Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Έλεγχος προγραμμάτων συνδυάζοντας συμβολική και συμπαγή εκτέλεση με αυτόματη παραγωγή τιμών εισόδου

Διπλωματική Εργασία

του

Άγγελου Γιάντσιου

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιανουάριος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Έλεγχος προγραμμάτων συνδυάζοντας συμβολική και συμπαγή εκτέλεση με αυτόματη παραγωγή τιμών εισόδου

Διπλωματική Εργασία

του

Άγγελου Γιάντσιου

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Ιανουαρίου, 2014.

.....	.....	.....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2014

.....
Άγγελος Γιάντσιος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σε ένα κόσμο όπου το λογισμικό υπάρχει παντού, από διαστημικά λεωφορεία μέχρι αντλίες ινσουλίνης, η δοκιμή λογισμικού έχει προεξέχουσα θέση. Παρόλαυτα, η δοκιμή λογισμικού γίνεται ακόμα σε μεγάλο βαθμό χειροκίνητα καθώς είναι αρκετά δύσκολο να παραχθούν σουίτες δοκιμών υψηλής κάλυψης κώδικα με αυτοματοποιημένα εργαλεία που χρησιμοποιούν την προσέγγιση "μαύρο κουτί".

Σε αυτή τη διπλωματική, παρουσιάζουμε το CutEr, ένα εργαλείο δοκιμών για τη γλώσσα Erlang που υλοποιεί μια δυναμική τεχνική της προσέγγισης "λευκό κουτί" που ονομάζεται συμπαγο-συμβολική δοκιμή. Αυτή η τεχνική συνδυάζει συμπαγή και συμβολική εκτέλεση ενός προγράμματος έτσι ώστε να δημιουργούμε τιμές εισόδου που θα εξερευνούν διαφορετικά μονοπάτια εκτέλεσης του.

Λέξεις Κλειδιά

concolic testing, Erlang, software testing, dynamic symbolic execution, SMT solving

Abstract

In a world where software is everywhere, from space shuttles to insulin pumps, software testing has a prominent role. However, software testing is largely still manual since it is very difficult to create high code coverage test suites with automated black-box tools.

In this thesis, we present CutEr, a testing tool for Erlang that implements a dynamic white-box technique called concolic testing. This technique combines concrete and symbolic execution of a program in order to generate inputs that will explore its different execution paths.

Keywords

concolic testing, Erlang, software testing, dynamic symbolic execution, SMT solving

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον Κωστή Σαγώνα για την καθοδήγησή του, την υπομονή του και την πίστη του σε εμένα. Σε μια περίεργη περίοδο της ακαδημαϊκής μου πορείας, η εμπιστοσύνη που μου έδειξε αποτέλεσε το σημαντικότερο κίνητρό μου.

Επίσης θα ήθελα να ευχαριστήσω τον Νίκο Παπασπύρου, καταρχάς για την πολύτιμη βοήθεια του, αλλά πολύ περισσότερο για το ότι αποτελεί τον βασικό λόγο για τον οποίο ασχολήθηκα με την Πληροφορική. Η διδασκαλία του και το ήθος του ήταν και συνεχίζουν να είναι έμπνευση για εμένα.

Ευχαριστώ την οικογένειά μου που με έκανε τον άνθρωπο που είμαι σήμερα.

Τέλος, ευχαριστώ την Μαιρούλα μου για την στήριξη της όλα αυτά τα χρόνια. Σε ευχαριστώ που είσαι εσύ και που με κάνεις καλύτερο!

Άγγελος Γιάντσιος

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In a world where software is everywhere from space shuttles to insulin pumps, software testing has a prominent role. Unreliable software can potentially cause human loss and incur huge monetary costs for the industries. Therefore, companies spend a large amount of their development budget to ensure the quality and reliability their software.

Many software testing techniques have been developed and used over the years. A notion that has become increasingly popular in the research community is automated software testing. Techniques that provide automatic static or dynamic analysis on a given code base have garnered a lot of attention and tools that implement them are being used daily both in academia and the real-world software industry.

However, software testing is largely still manual since an effective test suite must consist of inputs that exercise the program in different ways and achieve high code coverage. This a very difficult goal to achieve with automated black-box tools (which comprise the majority of automated software testing tools), thus test engineers resort to manual unit testing which is very expensive and time consuming. In addition, bugs may occur at inputs that may take place in very rare but plausible conditions which test engineers may not foresee.

As part of this thesis, we have explored ways of implementing a dynamic white-box technique for the Erlang language that generates inputs for the program using the program's code. This technique is called concolic testing and combines concrete and symbolic execution of a program. The key idea is that given a program and a random input, we record how the input affects the control flow of the execution and create a conjunctive logical formula that if an input satisfies we know that the execution will go along the same path. Then, we use a constraint solver to negate a term of the formula and generate a new input for our program that we expect to drive the execution along a different path. We then iteratively do the same process and hopefully enumerate all the execution paths of our program.

This technique has already been implemented for imperative and object-oriented languages such as C and Java. All these tools use a low level representation of the program's code, like Assembly or LLVM, thus limiting the expressiveness of the constraint solver. We decided that we wanted to investigate the effectiveness of using a high level representation of Erlang for our tool and enhance the expressiveness of the language used to interface with the constraint solver. Therefore, we decided to create the tool from scratch and not

use an existing symbolic execution engine, like KLEE [3], and we named it CutEr. The constraint solver we used is Z3 from Microsoft Reasearch.

Moreover, the current implementation of CutEr supports sequential programs and a subset of the Erlang language's type system.

Aim of the Thesis

The goal of this thesis is to research the capabilities and prospects of applying the concolic white-box testing technique in Erlang, a dynamically typed functional language.

Will the solver be able to reason over lists, tuples and user defined types? Will the concolic technique be successfully adapted to the concurrency model of Erlang? Will it be feasible to implement such a tool from scratch or do we need to build on an existent concolic engine? How helpful can the tool be to an Erlang programmer? All the above questions and more came to our minds when we chose this subject and these are the questions that we will try to answer in this thesis.

In Chapter 2, we will present all the necessary information regarding Erlang so as to fully comprehend the architecture and workflow of CutEr. First of all, we will discuss what is Erlang and how it is different from other programming languages. As we will be working on a lower level representation of Erlang programs, we will also discuss the basics of the Erlang Compiler and present Core Erlang, the language used for the Erlang intermediate code. Reasoning over symbolic constraints and generating new program inputs is heavily dependent on the language's type system thus we will highlight they key points of the Erlang type system and how it is integrated into Core Erlang.

In Chapter 3, we will describe in detail the notion of concolic testing and discuss its strengths, its limitations and what are the main challenges of implementing an Erlang concolic tool. We will also see how important is the constraint solver and the reasons why we chose Z3 SMT Solver.

In Chapter 4, we will present our experiences in the development of CutEr and we will discuss all the details of its current architecture.

In Chapter 5, we will see how we can use CutEr to test three sequential Erlang programs and we will talk about the challenges that arise from each case.

In Chapter 6, we summarize our findings and present our future work.

What we Achieved

In this thesis, we managed to develop a tool that is able to test sequential Erlang programs using the technique of concolic testing. It can reason for a large subset of the Erlang types. However, the lack of support for bitstrings and binaries has a notable impact on the number of programs it can test.

We have used CutEr to test some simple Erlang programs and will present our findings. More importantly, we believe that we managed to pinpoint the actual problems in applying the idea of concolic testing efficiently and with success in Erlang.

Chapter 2

The Erlang/OTP System

2.1 The Erlang Language

Erlang is a programming language designed at the Ericsson Computer Science Laboratory in the 1980's. It was initially used for developing software for telecommunications systems. Erlang was released as open source in 1998 and is mainly being used in domains such as banking, e-commerce, computer telephony and instant messaging.

The designers of Erlang decided that they wanted a language that would be suitable for the development of massively scalable and robust distributed systems that would be fault-tolerant and would ensure high availability.

The scalability and robustness were achieved by having lightweight processes that conformed to the actor model. When an Erlang process is spawned, it uses a minimum amount of system resources and can request additional ones as needed. The actor model suggests that every process can only communicate with one another process through asynchronous message-passing. Every communication is explicit, traceable and safe. This approach greatly facilitates code debugging in concurrent applications.

Fault tolerance is a controversial topic since it has different meaning to people. For example, to some it may mean that no process ever fails and no data is ever lost. However, fault tolerance in Erlang is that the critical core of a system does not crash. Surely some processes may fail and their data may be lost but the system will continue to be running despite these events. If we couple this feature with on-the-fly code loading, we get a language that supports the development of always-on systems.

Erlang is a strict and dynamically-typed language. The programmer is not obliged to statically annotate his code in any way with type information. However, type safety is ensured by the runtime environment that catches the occurring type error, at which point an exception would be raised in the execution of the program. This allows for more flexible code and is essential for the fault-tolerant behaviour that Erlang provides.

A vital part of the Erlang implementation is the OTP framework. It is a set of Erlang libraries and design principles providing middle-ware to develop Erlang applications. It includes its own distributed database, applications to interface with other languages, debugging and release handling tools [8].

2.2 The Erlang Compiler

The code of an Erlang application is organized in modules which are independently compiled and loaded to the runtime system. The Ericsson Erlang implementation uses a register-based virtual machine which is called BEAM. By default, the compiler generates BEAM bytecode which the loader converts it to threaded code at load time.

The conversion of the textual representation of the Erlang code to BEAM bytecode is done by the compiler. The first step in this process is to parse the text file and create the Abstract Syntax Tree (AST) of the module. Erlang code is quite expressive and has a lot of syntactic sugar in it, therefore a major transformation occurs that translates the Erlang syntax tree to Core Erlang syntax tree. Core Erlang [5] is an intermediate high level language that we will present extensively in the next section. This representation is more concise with explicit scopes for variables.

In the next step, the compiler transforms the Core Erlang AST to its BEAM bytecode representation. This is a register-based representation. Then, the compiler runs some optimization passes on the bytecode and creates the file that will be used by the loader.

2.3 Core Erlang

As we mentioned before, Core Erlang is an intermediate representation of Erlang, which is used by the compiler as a middle-ware tier between the textual representation of the source code and the low level bytecode. It has clear and simple semantics that allow for a straight forward translation from Erlang code to Core Erlang code and from Core Erlang code to bytecode. It has a simple grammar thus simplifying the process of pretty printing Core Erlang programs for humans to read and edit. Its main goal is to facilitate the development of tools that operate on the Erlang source code, such as profilers, debuggers, source code optimizers and various programs that perform source code instrumentations.

The current implementation of the Erlang compiler allows the programmer to transform his program to the equivalent Core Erlang program and store its textual representation. He can then read it to inspect or modify the code transformation or even write some extra Core Erlang code by hand. However, should he want to execute his code, he must compile it to BEAM bytecode and then execute it as usual. There is no implementation of a Core Erlang interpreter, even for debugging purposes, as there is for the Abstract Format of the Erlang parse trees stored in the debugging information of a module. This feature was necessary for our implementation, thus a major part of the development cycle of our tool was dedicated to the creation of a reliable Core Erlang interpreter.

In the rest of this section, we will provide a brief description of the Core Erlang Abstract Syntax Tree as it is used in the Erlang compiler and its semantics.

2.3.1 The Core Erlang Abstract Syntax Tree

The Core Erlang Abstract Syntax Tree has a dual representation in the Erlang compiler. The core parts of the compiler use the records defined in the file `lib/compiler/-core_parse.hrl`, whereas `cerl_inline` and all the modules in `lib/hipe/cerl` use the Abstract Data Type defined in `lib/compiler/cerl.erl`. Essentially, both definitions

are the same and can be used interchangeably so the decision on which representation to use boils down to the personal preference and style of the programmer. The records' representation allows for easier decomposition of the syntax tree nodes by using pattern matching whereas the ADT representation offers a more formal approach that favours parse transformations. We decided to use the records' representation as pattern matching greatly facilitates our implementation of the Core Erlang interpreter.

We feel that since the lexical analysis and grammar of Core Erlang are presented in detail in the Core Erlang language specification paper [5], we should rather present the undocumented definitions of the Core Erlang Abstract Syntax Tree in `lib/compiler/src/core_parse.hrl` and discuss their semantics in the context of their current usage in the Erlang compiler. First of all, let's note some principles that will apply for the rest of the subsection and continue to present the AST records.

- A Core Erlang AST is considered to be of type Tree which is a union of all the records we will describe.
- All the records have the `c_` prefix.
- A special case is the record `c_def`. Well, it is not a record per se, but it is mentioned as one in `core_parse.hrl` to denote the type of some record fields. It actually is a tuple of two elements, let's say {T1, T2}, where T1 and T2 are of type Tree. It is used when we want to show that T1 is defined by T2. We will see its exact uses as we describe the records in `core_parse.hrl`.
- Every record has an annotation field which is initialized to the empty list. Annotations are a list of associations between constant literals and a phrase. They are optional and their interpretation is implementation dependent. For simplicity, we will always have an empty annotation list in our records.
- We will mention the term environment. An environment is simply a mapping from names to Erlang values. Every expression is evaluated in a given environment that binds variables and functions to their values.
- Every evaluation of an expression can terminate normally and produce a sequence of values or end unexpectedly and throw an exception.

