805.2786837>. Cited 2 times at pages 17 and 19.

ARMSTRONG, Joe L; VIRDING, RH; WILLIAMS, Mike C. Erlang user's guide and reference manual version 3.2. **Ellemtel Utvecklings AB, Sweden**, 1991. Cited at page 21.

BAIER, Christel; KATOEN, Joost-Pieter. **Principles of Model Checking**. [S.l.: s.n.], 2008. v. 26202649. ISBN 978-0-262-02649-9. Cited 2 times at pages 12 and 13.

COMMUNITY, Ruby. **About Ruby**. 2021. Disponível em: <https://www.ruby-lang.org/en/about/>. Acesso em: May 15, 2021. Cited at page 21.

DIJKSTRA, Edsger W.; FEIJEN, W.H.J.; van Gasteren, A.J.M. Derivation of a termination detection algorithm for distributed computations. In: **Information Processing Letters**. [S.l.: s.n.], 1983. v. 14, p. 217–219. Cited at page 38.

DORMINEY, Star. **Kayfabe: Model-Based Program Testing with TLA+/TLC**. 2020. Disponível em: <https://conf.tlapl.us/2020/11-Star_Dorminey-Kayfabe_Model_based_program_testing_with_TLC.pdf>. Acesso em: 02 jul. 2021. Cited 3 times at pages 9, 19, and 27.

Elixir Core Team. **Elixir**. Disponível em: <https://elixir-lang.org/>. Cited 2 times at pages 9 and 21.

ENOIU, Eduard P et al. Automated test generation using model checking: an industrial evaluation. **International Journal on Software Tools for Technology Transfer**, Springer, v. 18, n. 3, p. 335–353, 2016. Cited at page 18.

HONG, HS et al. A temporal logic based theory of test coverage and generation. In: Katoen, JP and Stevens, P (Ed.). **TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANAYLSIS OF SYSTEMS, PROCEEDINGS**. HEIDELBERGER PLATZ 3, D-14197 BERLIN, GERMANY: SPRINGER-VERLAG BERLIN, 2002. (Lecture Notes in Computer Science, 2280), p. 327–341. ISBN 3-540-43419-4. ISSN 0302-9743. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002), GRENOBLE, FRANCE, APR 08-12, 2002. Cited at page 18.

KONNOV, Igor; KUKOVEC, Jure; TRAN, Thanh-Hai. Tla+ model checking made symbolic. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 3, n. OOPSLA, out. 2019. Disponível em: <https://doi.org/10.1145/3360549>. Cited 3 times at pages 9, 14, and 31.

KOO, Heon-Mo; MISHRA, Prabhat. Functional test generation using design and property decomposition techniques. **ACM Trans. Embed. Comput. Syst.**, Association for Computing

Machinery, New York, NY, USA, v. 8, n. 4, jul. 2009. ISSN 1539-9087. Disponível em: <https://doi.org.ez74.periodicos.capes.gov.br/10.1145/1550987.1550995>. Cited at page 19.

KUPRIANOV, Igor Konnov Andrey. **Model-based testing with TLA+ and Apalache**. 2020. Disponível em: <http://conf.tlapl.us/2020/09-Kuprianov_and_Konnov-Model-based_testing_with_TLA_+_and_Apalache.pdf>. Acesso em: 02 jul. 2021. Cited 2 times at pages 9 and 20.

LAMPORT, Leslie. The temporal logic of actions. **ACM Trans. Program. Lang. Syst.**, v. 16, n. 3, p. 872–923, 1994. Cited at page 12.

LAMPORT, Leslie. **Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers**. Addison-Wesley, 2002. Disponível em: <https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/>. Cited 5 times at pages 9, 14, 15, 16, and 32.

LAMPORT, Leslie. The specification language tla+. In: HENSON, Dines Bjørner e Martin C. (Ed.). **Logics of specification languages**. Berlin: Springer, 2008. p. 616–620. ISBN 3540741062. Disponível em: <http://lamport.azurewebsites.net/pubs/commentary-web.pdf>. Cited at page 9.

MAFRA, Gabriela Moreira. **Tradução automática de especificação formal modelada em TLA+ para linguagem de programação**. 69 p. Monografia (Trabalho de Conclusão de Curso) — Universidade do Estado de Santa Catarina, Joinville, 2019. Cited 3 times at pages 9, 11, and 21.

MERZ, Stephan. On the logic of tla+. **Computers and Artificial Intelligence**, v. 22, p. 351–379, 01 2003. Cited at page 12.

MILNER, Robin; PARROW, Joachim; WALKER, David. A calculus of mobile processes, i. **Information and Computation**, v. 100, n. 1, p. 1 – 40, 1992. ISSN 0890-5401. Disponível em: <http://www.sciencedirect.com/science/article/pii/0890540192900084>. Cited at page 9.

MOHALIK, Swarup et al. Software Testing, Verification and Reliability. **SOFTWARE TESTING VERIFICATION & RELIABILITY**, WILEY, 111 RIVER ST, HOBOKEN 07030-5774, NJ USA, 24, n. 2, p. 155–180, MAR 2014. ISSN 0960-0833. Cited at page 18.

NEWCOMBE, Chris et al. How amazon web services uses formal methods. **Commun. ACM**, ACM, New York, NY, USA, v. 58, n. 4, p. 66–73, mar. 2015. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2699417>. Cited at page 9.

PETRI, C. A. Fundamentals of a theory of asynchronous information flow. In: **IFIP Congress**. [S.l.: s.n.], 1962. p. 386–390. Cited at page 9.

ROBINSON-MALLETT, Christopher et al. Extended state identification and verification using a model checker. **INFORMATION AND SOFTWARE TECHNOLOGY**, ELSEVIER SCIENCE BV, PO BOX 211, 1000 AE AMSTERDAM, NETHERLANDS, 48, n. 10, p. 981–992, OCT 2006. ISSN 0950-5849. Workshop on Advances in Model-Based Software Testing, St.Louis, MO, MAY 15-16, 2005. Cited at page 19.

TEAM, The Elixir. **ExUnit - Unit testing framework for Elixir**. 2021. Disponível em: <https://hexdocs.pm/ex_unit/ExUnit.html>. Acesso em: 02 jul. 2021. Cited at page 25.

VISSER, Willem; PĂSĂREANU, Corina S.; KHURSHID, Sarfraz. Test input generation with java pathfinder. In: **Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2004. (ISSTA '04), p. 97–107. ISBN 1581138202. Disponível em: <https://doi.org.ez74.periodicos.capes.gov.br/10.1145/1007512.1007526>. Cited at page 18.

VLADIMIROV, Dmitry Khovratovich Mikhail. **ERC20 API: An Attack Vector on the Approve/TransferFrom Methods**. 2016. Disponível em: <https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/>. Acesso em: 08 aug. 2022. Cited at page 44.

XIA, Songtao; VITO, Ben Di; nOZ, César Mu*A*utomated test generation for engineering applications. In: **Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2005. (ASE '05), p. 283–286. ISBN 1581139934. Disponível em: <https://doi.org.ez74.periodicos.capes.gov.br/10.1145/1101908.1101951>. Cited at page 18.