Root Node of a Syntax Tree

The root of a syntax tree will always be a `c_module` record (Listing 2.1). The basic unit that is compiled is a module, a single source code file that consists of the module name, the list of exported functions, some optional attributes, such as type definitions and function specifications, and the definitions of the exported and internal functions.

As we see, these correspond with the fields of the `c_module` record. The `name` field is always a `c_literal` record that holds the name of the module. The `exports` field is a list of `c_var` records that hold the names and arities of the exported functions. The `attrs` field is a list of the aforementioned `c_def` constructs. In this case, each `c_def` consists of two `c_literal` records. The first one denotes the type of attribute (spec, type etc.) and the second one its definition. The interpretation of module attributes is implementation specific and we will discuss some of its uses in detail in Section 2.4. The `defs` field is also a

```
-record(c_module, {anno=[]  % annotation list,
                   name,    % name :: Tree,
                   exports, % exports :: [Tree],
                   attrs,   % attrs :: [#c_def{}],
                   defs}).  % defs :: [#c_def{}]
```

Listing 2.1: The `c_module` record definition

list of `c_def` constructs. The first element of each `c_def` is a `c_var` record that represents the name and arity of the defined function. The second element is a `c_fun` record that holds the definition of the respective function.

Expressions

Ordered sequence of single expressions (Listing 2.2). With the term single expression we mean that we want an expression that its evaluation will yield a value. An ordered sequence is the result of evaluating an expression and is by no means a value per se, meaning that we cannot have a sequence of sequences. The length of the `es` list denotes the degree of the sequence.

```
-record(c_values, {anno=[],
                   es}).    % es :: [Tree]
```

Listing 2.2: The `c_values` record definition

Atomic literal (Listing 2.3). The `c_literal` record holds a literal, a notation of constant value such as an integer or an atom. It represents the Erlang value denoted by that literal.

```
-record(c_literal, {anno=[],
                    val}).  % val :: literal()
```

Listing 2.3: The `c_literal` record definition

Variable name (Listing 2.4). The `name` field can be an integer() or an atom() (variable's name) or a {atom(), integer()} (function's name and arity). It evaluates to the value that is bound to the variable or function in the current environment. We should mention that when it comes to a function, the value that we expect is a closure.

```
-record(c_var, {anno=[],
                name :: cerl:var_name()}).
```

Listing 2.4: The `c_var` record definition

Erlang tuple (Listing 2.5). The `c_tuple` record holds a list of n expressions. This evaluates to the Erlang n-tuple where each element is the result of the evaluation of the respective

expression of the `es` list in the current environment. If the `es` field is an empty list, then the result is the 0-tuple {}.

```
-record(c_tuple, {anno=[],
                  es}). % es :: [Tree]
```

Listing 2.5: The `c_tuple` record definition

Erlang list constructor (Listing 2.6). Let `e` be the current environment. The `hd` field is the expression that evaluates to the head of the cons cell in `e`. The `tl` field is the expression that evaluates to the tail of the cons cell in `e`.

```
-record(c_cons, {anno=[],
                 hd,    % hd :: Tree,
                 tl}).  % tl :: Tree
```

Listing 2.6: The `c_cons` record definition

Erlang binary (Listing 2.7). The `segments` field is a list of `c_bitstr` records that each evaluates to an Erlang bitstring. The value of the `c_binary` is the concatenation of these individual bitstrings in the order defined in the segments list. All the segments are evaluated in same environment as the `c_binary` record.

```
-record(c_binary, {anno=[],
                   segments}).  % segments :: [#c_bitstr{}]
```

Listing 2.7: The `c_binary` record definition

Erlang bitstring (Listing 2.8). The `val` field specifies a value to be encoded as a bitstring, whereas the rest of the fields control the encoding. These control fields are evaluated at load time so they ought to be `c_literal` records. This limitation has complicated the implementation of the Core Erlang interpreter and its effect will be discussed in chapter 4. The `size` field is evaluated either to a non negative integer or to the atom `'all'`. The `unit` field evaluates to number between 1 and 256. The `type` field evaluates to either integer, float or binary. The `flags` field evaluates to a list that contains a value for the signedness (signed or unsigned) and a value for the endianess (big, little or native) of the bitstring. The specifics of the encoding will be discussed when we present the Erlang Type System in section 2.4.

Let definition (Listing 2.9). First of all, we evaluate the expression that is held in the `arg` field in the current environment `e`. This evaluation yields a sequence of `n` values `s`. We then expect a list of `n` `c_var` records in the `vars` field that represent the variables that are to be bound. We then bind these variables to their respective value in the sequence `s` and add them to the current environment `e` thus creating the new environment `e'`. The value of the whole let expression is the evaluation of the expression of the `body` field in the new environment `e'`.

Sequence of expressions (Listing 2.10). This record is syntactic sugar for a `c_let` without

```
-record(c_bitstr, {anno=[],
                   val,      % val :: Tree,
                   size,     % size :: Tree,
                   unit,     % unit :: Tree,
                   type,     % type :: Tree,
                   flags}). % flags :: Tree
```

Listing 2.8: The `c_bitstr` record definition

```
-record(c_let, {anno=[],
                vars,    % vars :: [Tree],
                arg,     % arg :: Tree,
                body}). % body :: Tree
```

Listing 2.9: The `c_let` record definition

a `vars` field. Its only purpose is to evaluate the expression of the arg field before the expression of the `body` field. Both expressions are evaluated in the current environment.

```
-record(c_seq, {anno=[],
                arg,       % arg :: Tree,
                body}).    % body :: Tree
```

Listing 2.10: The `c_seq` record definition

Case statement (Listing 2.11). This is the main control flow construct (along with `c_receive`). We begin with evaluating the expression of the `arg` field in the current environment which yields a sequence of values. Then, this sequence is matched against each of the clauses in the `clauses` field list until one of them succeeds. It is certain that at least one will succeed since the compiler adds a catch-all clause at the end of every non-exhaustive case statement that will raise the appropriate exception. The order of the clauses is important since only the first one that matched will be selected. For what happens when a clause is selected, refer to the explanation of the `c_clause` record.

```
-record(c_case, {anno=[],
                 arg,       % arg :: Tree,
                 clauses}). % clauses :: [Tree]
```

Listing 2.11: The `c_case` record definition

Function definition (Listing 2.12). The `vars` field containts a list of `c_var` records that denote the parameters of the function. The `body` field contains an expression that defines the body of the function. A `c_fun` is used to define three types of functions.

- An exported or internal function of a model
- An anonymous function

- A function that is bound in a letrec statement

In all three cases, the `c_var` evaluates to the closure that is defined by abstracting the expression in the `body` field with respect to the parameters in the `vars` field in the current environment.

```
-record(c_fun, {anno=[],
                vars,   % vars :: [Tree],
                body}). % body :: Tree
```

Listing 2.12: The `c_fun` record definition

Letrec statement (Listing 2.13). We must note that only functions can be bound by a Core Erlang `letrec` statement. Therefore, the `c_def` we encounter in `defs` field list has a `c_var` as its first element that denotes the name and arity of the function that is to be bound. The second element is a `c_fun` record that holds the definition of the particular function. Once we bind all the functions to their definitions and add them to the current environment `e`, we evaluate the expression of the `body` field in the resulting environment `e'`.

The definition of `e'` can be a little tricky. The environment `e'` is the smallest environment such that:

- contains all the elements of `e`, except for the functions that will be bound by the `letrec`,
- and every such function will be evaluated in `e'` itself.

This is a circular definition and its implementation requires a recursive environment. Essentially, we need an environment that arranges to evaluate the values for its bindings in an environment where the bindings are already in place. We will discuss how we implemented this feature in chapter 4.

```
-record(c_letrec, {anno=[],
                   defs,    % defs :: [#c_def{}],
                   body}).  % body :: Tree
```

Listing 2.13: The `c_letrec` record definition

Function application (Listing 2.14). The `op` field contains the expression that will evaluate to a closure `f` in the current environment. The `args` field contains a list of `n` expressions that will evaluate to `n` values respectively that will comprise the argument list of the closure. The result of the expression is the application of the argument list to the closure `f`. It is expected that the arity of the closure `f` will be the same as the length of the argument list.

Module-qualified function call (Listing 2.15). The `module` and `name` fields contain the information of the function with an explicit module, function name, and arity (MFA) that

```
-record(c_apply, {anno=[],
                  op,       % op :: Tree,
                  args}).   % args :: [Tree]
```

Listing 2.14: The `c_apply` record definition

will be called. They both evaluate to atoms. The `args` field contains a list of expressions that each is evaluated in the current environment. The resulting list of values will be passed as the argument list of the MFA. It is essential that the MFA is in its module's exported list.

```
-record(c_call, {anno=[],
                 module,    % module :: Tree,
                 name,      % name :: Tree,
                 args}).    % args :: [Tree]
```

Listing 2.15: The `c_call` record definition

Primitive operation (Listing 2.16). The `args` field contains a list of expressions that are evaluated in the current environment and yield a list of values respectively. The `name` field contains the primitive operation (mainly a `c_var`) to be performed. We then apply the list of values to the primitive operator to get the value of the whole expression.

The evaluation of a primitive operation is implementation dependent and may depend on many things, like the internal state of the process, the external state of the world etc. A common case where primitive operations are used in is bit comprehensions.

```
-record(c_primop, {anno=[],
                   name,    % name :: Tree,
                   args}).  % args :: [Tree]
```

Listing 2.16: The `c_primop` record definition

Try–catch statement (Listing 2.17). The `arg` field contains the main expression that we will try to evaluate in the current environment `e`. Depending on the result of this evaluation we have two alternatives:

1. The evaluation completes normally and yields a sequence of values. Then, we bind these values to the list of variables contained in the `vars` field and add these bindings to `e` thus creating the environment `e'`. The result of the try-catch expression is the result of the evaluation of the expression of the `body` field in the environment `e'`.

2. The evaluation stops abruptly raising an exception. Then, we bind the exception to the list of variables contained in the `evars` field and add these bindings to `e` thus creating the environment `e''`. The result of the try-catch expression is the result of the evaluation of the expression of the `handler` field in the environment `e''`.

```
-record(c_try, {anno=[],
                arg,    % arg :: Tree,
                vars,   % vars :: [Tree],
                body,   % body :: Tree
                evars,  % evars :: [Tree],
                handler}).  % handler :: Tree
```

Listing 2.17: The `c_` try record definition

Catch statement (Listing 2.18). This is syntactic sugar for a `c_try` with only the `body` field. This works as follows: We evaluate the `body` expression in the current environment and monitor its evaluation. There are two alternatives:

1. If the evaluation completed normally, the value of the whole expression is the value of the `body`.
2. If the evaluation stops abruptly and yields an exception, we do one of the following:
   (a) If it is a `throw` exception, we return the reason of the exception `r`.
   (b) If it is an `exit` exception, we return the reason of the exception `r` wrapped in the tuple `{'EXIT', r}`.
   (c) If it is an `error` exception, we return the reason of the exception `r` along with the stacktrace `s` wrapped in the tuple `{'EXIT', r, s}`.

```
-record(c_catch, {anno=[],
                  body}).   % body :: Tree
```

Listing 2.18: The `c_catch` record definition

Receive expression (Listing 2.19). This is the second type of control flow construct that exists in Core Erlang. Its evaluation is a bit complex thus is divided into stages.

1. The `timeout` field contains the expiry expression, which the first thing we evaluate in the current environment `e`. It may yield a non negative integer `t` or the atom `'infinity'`. If the result is not `'infinity'`, we start a timer that expires in `t` milliseconds.
2. Then, we run through the process's mailbox from start to end and try to match a message against one of the clauses in the `clauses` list field in the current environment `e`. This may have two outcomes:
   (a) A message matches one of the clauses and yields some mappings that are added to `e` to create the environment `e'`. Then, this message is deleted from the mailbox and we evaluate the expression indicated by the matched clause in the environment `e'`.
   (b) We tried all the current messages but none matched any of the clauses. If the expiry expression evaluated to 0 or the timer has expired, we evaluate the expression of the `action` field in `e`. Otherwise, we suspend the evaluation.

3. The evaluation is recovered if a message has arrived to the mailbox or if the timer has expired. In the former scenario we re-enter stage 2, whereas in the latter one we evaluate the expression of the `action` field in `e`.

```
-record(c_receive, {anno=[],
                    clauses,   % clauses :: [Tree],
                    timeout,   % timeout :: Tree,
                    action}).  % action :: Tree
```

Listing 2.19: The `c_receive` record definition

Clauses and Pattern Matching

Clause template (Listing 2.20). This record describes the template of a Core Erlang clause. The `pats` field contains the sequence of patterns that will be tried, in left-to-right order, against the sequence of values so as to determine if the clause matches the sequence. The patterns are evaluated in the current environment `e`.

- If the patterns and the sequence of values matched, some new bindings may have been generated which added to `e` will create the new environment `e'`. We then evaluate the expression of the `guard` field.
  - If the result is `'true'`, the clause has been definitively matched thus we continue to evaluate the expression of the `body` field in the environment `e'`.
  - If the result is `'false'`, the clause does not match so we move on to the next clause.
- If the patterns did not match the sequence, the next clause in order is tried.

```
record(c_clause, {anno=[],
                  pats,     % pats :: [Tree],
                  guard,    % guard :: Tree,
                  body}).   % body :: Tree
```

Listing 2.20: The `c_clause` record definition

A pattern is represented by the following records:

- A `c_literal` record represents an atomic literal pattern.
- A `c_var` record represents a variable name pattern.
- A `c_tuple` record represents a tuple pattern.
- A `c_cons` record represents a list constructor pattern.
- A `c_binary` record represents a binary pattern.

- A `c_alias` record (Listing 2.21) represents an alias pattern. The `pat` field contains the pattern that will be matched against a value `v`. If the match succeeds then a mapping is generated that binds the variable of the `var` field to `v`. If not, the whole match fails.

```
1 -record(c_alias, {anno=[],
2                   var,      % var :: Tree,
3                   pat}).    % pat :: Tree
```

Listing 2.21: The `c_alias` record definition

2.4 Erlang Types and Function Specifications

Erlang is a dynamically typed language. Type information exists in any piece of data and is checked during every operation at runtime so as to provide type safety. There are primitive types built in the language that can be combined by the programmer to create custom ones. There is even a notation for declaring the type specification of a function concerning its intended use. This piece of information is critical for an effective testing tool since it will report bugs relevant to the program's actual use [17]. We will briefly describe the types in Erlang and the function specifications and focus on its representation in the Core Erlang Abstract Syntax Tree.

Any piece of data in Erlang is called a term. A term can belong to a predefined (or built-in) type or to a user-defined type constructed from the predefined ones.

2.4.1 Predefined Types

The predefined types are the following:

- *Integer.* An integer is a numeric literal.
- *Float.* A float is a numeric literal as well.
- *Atom.* An atom is a literal, a named constant.
- *Bit string and binary.* A bit string is a series of bytes stored in an area of untyped memory. If the number of bits is divisible by eight, it is also a binary.
- *Reference.* A reference is a term which is unique in an Erlang runtime system.
- *Fun.* A fun is callable function object represented with a closure.
- *Port identifier.* A port identifier identifies an Erlang port.
- *Pid.* A pid, or process identifier, is a handle that uniquely identifies an Erlang process.
- *Tuple.* A tuple is a compound data type with a fixed number of terms (not necessarily of the same type).

- *List.* A list is a compound data type with a variable number of terms (not necessarily of the same type).

There are also some types that are defined in terms of the above primitives and are essentially syntactic sugar.

- *Number.* A number is either an integer or a float.
- *Char.* A char is an integer in the range from 0 to 1114111.
- *String.* A string is shorthand for a list of chars.
- *Record.* A record is actually a tuple with an atom as its first element. Records are declared with `-record` compiler attribute and are converted to tuples at compile time.
- *Boolean.* There is no boolean type in Erlang. However, the atoms `'true'` and `'false'` have a special use for acting as boolean values.

2.4.2 User Defined Types

Erlang allows the programmer to define his own types using the `-type` and `-opaque` compiler attributes. User defined types are often useful for parametric or recursive types and may serve as type aliases.

The basic syntax of a type is an atom, i.e. the type's name, followed by closed parentheses. Type declarations can also be parametrized by including type variables between the parentheses, used for parametric types. The definition is a valid type expression (Subsection 2.4.3). Let's note that users are not allowed to define types with the same names and arity as the predefined or built-in ones.

2.4.3 Representation of the Types on the Core Erlang AST

Type expressions are used in type definitions and function specifications. In this subsection we will describe the notation for writing type expressions and we will see how the compiler transforms them into the Core Erlang AST.

We should note that the Erlang compiler stores debug information containing the source code's line numbers in the AST. Thus, you will see the symbol `Ln` used in the Core Erlang AST representation of the type notation, which denotes the line number it refers to in the source code file.

Notation of Predefined Types

Type Notation	Description	Core Erlang Representation
integer()	Any Erlang integer	`{type, Ln, integer, []}`
<Int>	Singleton integer type (only the specific *<Int>*)	`{integer, Ln, <Int>}`
<Lo>...<Hi>	Integers in the range from *<Lo>* to *<Hi>*	`{type, Ln, range, [Lo, Hi]}` where `Low :: {integer, Ln, <Lo>}`, `High :: {integer, Ln, <Hi>}`
neg_integer()	Any negative Erlang integer	`{type, Ln, neg_integer, []}`
non_neg_integer()	Any non-negative Erlang integer	`{type, Ln, non_neg_integer, []}`
pos_integer()	Any positive Erlang integer	`{type, Ln, pos_integer, []}`
float()	Any Erlang float	`{type, Ln, float, []}`

Table 2.1: Notation of numeric types

Type Notation	Description	Core Erlang Representation
tuple()	Any Erlang tuple	`{type, Ln, tuple, any}`
{}	Singleton tuple type (only the 0-tuple)	`{type, Ln, tuple, []}`
{*Type*$_1$,...,*Type*$_N$}	An tuple of N elements of types *Type*$_1$,...,*Type*$_N$ respectively	`{type, Ln, tuple, [T1,...,TN]}` where `Ti` is a valid type expression

Table 2.2: Notation of tuples

Type Notation	Description	Core Erlang Representation
list()	Any Erlang list	`{type, Ln, list, []}`
nonempty_list()	Any Erlang non-empty list	`{type, Ln, nonempty_list, []}`
[]	Singleton list type (only the empty list)	`{type, Ln, nil, []}`
[*Type*$_1$|...|*Type*$_N$]	An list of elements of types *Type*$_1$,...,*Type*$_N$	`{type, Ln, list, [T1,...,TN]}` where `Ti` is a valid type expression
[*Type*$_1$|...|*Type*$_N$, ...]	An non-empty list of elements of types *Type*$_1$,...,*Type*$_N$	`{type, Ln, nonempty_list, [T1,...,TN]}` where `Ti` is a valid type expression

Table 2.3: Notation of lists

Type Notation	Description	Core Erlang Representation
bitstring())	Any Erlang bitstring	`{type, Ln, bitstring, []}`
binary()	Any Erlang binary	`{type, Ln, binary, []}`
«_ :M»	A bitstring that is M bits long	`{type, Ln, binary, [{integer, Ln, M}, {integer, Ln, 0}]}`
*«_ :_ *N»*	A bitstring that is (k*N) bits long	`{type, Ln, binary, [{integer, Ln, 0}, {integer, Ln, N}]}`
*«_ :M, _ :_ *N»*	A bitstring that is M + (k*N) bits long	`{type, Ln, binary, [{integer, Ln, M}, {integer, Ln, N}]}`

Table 2.4: Notation of binaries and bit strings

Type Notation	Description	Core Erlang Representation
fun()	Any Erlang function	`{type, Ln, fun, []}`
fun((...) -> Type)	A function of any arity that returns Type	`{type, Ln, fun, [{type, Ln, any}, {type, Ln, Type, []}]}`
fun(() -> Type)	A function of 0 arity that returns Type	`{type, Ln, fun, [{type, Ln, product, []}, {type, Ln, Type, []}]}`
fun((Tlist) -> Type)	A function with Tlist as arguments that returns Type	`{type, Ln, fun, [{type, Ln, product, Tlist}, {type, Ln, Type, []}]}`

Table 2.5: Notation of function types

Type Notation	Description	Core Erlang Representation
term()	Any Erlang term	`{type, Ln, term, []}`
any()		`{type, Ln, any, []}`
none()	No type	`{type, Ln, none, []}`
no_return()	No return type (Used when an exception is raised)	`{type, Ln, no_return, []}`
$Type_1$ | ... | $Type_N$	An union of N terms of types $Type_1$,...,$Type_N$ respectively	`{type, Ln, union, [T1,...,TN]}` where `Ti` is a valid type expression
atom()	Any Erlang atom	`{type, Ln, atom, []}`
<Atom>	Singleton atom type (only the specific *<Atom>*)	`{atom, Ln, <Atom>}`
reference()	Any Erlang reference	`{type, Ln, reference, []}`
port()	Any Erlang port identifier	`{type, Ln, port, []}`
pid()	Any Erlang process identifier	`{type, Ln, pid, []}`

Table 2.6: Notation of other useful types

Type Notation	Description	Core Erlang Representation
number()	Any Erlang integer or float	`{type, Ln, number, []}`
char()	Any Erlang char	`{type, Ln, char, []}`
string()	Any Erlang string	`{type, Ln, string, []}`
nonempty_string()	Any non-empty Erlang string	`{type, Ln, nonempty_string, []}`
boolean()	One of the atoms `'true'` or `'false'`	`{type, Ln, boolean, []}`
byte()	Any integer in the range from 0 to 255	`{type, Ln, byte, []}`
node()	An atom that represents an Erlang VM node	`{type, Ln, node, []}`
module()	An atom that represents a module	`{type, Ln, module, []}`
mfa()	An tuple that represents an Erlang MFA ({*Module, Function, Arity*})	`{type, Ln, mfa, []}`

Table 2.7: Notation of aliases for other types

Notation of User Defined Types

We have mentioned that the root element of a Core Erlang AST is a `c_module` record. The `attrs` field contains a list of `c_def` records concerning implementation specific information. One type of such information is the declaration of a user defined type or a record. Let's call such a `c_def` record, a `TypeAttrDef`. In Table 2.8 we can see the basic syntax of a `TypeAttrDef`.

```
TypeAttrDef :: {{c_literal, Ln, type}, {c_literal, Ln, [TypeAttr]}}
TypeAttr    :: RecordDef | TypeDef
RecordDef   :: {{record, RecordName :: atom()}, (RecordField)+, []}
RecordField :: UntypedRF | TypedRF
UntypedRF   :: {record_field, Ln, FldName :: atom()}
             | {record_field, Ln, FldName :: atom(), DefVal :: term()}
TypedRF     :: {typed_record_field, UntypedRF, Type}
TypeDef     :: {TypeName :: atom(), Type, (FreeVar)*}
FreeVar     :: {var, Ln, VarName :: atom()}
Type        :: Valid Type Representation
```

Table 2.8: Basic Grammar of a `TypeAttrDef`

2.4.4 Function Specifications

A function specification is a contract which explicitly states the programmer's intented use for a function. It is declared using the `-spec` compiler attribute. The basic syntax is the following:

$$\texttt{-spec Fun(ArgType}_1\texttt{, ArgType}_2\texttt{, ..., ArgType}_N\texttt{)} \rightarrow \texttt{ReturnType}$$

There are also other ways of stating a function specification which are variants of the basic one. They may be used for documentation purposes, to express overloaded specifications etc. They are described in detail in the Erlang Type Specifications [9].

The information of a function's specification is stored in Core Erlang AST. You can find it as a `c_def` in the `attrs` field of the root `c_module` record, which we'll call `SpecAttrDef`. In Table 2.9 we can see the basic syntax of a `SpecAttrDef`.

```
SpecAttrDef :: {{c_literal, Ln, spec}, {c_literal, Ln, [SpecAttr]}}
SpecAttr    :: {{F :: atom(), A :: byte()}, (Spec)+}
Spec        :: Fun | BoundedFun
Fun         :: {type, Ln, 'fun', [Product, Type]}
Product     :: {type, Ln, product, (Type)*}
BoundedFun  :: {type, Ln, bounded_fun, [Fun, (Constraint)*]}
Constraint  :: {type, Ln, constraint, SubTypeCnst}
SubTypeCnst :: [{atom, Ln, is_subtype}, [VarName :: atom(), Type]]
Type        :: Valid Type Representation
```

Table 2.9: Basic Grammar of a `SpecAttrDef`

Chapter 3

Concolic Testing

3.1 The Basic Idea

The word ”concolic” is a portmanteau of the words concrete and symbolic and it depicts the hybrid nature of the concolic software testing technique. Concolic testing interleaves concrete execution with symbolic execution in order to achieve high path coverage. The basic idea is to use symbolic execution to generate inputs that will hopefully steer the program to alternate execution paths whilst concrete execution is used to guide the symbolic execution along a concrete path [14].

During the execution, we keep a concrete and a symbolic state. The concrete state maps the variables to their concrete values whereas the symbolic state maps the variables that have non-concrete values to their symbolic representation. In order to start the process we need a seed concrete input, that is either provided by the user or is randomly generated. We then execute the program with this input and gather symbolic constraints at all the conditional statements along the execution. Each constraint is essentially a logical formula which is expressed as a function of the input symbolic variables and denotes the boolean value of the respective conditional statement during the execution. The conjunction of all the constraints is the path predicate. The next step is to feed the path predicate to a constraint solver in order to infer new variants of the input that will likely guide the program to a different feasible execution path. We, then, use the generated input as our new seed. This process is repeated systematically until every feasible execution path is explored.

In the concurrent version, a new path may be explored by producing a different interleaving of the program threads instead of generating a different input. This instrumentation records and permutes the various events that lead to data and lock races [20, 21].

Concolic testing is also widely used for verification purposes [15] as it can assist eliminate the false positive warnings of the static analyzers [24, 11, 13].

3.1.1 A Simple Example

Consider the following simple Erlang function (Listing 3.1) that has an erroneous branch:

```
1 foo(X, Y) ->
2   Z = 2 * Y,
3   case X =:= 100000 andalso X < Z of
4     false -> ok;
5     true  -> error(assertion)
6   end.
```

Listing 3.1: Simple Erlang function

Any black-box testing technique would most likely require a lot of time, if ever, to locate and reproduce the failure.

Let's see how concolic testing will help locate promptly the error. As we said, we need a seed input to initialize our search. We arbitrary choose the input `X = 0` and `Y = 0` and execute the code with these concrete values. In line 2, we get `Z = 2 * Y = 0`. In line 3, the condition evaluates to `false` as `X =/= 100000` and `erlang:andalso/2` is short-circuited. Therefore, we follow the "false" branch and the function returns `ok` and ends normally. At the same time we execute the function symbolically following the same path as the concrete execution. This yields the path constraint `(X =/= 100000)`.

In order to force the program to follow a different a execution path in the next iteration, we negate the only predicate in the patch constraint. This yields the binding `X = 100000`, so we execute the function again with the concrete values `X = 100000` and `Y = 0`. In line 2, we still get `Z = 0`. In line 3, the expression `X =:= 100000` evaluates to `true` but the expression `X < Z` evaluates to `false`, thus making the whole condition evaluate to `false` again. The function again returns `ok` and ends normally. After the symbolic execution we get the path constraint `(X =:= 100000)` $\wedge$ `(X >= 2 * Y)`.

This time we invoke the solver to negate the second predicate while keeping the first predicate true. The solution may be possibly be `X = 100000` and `Y = 50001`. By running the function with this concrete input, we manage to reach the branch that raises the exception.

As we see, this approach manages to locate the error in only three iterations.

3.1.2 Strengths and Limitations

The concolic approach, as any approach that has an internal perspective of a system, allows for thorough testing and is able to reveal errors that would otherwise remain hidden with random testing. It has also the prospect of providing test suites with high code coverage.

However, this approach has a number of challenges and there is a lot of research on how to tackle them.

Incompleteness of Constraint Solving.

The main bottleneck of a concolic execution is the constraint solving. Real world software may produce quite complex and non-linear constraints that will require a lot of time for the SMT solver to process and solve them. In many cases, the solver may actually be unable to

solve the given set of constraints. This limits the efficiency and may lead to poor coverage as there will be no testcases generated for some feasible execution paths [23, 22].

Path Explosion.

Even the smallest programs will most likely generate huge amounts of trace data and subsequently create long path predicates. The length of the path is exponential to the number of the branches in the program. It is obvious that, even if some paths are deemed to be infeasible, there need to be some search heuristics [16] in order to steer the concolic execution to possible bugs given that we want to get results in a reasonable amount of time [2, 3, 4].

Non-deterministic Behaviour of Programs.

The main reason why we use an SMT solver is our belief that the testcases it will generate will exercise a new execution path of the code [12]. So there is a big problem when we predict that the program will follow a specific path and it turns out that we were disillusioned. This may be the case with programs that show non-deterministic behaviour and will cause the concolic testing to be extremely ineffective.

3.1.3 Challenges in Erlang

Erlang provides unique challenges to implementing a concolic testing tool. First of all, there is hardly any shared memory so there are few data races for such a technique to find. In addition, Erlang is mainly used for concurrent applications, and concolic execution is much more complicated for multi-process systems [19].

In addition, Erlang is a functional language that heavily uses algebraic datatypes. There is a very small number of SMT solvers that support algebraic datatypes. Still, this theory is not as advanced and efficient as the theories of bitvectors and arrays that are mainly used for imperative languages [7].

3.1.4 Z3 SMT Solver

The theorem prover we chose to use is Z3 from Microsoft Research [6]. It is an efficient SMT solver that is targeted at solving problems in software analysis and software verification [7]. It supports the SMT-LIBv2 standard and it provides APIs for C/C++, .NET, OCaml and Python. We used the Python API, namely Z3Py.

We use Z3 to check the satisfiability of logical formulas and get an instance of the universe in which these formulas are satisfiable. Z3 is based on first-order logic and decision procedures [10].

We should note that we used it under the Microsoft Research License Agreement for Non-Commercial use.

Datatypes

In Z3, expressions are directed acyclic graphs. Every expression has a `sort`, i.e. a type. For example, we can define x to be an integer variable with $x = Int(`x`)$, which is equal to $x = Const(`x`, IntSort())$. Z3 supports all the basic types you will need in a language like C, i.e. booleans, integers, floats, arrays and bitvectors. It also supports algebraic datatypes. It is a convenient way to define finite lists, trees, tuples, records, enumerations, mutually recursive datatypes etc.

For example, in Listing 3.2 we see how we can a define a finite list of integers. There are two phases.

- At first, we declare the *List* datatype and create a placeholder for the definitions of its constructors and accessors. Then, we declare all the constructors with their respective accessors. Each accessor is associated with a sort or a reference to the datatypes being declared. In our case, we declare *cons* that builds a *List* using an *Integer* and a *List*, and *nil* that takes no arguments.
- After all the constructors have been declared, we create the actual datatype in Z3.

Functions

Z3 supports most of Python operators such as $+, -, <$ and has some simple built-in functions that help in type conversions, like $ToInt$. For more complex operations, one can define one's own functions. However, functions in pure first-order logic are uninterpreted which means that there is no interpretation attached when declared. In addition, they have no side-effects and are total. For example, in Listing 3.3 we declare a function $f : Int \to Int$ and two integer variables x and y. We then define an empty universe and assert the constraints $x > 42$, $x < y$ and $f(f(10)) = 1$. Functions are total so we are certain that $f(10)$ is in f's set of inputs and therefore $f(f(10))$ is valid.

A valid model that Z3 returns and that satisfies the above constraints is

$$[y = 44, x = 43, f = [10 \to 0, 0 \to 1, else \to 0]]$$

As we see, f is defined on all integers.

SMT-LIBv2 does not allow recursive function definitions without the use of quantifiers. In Listing 3.4, we see how we can define the length of a list. First, we declare an uninterpreted function *len* and then add its definition as a quantified axiom. If we try to prove

$$len(x :: xs) = 1 + len(xs), \text{ where } x : Int \text{ and } xs : List,$$

the prover will not be able to respond *sat* or *unsat* as it cannot yet prove theorems that require induction. However, if we try to prove the negation,

$$\neg(len(x :: xs) = 1 + len(xs))$$

it will respond *unsat* so the first theorem is satisfiable. This is the trick for simple goals, like this one, but for more complex ones, Z3 will start responding with *unknown* or *timeout*.

```
# import z3py
from z3 import *

# ###################
# List Declaration
# ###################

# Phase 1
# Declare a List of integers
List = Datatype('List')
# Constructor cons: (Int, List) -> List
# Accessors: hd(List) -> Int, tl(List) -> List
List.declare('cons', ('hd', IntSort()), ('tl', List))
# Constructor nil: List
List.declare('nil')

# Phase 2
# Create the actual datatype
List = List.create()

# ###################
# Example of Usage
# ###################

# Should print 'True' as List is a now a valid type
print is_sort(List)

# Define shorthands for List Constructors and Accessors
cons = List.cons
hd = List.hd
tl = List.tl
nil = List.nil

# Define a list
l = cons(42, cons(24, nil))

# Will print '42'
print simplify(hd(l))
# Will print 'cons(24, nil)'
print simplify(tl(l))
# Will print 'False'
print simplify(l == nil)
```

Listing 3.2: Define Finite List of Integers in Z3

Suitability for Erlang

Z3 has the huge advantage of supporting algebraic datatypes. This feature allows us to describe at a higher level the basic linear constructs of Erlang, such as lists and tuples. In addition, it provides a user-friendly Python API that you can easily invoke from Erlang ports.

However, it does not allow the definition of recursive functions thus hindering the emulation of the semantics of any Erlang function that operates on recursive data structures.

```
# import z3py
from z3 import *

f = Function('f', IntSort(), IntSort())  # declare f : Int -> Int
x, y = Ints('x y')                       # declare the Integer Variables x, y

s = Solver()                             # create an empty universe
s.add(x > 42, x < y, f(f(10)) == 1)      # assert the constraints

print s.check() # will print 'sat'
print s.model() # will print an interpretation of the universe
```

Listing 3.3: Simple function in Z3

```
# import z3py
from z3 import *

# Define List to be a list of integers
List = Datatype('List')
List.declare('cons', ('hd', IntSort()), ('tl', List))
List.declare('nil')
List = List.create()

# Declare some variables
x = Int('x')
xs = Const('xs', List)
s = Solver()

# Declare len : List -> Int
len = Function('len', List, IntSort())

# Assert defining equations for len as an axiom
s.add(
  ForAll(xs,
    If(List.is_nil(xs), len(xs) == 0, len(xs) == 1 + len(List.tl(xs)))
  )
)

# Attempt #1
s.push()
# Try to prove that len(x::xs) == 1 + len(xs)
s.add(1 + len(xs) == len(List.cons(x, xs)))
# Will result in 'timeout' as Z3 cannot handle
# (for now) proofs that require induction
print s.check()
s.pop()

# Attempt #2
s.push()
# Try to prove the negation of len(x::xs) == 1 + len(xs)
s.add(Not(1 + len(xs) == len(List.cons(x, xs))))
# Will print 'unsat' thus the negation is satisfiable
print s.check()
s.pop()
```

Listing 3.4: Recursive functions in Z3

Chapter 4

CutEr

CutEr is the concolic testing tool for Erlang that we developed. In its current form it can test sequential programs and can reason about types that are based on integers, floats, atoms, lists and tuples. It comprises of three main components.

- The *Core Erlang interpreter* that executes the program both concretely and symbolically while collecting the symbolic constraints.
- The *Integration with Z3 SMT solver* that is delegated to properly encode the symbolic constraints and invoke Z3 to solve them.
- The *Instrumentation algorithm* that drives the exploration of the execution path state space. It coordinates the other two components and reports all the findings to the user.

4.1 Core Erlang Interpreter

4.1.1 Why Core Erlang?

The most basic operations of the concolic technique are the concrete and symbolic executions of a program so this is what we opted to primarily implement. Essentially, what we wanted was a component that would take an Erlang program and its arguments as input and would perform the following operations.

- Perform the concrete execution of the program and record the result of the branch statements so as to guide the symbolic execution.
- Perform the symbolic execution of the program and record the symbolic constraints on the branch statements. Then, append the constraints to formulate the path predicate of the specific execution and return it as the output of the component.

The Erlang VM does not have any available trace flags for step execution, let alone symbolic execution. We implemented our own infrastructure for this component.

The only program that features some of the desired functionality, at least for the concrete execution, is the `debugger` that is included in the Erlang/OTP release. We were interested in the ability to single step the code and to monitor the values of all the variables. The `debugger` is actually an interpreter which evaluates the Abstract Syntax Tree that is stored in the debugging information of a module. The level of details and control over the execution that it provides inspired us to implement our own custom interpreter. The other option was to build a VM but with respect to the difficulty and the amount of time required for each solution, we decided that an interpreter seemed a more reasonable choice.

The most well established concolic tools employ tracing and emulation on a low level representation of the program's code. For example, KLEE uses LLVM code [3] and SAGE uses x86 bytecode [12]. This approach has the advantage of emulating optimized code and it works well for languages, like C, Java and .NET, the type system of which does not change dramatically from source code to bytecode. This is not the case with Erlang and we wanted to use a representation that would retain the information on types likes lists and tuples. The reason is that the Z3 prover can handle such recursive datatypes and we wanted to take advantage of this native support.

Core Erlang fulfils this criteria and has clear semantics by design, thus making its evaluation straightforward. The Erlang compiler provides a module that generates the Core Erlang AST from the source code, so we would easily access this representation. If we also take into account that the BEAM bytecode is largely undocumented, choosing the Core Erlang representation for our interpreter seemed natural.

4.1.2 Architecture of the Interpreter

Our interpreter uses four types of processes during a typical concolic execution. Briefly,

- The actual execution takes place in the worker processes, that we call `interpreter processes`.
- A process that will oversee the execution and perform regulatory actions is needed. It is the root of the hierarchical process structure and we call it the `supervisor processes`.
- Our interpreter needs the Core Erlang AST of a program, so there has to be a process that will perform the source code compilation independently and provide the AST to the interpreter processes. We call such processes `code servers`. The interpreter spawns one code server per node.
- The basic goal of our tool is to observe exceptions in the execution of a program. In order to do so, we need a process that will have a link or a monitor to every interpreter process and that will be notified when an exception occurs. Since links are bidirectional and are used mainly for propagating exit signals, we chose to use monitors. We call processes with this role `monitor servers`. The interpreter spawns one monitor server per node.

We can see a sample setup in Figure 4.1. In this example, the program we want to run spawns six processes over two nodes. The execution of their code is emulated by our interpreter therefore our interpreter will spawn six interpreter processes. These processes

request the AST of their code from the code server of their respective node. Of course, we perform some caching on these requests so that we do not need to constantly poll the code servers. The execution is monitored by the monitor server of each node which wait for the interpreter processes to finish either normally or with an exception. These administrative processes are spawned and monitored by the supervisor process. In addition, the supervisor spawns the first interpreter process to begin the execution.

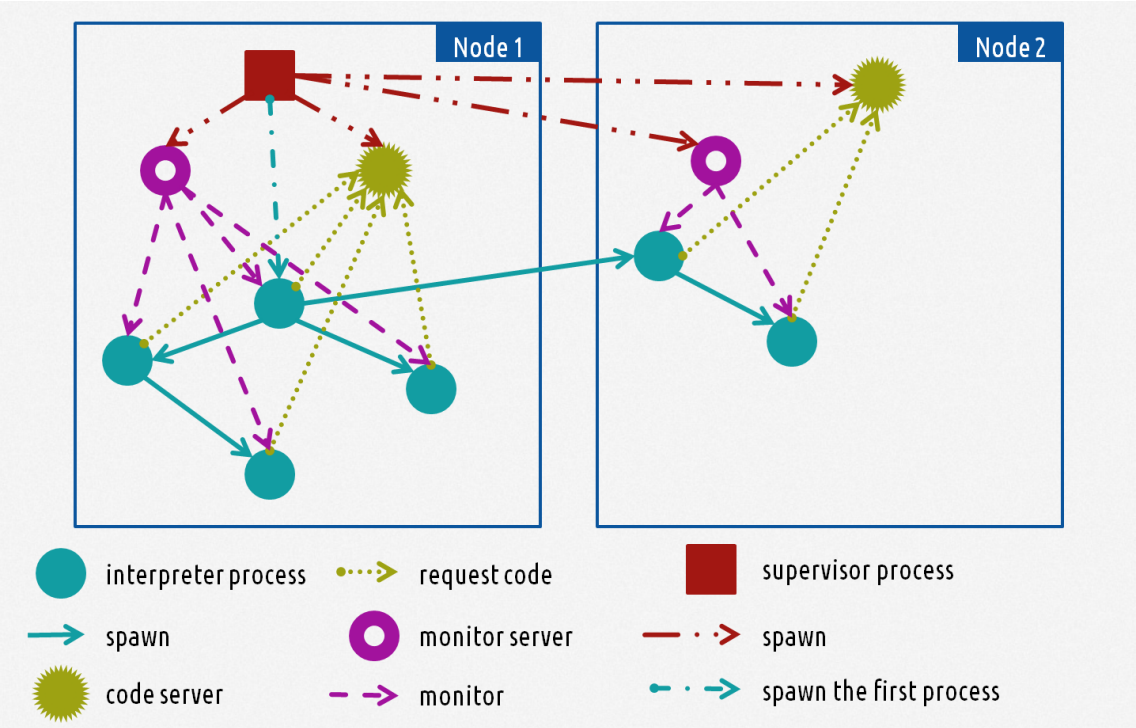

Figure 4.1: The architecture of the interpreter

Supervisor

The supervisor is a `gen_server` that is responsible for starting, stopping and monitoring most of the other processes. More specifically

- it starts a code server and a monitor server in every node
- it starts the first interpreter process that emulates the entry point function
- once the execution has ended, either normally or with an exception, it collects the logs from the code and monitor servers and stores them for later use by the instrumentation algorithm.

Naturally, one supervisor process is spawned per concolic execution.

Code Server

A code server is a `gen_server` that is responsible for loading a module's code so as to be used by an interpreter process.

A named function in Erlang can be uniquely identified with the notation `Module:Function/Arity`. Let's assume that an interpreter process wants to emulate the function `M:F/A`. It doesn't have access to its code so it sends a synchronous request to the code server of its node. Once the code server receives the requests, it checks whether it has already loaded the code of module `M` in an ETS table.

- If it has, it responds with the table's identifier so the interpreter process can retrieve the function's code and continue with the emulation.
- If it has not, it locates the source file of the module and compiles it to Core Erlang. It then parses the output, creates the Core Erlang AST and stores it in a new ETS table. It can now respond with the table's identifier so the interpreter process can retrieve the function's code and continue with the emulation.

With this simple scheme, every function calls means that there will be one call to a code server, which is pretty inefficient. That's why we've employed a means of caching. Whenever an interpreter process receives the identifier of the ETS table where a module's code is stored, it puts it in its process dictionary. In addition, once it retrieves a function's code, it also puts it in its process dictionary so as to eliminate the overhead of reading from the ETS table for future calls to this function. This approach seems to work fine for now as the majority of the programs we tested use functions from a small number of modules and most of them are recursive.

We spawn one code server process per node.

Monitor Server

A monitor server is a `gen_server` that is responsible for monitoring all the interpreter processes that are spawned in a node.

Every time, a new interpreter process is spawned, it first makes a synchronous call to the monitor server of its node. The monitor server starts monitoring the interpreter process. That means that when the process exits, either normally or with an exception, the monitor server will receive a message.

When the monitor server receives a message that a monitored process has exited

- If it was a normal exit, it just removes the process from the list of the active interpreter processes. If the list becomes empty, it sends a message to the supervisor that the execution has ended in that node.
- If an exception was raised, it captures the exception message and notifies the supervisor, which will in turn shut down the concolic execution and report the error.

Let's talk about the behaviour of our interpreter in the case where an exception was raised. In Erlang, it is common for programmers to employ the "let it crash" approach. This approach suggests that error propagation paths should be used for errors between processes. Basically, if a process dies unexpectedly, there is another process that will observe the death and take appropriate actions to recover from the error. In that case,

the execution will most likely continue and the problematic process may be restarted. However, our interpreter will not exhibit the same behaviour as it will shut down the execution once the exception is raised. We chose this approach because every exception is a potential bug and the erroneous behaviour should be recorded despite the fact that the programmer was thoughtful enough to make his program fault-tolerant.

We spawn one monitor server process per node.

Interpreter Process

An interpreter process is a process that emulates the execution of a program and records the symbolic constraints. Assuming the program spawns n processes during its execution in the Erlang VM then n interpreter processes will be spawned in our interpreter.

Concolic execution. As we have mentioned, the basic approach of concolic execution is to execute a program with concrete values and record the outcome of the branch statements in a trace file which can then guide the symbolic execution. So the symbolic execution is done in an asynchronous manner after the concrete execution. In our implementation, we run both simultaneously. Every node of the AST is evaluated both concretely and symbolically before moving on to the next. A trace file is kept with only the symbolic constraints of the control statements. This choice was made primarily because the type of tracing we require generates a large amount of data at an almost constant rate, so the time spent doing I/O for recording and recovering information slowed our interpreter down significantly.

Main function of the interpreter. The above choice has some implications for the design of the interpreter.

- An interpreter is basically a recursive function, let's say `eval`, that takes an AST t and an environment ϱ (mapping of variables to their values) as arguments and returns v which is the value of t, essentially `eval` t $\varrho \rightarrow v$. In our case, the `eval` function needs to have two environments as arguments. The first one will consist of the mappings of the variables to their concrete values and the second one to their symbolic values.
- The `eval` function should not return just one value but rather a pair of values. The concrete value and the symbolic value of the AST node. We used the convention that `eval` will return a 2-element tuple with the concrete value as the first element and the symbolic value as the second element.

Therefore `eval` takes the following form: `eval` t $\varrho_{concrete}$ $\varrho_{symbolic} \rightarrow \{v_{concrete}, v_{symbolic}\}$. The process of evaluating an AST node remains largely unaffected from these changes and it follows the workflow we described in the semantics of each node type in Section 2.3.1. However, we had to make some adjustments and it is interesting to explore them so let's see some more details on the evaluation of Core Erlang.

BIF evaluation. In Erlang, there are some functions that are preloaded to the VM. They can be implemented either in C or Erlang. Most of them belong to the `erlang` module but there are more in other commonly used modules. In either case, we cannot access their source code through information from the VM therefore cannot compile them to

Core Erlang and evaluate them, we have to call them explicitly. These are called built-in functions (BIFs). The problem with these functions is that they expect concrete values and not the construct with the pair of concrete and symbolic values that we created . In order to avoid this incompatibility, whenever we want to call a function, we check if it is a BIF and if so, we just pass the concrete values. However, we ought to return a symbolic value as well. Depending on the BIF, we create the symbolic value that we think is appropriate.

For example, let's consider a simple Erlang function that takes two numbers and returns their sum (Listing 4.1).

```
%% Erlang
foo(X, Y) -> X + Y.

%% Core Erlang
{
  {c_var, [], {foo, 2}},                    %% Name and arity of the function
  {c_fun, [],
    [{c_var, [] ,'X'}, {c_var, [], 'Y'}],   %% Parameters of foo/2
    {c_call, [],                            %% Module-qualified function call
      {c_literal, [], erlang},              %% to erlang:'+'/2
      {c_literal, [], '+'},
      [{c_var, [], 'X'}, {c_var, [], 'Y'}] %% Parameters of erlang:'+'/2
    }
  }
}
```

Listing 4.1: Simple Erlang function and its Core Erlang representation

Let's assume we want to run the concolic interpreter for this function with the concrete input $X = 1$ and $Y = 2$. We start by evaluating the `c_fun`. We abstract the input with two symbolic values, let's say x and y respectively. So we create the concrete environment $[X \to 1, Y \to 2]$ and the symbolic environment $[X \to x, Y \to y]$. We continue to evaluate the `c_call`. In order to do so, we need the code of `erlang:'+'/2` which is infeasible since it is a BIF function. Therefore, we make an explicit call to `erlang:'+'/2` with concrete arguments and get the concrete value 3. The symbolic value should be something that represents $x + y$, let's say the tuple $\{+, x, y\}$. Since our evaluator returns a pair of the concrete and symbolic values, in this case it should return the tuple $\{3, \{+, x, y\}\}$.

A special case are the BIFs that raise exceptions. These are `erlang:raise/3`, `erlang:error/1,2`, `erlang:throw/1`, `erlang:exit/1,2`. When we evaluate them on any input, they raise an exception whatever we gave them as input for reason. So there is no reason to feed them only the concrete input, we can use our concrete / symbolic pair as it is. However, we need to modify the semantics of the try (`c_try`) and catch (`c_catch`) statements to expect this type of reason for the exception.

Spawning Processes. A new process is spawned with a call to specific Erlang BIFs. The most commonly used is `erlang:spawn/3`. Let's assume that a program we are interpreting has the call $spawn(foo, bar, [42])$ somewhere in its code. Once the execution reaches this point, our interpreter will see that this is a BIF and should therefore make an explicit call with concrete arguments so as to get its concrete value. However, this will spawn a new process that will run $foo{:}bar(42)$ instead of interpreting it via our `eval` function. In order

to avoid this, we spawn a process starting the execution of a closure that will notify the appropriate monitor server and then call our `eval` with $[foo, bar, [42]]$ as arguments. In this way, we ensure that there are monitors to all the interpreter processes and there will be no ghost processes after the end of the concolic execution.

Creating anonymous functions. Closures in Core Erlang are created by a `c_fun` or with a call to `erlang:make_fun/3`. In Listing 4.2 we can see a simple example.

```
1  %% Erlang
2  foo(L) -> lists:map(fun foo:bar/1, L).
3
4  %% Core Erlang
5  {
6    {c_var, [], {foo, 1}},              %% Name and arity of the function
7    {c_fun, [],
8      [{c_var, [], 'L'}],               %% Parameter of the function
9      {c_let, [],                       %% Let definition
10       [{c_var, [], '_cor1'}],         %% Bound variable in let
11       {c_call, [],                    %% Call to erlang:make_fun/3
12         {c_literal, [], erlang},      %% to create a closure
13         {c_literal, [], make_fun},
14         [
15           {c_literal, [], 'foo'},     %% with [foo, bar, 1] as arguments
16           {c_literal, [], 'bar'},
17           {c_literal, [], 1}
18         ]
19       },
20       {c_call, [],                    %% Call to lists:map/2
21         {c_literal, [], lists},       %% to map the closure over the parameter
22         {c_literal, [], map},
23         [{c_var, [], '_cor1'}, {c_var, [], 'L'}]
24       }
25     }
26   }
27 }
```

Listing 4.2: Creation of simple closure

Once the execution reaches the `c_call` at line 11, the interpreter will see that there is a BIF call. If it makes a call with concrete arguments, the concrete result will be $erlang{:}make\_fun(foo, bar, 1)$. Problems arise once this closure is applied to an argument X.

- There will be a direct evaluation of $foo{:}bar/1$ instead of interpreting it with our `eval` function.
- X is a tuple of a concrete and a symbolic value but $foo{:}bar/1$ expects only the concrete one. This will lead to either an exception or a different result than expected.

Therefore, we create a closure that, once applied to its arguments, unwraps the concrete from the symbolic values and adds them to the respective environment. It then calls `eval` to interpret the body of the closure. In our example, it follows the workflow

- extracts the two values from X, let's say $X_{concrete}$ and $X_{symbolic}$

- creates the concrete environment $e_c = [X \rightarrow X_{concrete}]$
- creates the symbolic environment $e_s = [X \rightarrow X_{symbolic}]$
- retrieve t, the Core Erlang AST of `foo:bar/1`
- call *eval* t e_c e_s to continue the interpretation

Function application. In Core Erlang, a function application is either applying a closure to some arguments (using `c_apply` or `erlang:apply/2`) or using `erlang:apply/3` to emulate a call like $Module{:}Function(Arg_1, Arg_2, ..., Arg_N)$. In the first case, there is no problem since the closure was created by our interpreter and will correctly expect a tuple of two values for each argument (a concrete and a symbolic one). In the second case, we have a BIF that when called with concrete arguments will be directly evaluated and not interpreted. Let's see an example. In Listing 4.3, we use `erlang:apply/3` to evalute the result of applying `foo:bar/`N to As, a list of N terms.

```
%% Erlang
app(As) -> apply(foo,  bar,  As).

%% Core Erlang
{
  {c_var, [], {app, 1}},
  {c_fun, [],
    [{c_var, [], 'As'}],          %% As is the parameter of the function
    {c_call, [],                  %% Module-qualified call to
      {c_literal, [], 'erlang'},  %% erlang:apply/3
      {c_literal, [], 'apply'},
      [
        {c_literal, [], 'foo'},   %% with [foo, bar, As] as arguments
        {c_literal, [], 'bar'},
        {c_var, [], 'As'}
      ]
    }
  }
}
```

Listing 4.3: Using `c_apply`

Let's say that $foo{:}bar$ has an arity of 2 and we want to interpret $app([6, 7])$. First, we abstract As with a symbolic value, let's say $[x, y]$, and then create the concrete environment $e_c = [As \rightarrow [6, 7]]$ and the symbolic environment $e_s = [X \rightarrow [x, y]]$. The execution starts by calling *eval* t e_c e_s, where t is the AST of *app*.

The next step is to evaluate a call to `erlang:apply/3`. Since it is a BIF we should first evaluate it with concrete arguments. However, that would cause a direct evaluation of `foo:bar/2` rather than interpreting it. Therefore, instead of following our usual strategy for BIFs, we should call *eval* t' e'_c e'_s, where

- t' is the AST of `foo:bar/2`,
- e'_c is the concrete environment $[P_1 \rightarrow 6, P_2 \rightarrow 7]$
- e'_s is the symbolic environment $[P_1 \rightarrow x, P_2 \rightarrow y]$

assuming P_1 and P_2 are the names of the parameters of `foo:bar/2`. The interpretation can now continue normally.

Message passing. The messages that are being transmitted and received are a pair of a concrete and a symbolic value as well. In order for that to work properly, we want both the processes that send and receive this message to be interpreter processes. However, this is not always the case. There some processes the Erlang VM spawns at start-up. Perhaps, the most commonly used of them is the I/O server. The I/O server handles the I/O requests of the processes that belong to its group and performs the tasks on the I/O devices. These are requests are essentially messages. So, basically, when printing a message or reading some input, a message is sent to this process. In our evaluation, this will surely raise an exception due to incompatibility with our messages' format. The solution is to check before every send or receive if the recipient or the sender process is an interpreter process. If it is not, we just use the concrete value of the message. This information is provided by the monitor server, so we just need to make a synchronous request to it.

Symbolic values. A symbolic value can be either a concrete value or a symbolic variable. We use concrete values

- for BIFs that always evaluate to a specific value (e.g. `erlang:link/1`)
- for operations that our solver cannot handle yet. For example, our Z3 interface cannot model `binary:bin_to_list/1` yet. There is no reason for us to symbolically abstract the evaluation of this function and record it as we cannot use it in any way. Therefore we use the concrete value instead and continue the execution.

In any other case we use a symbolic variable. A symbolic variable is represented by an Erlang reference.

Keeping a trace file. Before expanding on the topic of symbolic evaluation and symbolic constraints, we should have a look on how we keep our trace files. The first approach was to create a specific process that would act as a trace server and receive all the information from the interpreter processes and record them in a file. However, this proved to be catastrophic as its mailbox would rapidly overflow and the process would die. Basically, the rate of receiving information was far greater than the rate of writing them to a file. Therefore, we decided that every interpreter process would be responsible to record its trace information in its own trace file. The rate of producing information slowed down as the processes have to pause execution and write the messages in a file instead of simply transmitting them. We must mention that the files are opened at raw mode because it allows faster access as no Erlang process is needed to handle the file. Our interpreter process can write binary information directly to it.

The data are written in the file after being encoded in a JSON format we defined for this use. This format preserves subterm sharing in order to compress the data.

JSON encoding of Erlang terms. In general we represent an Erlang term with

$$\{\text{"}t\text{"} : type, \text{"}v\text{"} : value, \text{"}d\text{"} : dictionary\_of\_shared\_subterms\}$$

As we see in Section 4.2 our current implementation supports reasoning over numbers, atoms, tuples, and lists. Therefore, we defined our JSON representation to support these types for now. Listing 4.4 shows the representation of the different types of terms.

- The value of an integer or float is the literal itself.
- The value of an atom is the list that corresponds to the textual representation of the atom.
- The value of a list is a JSON list that has as elements the elements of the erlang list.
- The value of a tuple is a JSON list that has as elements the elements of the tuple.

Let's note that the "*d*" attribute is optional and it appears only if the term has subterm sharing. Of course it has meaning to exist only in lists and tuples.

```
JsonTerm :: {''t'' : ''Int'', ''v'' : int-value}                                (integer)
          | {''t'' : ''Real'', ''v'' : float-value}                             (float)
          | {''t'' : ''Atom'', ''v'' : [int-value+]}                            (atom)
          | {''t'' : ''List'', ''v'' : [JsonTerm*], ''d'' : shared-subterms}    (list)
          | {''t'' : ''Tuple'', ''v'' : [JsonTerm*], ''d'' : shared-subterms}   (tuple)
```

Listing 4.4: JSON Encoding of Erlang Terms

The dictionary of shared subterms is a JSON object that has as many attributes as the shared subterms. Eash subterm is identified by a unique string (name of the attribute) and its value is the JSON representation of the term. We can use the object

$$\{"l" : subterm\_identifier\}$$

instead of the actual value of the subterm where needed. In Listing 4.5 we can see some examples of this JSON representation. As you can see in example 3, the term LLL has no sharing but we treated it as if it was the term $[LL, LL]$. Our algorithm actually performs subexpression elimination so we can achieve better compression of the term's size.

The algorithm that performs the encoding is based on the current implementation of `erts_debug:size/1` and its complexity is $O(nlogn)$. It performs two passes of the term.

- On the first pass it finds the common subterms. This done by putting every unique subterm on a `gb_tree` ($O(n)$ time to traverse the term and $logn$ amortized time for the lookup and insert operations on the `gb_tree`).
- On the second pass it creates the JSON string. It traverses again the term and replaces common subterms with their unique identifier. It also traverses and encodes the `gb_tree` into the dictionary attribute of the JSON representation. ($O(n)$ time to traverse the term, $O(n)$ time to traverse the `gb_tree` and $O(logn)$ amortized time for the lookup operations on the `gb_tree`).

JSON encoding of Erlang types. Another piece of information we want to record in our trace file is the type of the initial function's arguments as defined in its spec (if one is provided). As initial function we mean the function that was the entry point of the program. This type information is stored in the Core Erlang AST as we described in Section 2.4.3. Once we extract it, we encode it to JSON and write it in our trace file. In Listing 4.6 we see how each type is encoded.

```
%% 1) A simple 2-element list
Term:
  L = [1, 2]
JSON:
  {
    ''t'' : ''List'',
    ''v'' : [
      {''t'':''Int'', ''v'':1},
      {''t'':''Int'', ''v'':2}
    ]
  }

%% 2) Let's throw in some sharing
Term:
  LL = [L, L]
JSON:
  {
    ''t'': ''List'',
    ''v'': [
      {''l'': ''0.0.0.42''},
      {''l'': ''0.0.0.42''}
    ],
    ''d'': {
      ''0.0.0.42'': {
        ''t'': ''List'',
        ''v'': [
          {''t'':''Int'', ''v'':1},
          {''t'':''Int'', ''v'':2}
        ]
      }
    }
  }

%% 3) More than sharing
Term:
  LLL = {LL, [[1, 2], [1, 2]]}
JSON:
  {
    ''t'': ''Tuple'',
    ''v'': [{''l'': ''0.0.0.59''}, {''l'': ''0.0.0.59''}],
    ''d'': {
      ''0.0.0.59'': {
        ''t'': ''List'',
        ''v'': [{''l'': ''0.0.0.58''}, {''l'': ''0.0.0.58''}]
      },
      ''0.0.0.58'': {
        ''t'': ''List'',
        ''v'': [
          {''t'':''Int'', ''v'':1},
          {''t'':''Int'', ''v'':2}
        ]
      }
    }
  }
```

Listing 4.5: Examples of JSON encoding of Erlang Terms

```
JSON_Type :: {''t'' : ''literal'', ''i'' : JSON_Term}            (literal)
           | {''t'' : ''any''}                                   (any() | term())
           | {''t'' : ''atom''}                                  (atom())
           | {''t'' : ''boolean''}                               (boolean())
           | {''t'' : ''byte''}                                  (byte())
           | {''t'' : ''char''}                                  (char())
           | {''t'' : ''float''}                                 (float())
           | {''t'' : ''integer'', ''i'' : itype}                (integer())
           | {''t'' : ''list'', ''i'' : Type}                    (list())
           | {''t'' : ''nelist'', ''i'' : JSON_Type}             (non_empty_list())
           | {''t'' : ''number''}                                (number())
           | {''t'' : ''range'', ''a'' : [JSON_Term1, JSON_Tem2]} (range())
           | {''t'' : ''string''}                                (string())
           | {''t'' : ''nestring''}                              (non_empty_string())
           | {''t'' : ''timeout''}                               (timeout())
           | {''t'' : ''tuple'', ''a'' : [JSON_Type*]}           (tuple())
           | {''t'' : ''union'', ''a'' : [JSON_Type+]}           (union())

itype :: ''any'' | ''pos'' | ''neg'' | ''non_neg''
```

Listing 4.6: JSON encoding of Erlang Types

Symbolic abstraction of operations. When we want to symbolically evaluate an operation *op* with arguments *es*, we create a fresh symbolic variable *s* that represents the symbolic result of applying *op* to *es*. We record this equivalence in our trace file and continue the execution of the program. With this technique we alleviate the burden of keeping complex symbolic expressions in our environment. There is an increased overhead of writing more often to our trace files but the terms we record are simpler with much less sharing involved.

In Listing 4.7 we see a simple function that calculates the Manhattan distance of two points on a 2-D space based on their coordinates.

```
1 -spec manhattan({number(), number()}, {number(), number()}) -> number().
2 manhattan(P1, P2) ->
3   X1 = element(1, P1),
4   Y1 = element(2, P1),
5   X2 = element(1, P2),
6   Y2 = element(2, P2),
7   DX = abs(X1 - X2),
8   DY = abs(Y1 - Y2),
9   DX + DY.
```

Listing 4.7: Function that calculates the Manhattan distance of 2 points

Let's say that we want to evaluate $manhattan(\{2,3\},\{1,5\})$. In Table 4.1 we can see how we symbolically abstract the operations during its evaluation. We also show the data that is recorded in the trace file after every operation in a readable form.

No	Operation	Concrete Environment	Symbolic Environment	Trace File
		$e_c = \{P1 \rightarrow \{2,3\}, P2 \rightarrow \{1,5\}\}$	$e_s = \{P1 \rightarrow s0, P2 \rightarrow s1\}$	
1	$X1 = element(1, P1)$	$e_c \cup \{X1 \rightarrow 2\}$	$e_s \cup \{X1 \rightarrow s2\}$	$s2$ equals $element(1, s0)$
2	$Y1 = element(2, P1)$	$e_c \cup \{Y1 \rightarrow 3\}$	$e_s \cup \{Y1 \rightarrow s3\}$	$s3$ equals $element(2, s0)$
3	$X2 = element(1, P2)$	$e_c \cup \{X2 \rightarrow 1\}$	$e_s \cup \{X2 \rightarrow s4\}$	$s4$ equals $element(1, s1)$
4	$Y2 = element(2, P2)$	$e_c \cup \{Y2 \rightarrow 5\}$	$e_s \cup \{Y2 \rightarrow s5\}$	$s5$ equals $element(2, s1)$
5	$DX = abs(X1 - X2)$	$e_c \cup \{DX \rightarrow 1\}$	$e_s \cup \{DX \rightarrow s7\}$	$s6$ equals $s2 - s4$ $s7$ equals $abs(s6)$
6	$DY = abs(Y1 - Y2)$	$e_c \cup \{DY \rightarrow 2\}$	$e_s \cup \{DY \rightarrow s9\}$	$s8$ equals $s3 - s5$ $s9$ equals $abs(s8)$
7	$DX + DY$	$e_c \cup \{V0 \rightarrow 3\}$	$e_s \cup \{V0 \rightarrow s10\}$	$s10$ equals $s7 + s9$

Table 4.1: Evaluation of $manhattan(\{2,3\}, \{1,5\})$

The evaluation starts with the environments only having the mappings of the function's parameters. At each operation, a fresh symbolic variable ($s2$, $s3$, ...) is created to represent the symbolic result of the operation. This equivalence is also recorded in the trace file. Note that in the fifth and sixth commands there are more variables generated since it is a complex operation. Core Erlang breaks down complex statements and creates intermediate variables as is shown in operation 7. The final result of the evaluation is the tuple $\{3, s10\}$.

Symbolic constraints. The only control statements in Core Erlang are the `c_case` and the `c_receive`. In both cases they actual matching primitives are the same. Each result of a matching creates a constraint that we record. We can distinguish four types of constraints (we omit the bitstring matchings as we do not currently support bitstrings and binaries) which we can see in Table 4.2.

Type	Possible Constraints
Result of a Guard	Guard evaluates to true or Guard evaluates to false
Matching Literals	Literals are equal or Literals are not equal
Matching Tuples	Is a tuple of n elements, Is not a tuple of n elements or Is not a tuple
Matching Lists	Is a non empty list, Is an empty list or Is not a list

Table 4.2: Constraints in Core Erlang

At each constraint we encounter, we record

- the symbolic values of the variables that decide the outcome of the matching
- the outcome of the constraint

Let's see an example. In Listing 4.8, we can see a simple tail recursive function that calculates the factorial of an integer.

```
%% Erlang
fact(N) -> fact(N, 1).

fact(1, Acc) -> 1;
fact(N, Acc) -> fact(N - 1, N * Acc).

%% Core Erlang
{
  {c_var, [], {fact, 1}},                              %% fact/1
  {c_fun, [],
    [{c_var, [], 'N'}],                                %% Parameter: N
    {c_apply, [],                                      %% Call fact(N, 1)
      {c_var, [], {fact, 2}},
      [{c_var, [], 'N'}, {c_literal, [], 1}]
    }
  }
}

{
  {c_var, [], {fact, 2}},                              %% fact/2
  {c_fun, [],
    [{c_var, [], 'v0'}, {c_var, [], 'v1'}],            %% Parameters: v0, v1
    {c_case, [],                                       %% case statement
      {c_values, [],
        [{c_var, [], 'v0'}, {c_var, [], 'v1'}]         %% Switch expression
      },                                               %% is <v0, v1>
      [
        {c_clause, [],                                 %% 1st clause
          [{c_literal, [], 1}, {c_var, [], 'Acc'}],    %% Match <1, Acc>
          {c_literal, [], true},                       %% Guard is 'true'
          {c_var, [], 'Acc'}                           %% return Acc
        },
        {c_clause, [],                                 %% 2nd clause
          [{c_var, [], 'N'}, {c_var, [], 'Acc'}],      %% Match <N, Acc>
          {c_literal, [], true},                       %% Guard is 'true'
          {c_let, [],
            [{c_var, [], 'v2'}],                       %% Bind v2
            {c_call, [],                               %% to N - 1
              {c_literal, [], 'erlang'},
              {c_literal, [], '-'},
              [{c_var, [], 'N'}, {c_literal, [], 1}]},
            {c_let, [],
              [{c_var, [], 'v3'}],                     %% Bind v3
              {c_call, [],                             %% to N * Acc
                {c_literal, [], 'erlang'},
                {c_literal, [], '*'},
                [{c_var, [], 'N'}, {c_var, [], 'Acc'}]},
              {c_apply, [],                            %% return fact(v2, v3)
                {c_var, [], {fact, 2}},
                [{c_var, [], 'v2'}, {c_var, [], 'v3'}]
}}}}]}}}
```

Listing 4.8: Factorial function

In Table 4.3, we see how we evaluate $fact(2)$ and what is the trace that we record.

No	Operation	Concrete Environment	Symbolic Environment	Trace File
1	Call $fact(2)$	$e_{c1} = \{N \to 2\}$	$e_{s1} = \{N \to s0\}$	
2	Call $fact(2,1)$	$e_{c2} = \{v0 \to 2,$ $v1 \to 1\}$	$e_{s2} = \{v0 \to s0,$ $v1 \to s1\}$	
3	Check 1st clause: $v0$ does not match 1			$s0 \neq 1$
4	Check 2nd clause: $v0$ matches N $v1$ matches Acc	$e_{c2} \cup \{N \to 2\}$ $e_{c2} \cup \{Acc \to 1\}$	$e_{s2} \cup \{N \to s0\}$ $e_{s2} \cup \{Acc \to s1\}$	Guard $true$ is $true$
5	Bind $v2$ to $N - 1$	$e_{c2} \cup \{v2 \to 1\}$	$e_{s2} \cup \{v2 \to s2\}$	$s2$ equals $s0 - 1$
6	Bind $v3$ to $N * Acc$	$e_{c2} \cup \{v3 \to 2\}$	$e_{s2} \cup \{v3 \to s3\}$	$s3$ equals $s0 * s1$
7	Call $fact(v2, v3)$	$e_{c3} = \{v0 \to 1,$ $v1 \to 2\}$	$e_{s3} = \{v0 \to s2,$ $v1 \to s3\}$	
8	Check 1st clause: $v0$ matches 1 $v1$ matches Acc	$e_{c3} \cup \{Acc \to 2\}$	$e_{s3} \cup \{Acc \to s3\}$	$s2 = 1$ Guard $true$ is $true$
9	Return Acc			

Table 4.3: Evaluation of $fact(2)$

The constraints we record are at operations 3, 4 and 8. However, we can omit the guard constraints since the expressions are trivial and do not involve any symbolic variables. The concolic result of the evaluation is the tuple $Acc \to \{2, s3\}$.

4.2 Z3 Interface

Once the instrumentation algorithm selects the trace of a specific execution to be solved in order to produce a new testcase, it invokes the component that interfaces with the Z3 solver. This component is responsible for

1. transmitting the trace information to the Python interface of Z3,
2. properly encoding the information for the solver,
3. invoking Z3 and returning the result.

First, let's see how we encode the trace information.

4.2.1 Implementing the Erlang Type System

As we stated in Section 3.1.2, Z3 supports algebraic datatypes. Defining most of Erlang types is therefore straightforward. However, problems arise when one tries to define

bitstrings and binaries. Z3 supports bit-vectors, that are designed to be used as the representation of binary types, but does not allow bit-vectors with variable lengths. Therefore, we cannot define datatypes that are parametrized over the bit-vector length. For this reason, we decided to simply do not support reasoning over bitstrings and binaries until we find a proper way of circumventing this limitation.

In the current implementation of the type system, we consider an Erlang term to be either:

- an integer number,
- a real number,
- an atom,
- a proper list or
- a tuple.

The rest of the datatypes are either too specific to Erlang to be encoded for Z3 (like PIDs and References) or it is not possible for a current solver to reason over them (like funs).

In Listing 4.9, we can see the definition of the Erlang Type System in Z3. Integers and floats are built-in types, lists and tuples are defined as a connected list of terms and atoms are essentially a list of integers that corresponds to the result of `atom_to_list/1`. A term is a union of the above.

```
# Declare datatypes
Term = Datatype('Term')
TermList = Datatype('TermList')
IntList = Datatype('IntList')

# Define an Erlang term
#
# Term :: int(Int)       # integer
#       | real(Real)     # real
#       | lst(TermList)  # list
#       | tpl(TermList)  # tuple
#       | atm(IntList)   # atom
Term.declare('int', ('ival', IntSort()))
Term.declare('real', ('rval', RealSort()))
Term.declare('lst', ('lval', TermList))
Term.declare('tpl', ('tval', TermList))
Term.declare('atm', ('aval', IntList))

# Define helper datatypes
TermList.declare('nil')
TermList.declare('cons', ('hd', Term), ('tl', TermList))
IntList.declare('anil')
IntList.declare('acons', ('ahd', IntSort()), ('atl', IntList))
```

Listing 4.9: Defining the Erlang Type System in Z3

In Table 4.4 we can see how some simple Erlang terms are encoded in this format.

Erlang Term	Encoded Representation in Z3
42	$int(42)$
[3.14]	$lst(cons(real(157/50), nil))$
$\{6, 7\}$	$tpl(cons(int(6), cons(int(7), nil)))$
ok	$atm(acons(int(111), acons(int(107), anil)))$

Table 4.4: Examples of encoding Erlang terms in Z3

4.2.2 Emulating the Semantics of Erlang Built-in Functions

The largest part of the trace information is the symbolic abstraction of the Erlang BIFs. Each entry refers to one call of a BIF and correlates the symbolic arguments with the symbolic result. For instance, the entry $s2 = s0 + s1$ denotes that $s2$ is the result of adding $s1$ to $s0$ in a symbolic manner.

In order to import this correlation into the solver, we must translate it into a set of axioms that can be asserted to the universe. Most of the translations are straightforward. Let's see some examples.

Example 1: `erlang:'+'/2`

Consider the abstraction of `erlang:'+'/2` that we mentioned, namely $s2 = s0 + s1$. First of all, $s0$, $s1$ and $s2$ are generally terms. However, the operator + restricts them to integers and reals thus allowing four alternatives.

1. $s0$, $s1$ and $s2$ are all integers. The generated axioms are

$$Ax1 = is\_int(s0) \wedge is\_int(s1) \wedge is\_int(s2) \wedge ival(s0) + ival(s1) = ival(s2)$$

2. $s0$, $s1$ and $s2$ are all real numbers. The generated axioms are

$$Ax2 = is\_real(s0) \wedge is\_real(s1) \wedge is\_real(s2) \wedge rval(s0) + rval(s1) = rval(s2)$$

3. $s0$ is integer, $s1$ and $s2$ are real numbers. The generated axioms are

$$Ax3 = is\_integer(s0) \wedge is\_real(s1) \wedge is\_real(s2) \wedge ival(s0) + rval(s1) = rval(s2)$$

4. $s1$ is integer, $s0$ and $s2$ are real numbers. The generated axioms are

$$Ax4 = is\_real(s0) \wedge is\_int(s1) \wedge is\_real(s2) \wedge rval(s0) + ival(s1) = rval(s2)$$

Therefore, the + operation can be described with the axiom A, where

$$A = Ax1 \vee Ax2 \vee Ax3 \vee Ax4$$

Example 2: `erlang:hd/1`

Another simple BIF to encode is the `erlang:hd/1`. Our trace file will have an entry like $s1 = hd(s0)$. This means that

- $s0$ is a list ($is\_lst(s0)$),
- is non-empty ($is\_cons(lval(s0))$) and
- $s1$ is its head ($hd(lval(s0)) = s1$).

Putting all these together, the axiom A for this operation is

$$A = is\_lst(s0) \wedge is\_cons(lval(s0))) \wedge hd(lval(s0)) = s1$$

Example 3: `erlang:is_tuple/1`

To encode BIFs that return boolean values we take advantage of the built-in `If` expression. Consider that we have recorded the operation $s1 = is\_tuple(s0)$. If $s0$ is a tuple, then $s1$ will evaluate to the atom $true$. Whereas if $s0$ is not a tuple, then $s1$ will evaluate to the atom $false$. We can encode this behaviour with the `If` expression and create the axiom A.

$$A = If(is\_tpl(s0), s1 = atom\_true, s1 = atom\_false)$$

Example 4: `erlang:length/1`

As we saw in Section 3.1.4, Z3 does not allow the definition of recursive functions. If we use quantifiers to describe the recursive behaviour, then the solver will most likely be unable to to solve the set of constraints that we will provide. In order to overcome this limitation, we applied a heuristic that limits the depth of the recursion allowing us to unroll the recursive definition.

Let's see an example of how it works. Consider that we have the entry $s1 = length(s0)$. The function $length$ is recursive over the list. We select an integer n and say that for lists that consist of up to n elements, we know and can define the value of $length$. For lists that consist of more than n elements, we only know that the value of $length$ is greater than n but not what exactly. If we select $n = 2$, the entry $s1 = length(s0)$ is represented by the axiom shown in Listing 4.10.

Although this technique allows us to reason over recursive operations, is only an approximation to the actual semantics. If choose a small n, we will not get accurate results. Whereas, if we choose a large n, we will be more accurate but the axioms produced will be quite large and the solver may still be unable to produce an answer. Due to the lack of statistical data, we select $n = 100$ as an acceptable limit that we felt would be reasonable for most real-world programs.

```
And(
  is_lst(s0),                                  # s0 is a list
  is_int(s1),                                  # AND s1 is an integer
  If(                                          # AND
    is_nil( lval(s0) ),                        # IF s0 == []
    s1 == 0,                                   # THEN s1 == 0
    And(
      is_cons( lval(s0) ),                     # ELSE s0 is a nonempty list
      If(                                      # AND
        is_nil( tl(lval(s0)) ),                # IF tl(s0) == []
        s1 == 1,                               # THEN s1 == 1
        And(
          is_cons( tl(lval(s0)) ),             # ELSE tl(s0) is a nonempty list
          If(                                  # AND
            is_nil( tl(tl(lval(s0))) ),        # IF tl(tl(s0)) == []
            s1 == 2,                           # THEN s1 == 2
            s1 > 2,                            # ELSE s1 > 2
))))))
```

Listing 4.10: Axiom that represents `erlang:length/1` with approximation $n = 2$

4.2.3 Encoding the Symbolic Constraints

The rest of the trace information contains the symbolic constraints' entries. We must generate the respective axioms of each constraint and add them to our universe. However, the last constraint will be added with its reversed semantics as we want our solver to generate a testcase where this constraint will be invalidated. In Table 4.5, we can see the necessary axioms for each constraint in its normal and reversed semantics.

Constraint	Axioms	
	Normal Semantics	Reversed Semantics
$s_{Guard} = true$	$s_{Guard} = atom\_true$	$s_{Guard} = atom\_false$
$s_{Guard} = false$	$s_{Guard} = atom\_false$	$s_{Guard} = atom\_true$
$s1$ equals $s0$	$s1 = s0$	$s1 \neq s0$
$s1$ not equals $s0$	$s1 \neq s0$	$s1 = s0$
s : tuple of n elements	$is\_tpl(s) \wedge$ $length(tval(s)) = n$	$\neg is\_tpl(s) \vee$ $length(tval(s)) \neq n$
s : tuple of not n elements	$is\_tpl(s) \wedge$ $length(tval(s)) \neq n$	$\neg is\_tpl(s) \vee$ $length(tval(s)) = n$
s : not a tuple	$\neg is\_tpl(s)$	$is\_tpl(s)$
s : non empty list	$is\_lst(s) \wedge$ $is\_cons(lval(s))$	$\neg is\_lst(s) \vee$ $is\_nil(lval(s))$
s : empty list	$is\_lst(s) \wedge is\_nil(lval(s))$	$\neg is\_lst(s) \vee$ $is\_cons(lval(s))$
s : not a list	$\neg is\_lst(s)$	$is\_lst(s)$

Table 4.5: The axioms of each symbolic constraint

4.2.4 Interfacing Erlang with z3Py

The other major function of this component is to send the trace information to the Python interface of Z3 and invoke the solver. We achieve this by spawning a dedicated `gen_fsm` process to act as a middleware tier between Erlang and Python whenever we want to solve a set of symbolic constraints. As we see in Figure 4.2, multiple `gen_fsm` processes can exist simultaneously solving different sets of constraints.

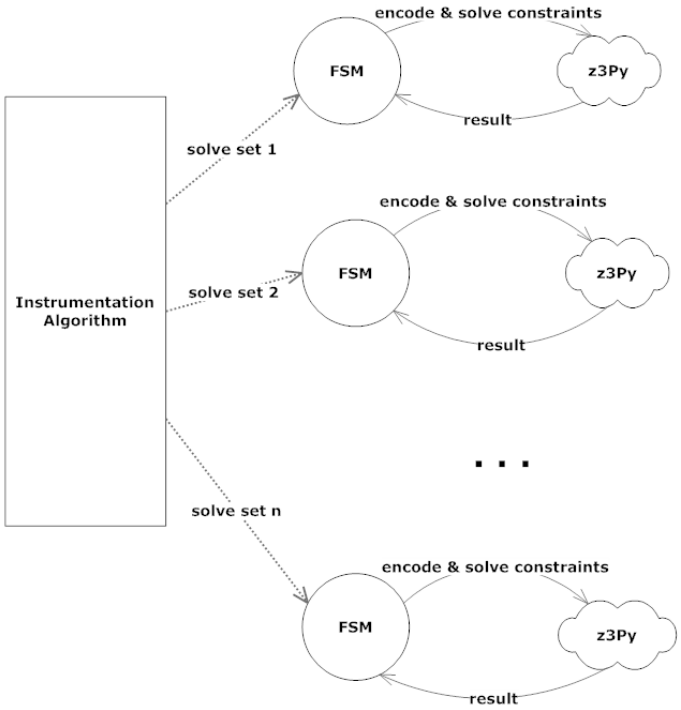

Figure 4.2: Overview of the interface component

The communication between the fsm process and the python interface is done via Erlang ports. The fsm opens a Erlang port that

1. creates a Python process,
2. create a pipe between the Erlang VM and the Python process and
3. attach the pipe to the stdin and stdout on the Python end.

We send binary data through the port in the format shown in Figure 4.3.

Figure 4.3: Format of the binary data sent throw the Erlang - Python pipe

In order to minimize the data transactions, we do not send the JSON encoded trace data through the port but rather the start and end position of the data in the file that they are recorded. The python process then opens the trace file and loads the appropriate data.

`gen_fsm` States

In Figure 4.4, we can see the different states of the fsm and the transitions between them. Briefly, the fsm starts at the `idle` state. It opens the port to start the python process and transitions to the `waiting` state. While it sends the locations of the trace data through the port it remains on this state. Once all the trace data are sent, it queries the solver for the satisfiability of the transmitted data and moves to the `solving` state. They are two possible scenarios from here.

- *The solver cannot solve the constraints.* In this scenario, the solver responds with either *unsat*, *timeout* or *unknown*. In all three cases, we consider the model to be unsatisfiable. The fsm reaches the final state `finished` and exits.
- *The solver can solve the constraints.* In this scenario, the solver responds with *sat*. The fsm transitions to the `solved` state. Then it requests the instance of the model and moves to the `generating_model` state. Once it receives the reply, it reaches the final state `finished` and returns the model's representation to the instrumentation algorithm.

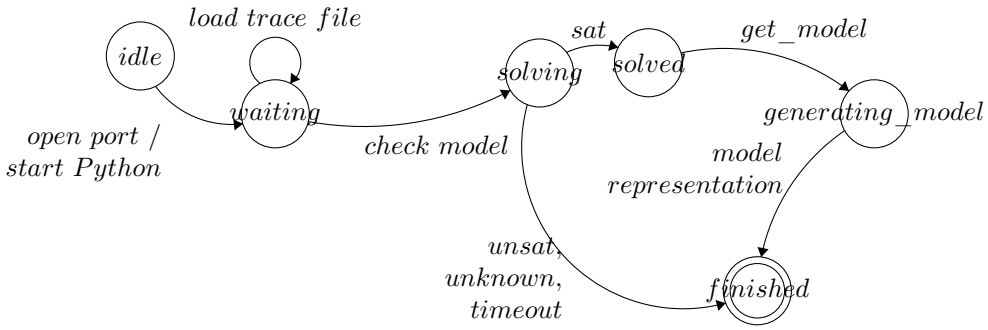

Figure 4.4: `gen_fsm` state diagram

Let's see some sequence diagrams for these two possible scenarios and explain the processes in detail.

Scenario I: Z3 Can Solve the Set of Symbolic Costraints

The workflow in this scenario is the following:

1. The instrumentation process spawns a new `gen_fsm` that is initialized to the `idle` state. The fsm remains idle until it receives the exact command that it should run through the port and spawn the python process.
2. Once it receives that command, it opens a new port and starts the python process. It responds *ok* and transitions to the `waiting` state.

3. The set of constraints that must be loaded to the solver may be dispersed in multiple trace files (for example if the program we are testing is concurrent). For each of the trace files, the fsm receives a message to load specific chunks of the file's data to the solver. It sends this information through the port and the python interface on the other side opens the files, properly encodes the data and asserts the axioms to the solver.

4. After all the trace files have been loaded, the fsm receives the message to start solving these constraints. It propagates this call to the python process which in turn invokes Z3's *check* method on the created model. The fsm transitions to the `solving` phase and waits for the result.

5. In this scenario, the model is satisfiable and Z3 responds with *sat*. The fsm receives the response, transitions to the *solved* state and notifies the instrumentation algorithm for the successful result.

6. The fsm receives the command to retrieve the resulting instance of the model. The task is delegated to the python process and the fsm transitions to the `generating_model` state.

7. The model that is generated by Z3 is retrieved and encoded in the JSON format and transmitted via the pipe to the fsm. The fsm decodes the response and returns it to the instrumentation algorithm and transitions to the final `finished` state.

8. Once the result is successfully received, the fsm is stopped and the python process terminated.

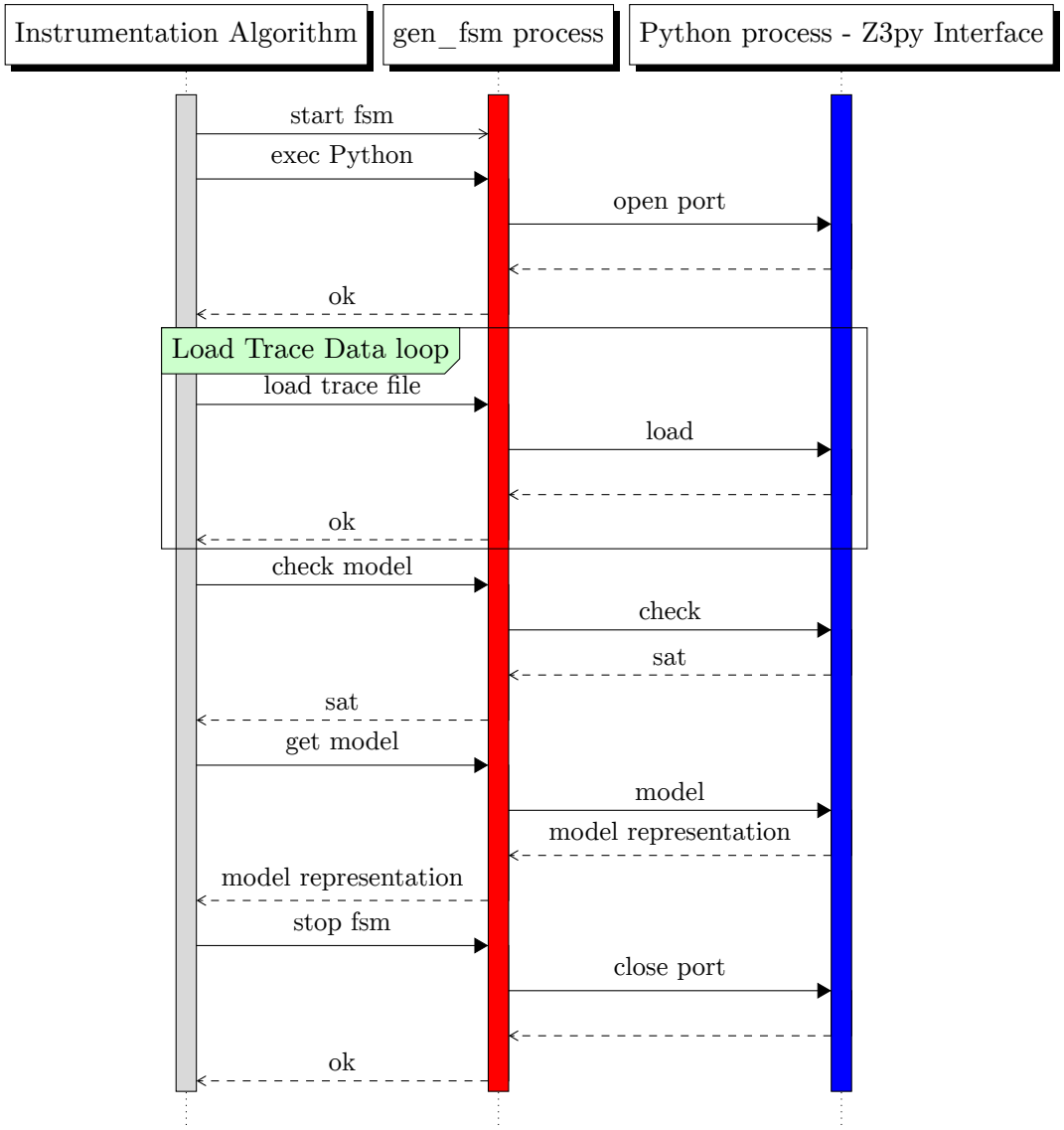

Figure 4.5: Sequence diagram for the scenario where Z3 can solve the constraints

Scenario II: Z3 Cannot Solve the Set of Symbolic Constraints

The workflow in this scenario diverges from the previous one in step 5.

6. In this scenario, the model is may be unsatisfiable and Z3 responds with *unsat*, Z3 may be unable to provide an answer and respond with *unknown* or *timeout*. In all these cases, we do not amend the constraints and resend them for solving but rather concede that the program state we want to reach is unreachable. The fsm notifies the instrumentation algorithm for the unsuccessful result and transitions to the final $finished$ state.

7. As before, once the result is successfully received, the fsm is stopped and the python process terminated.

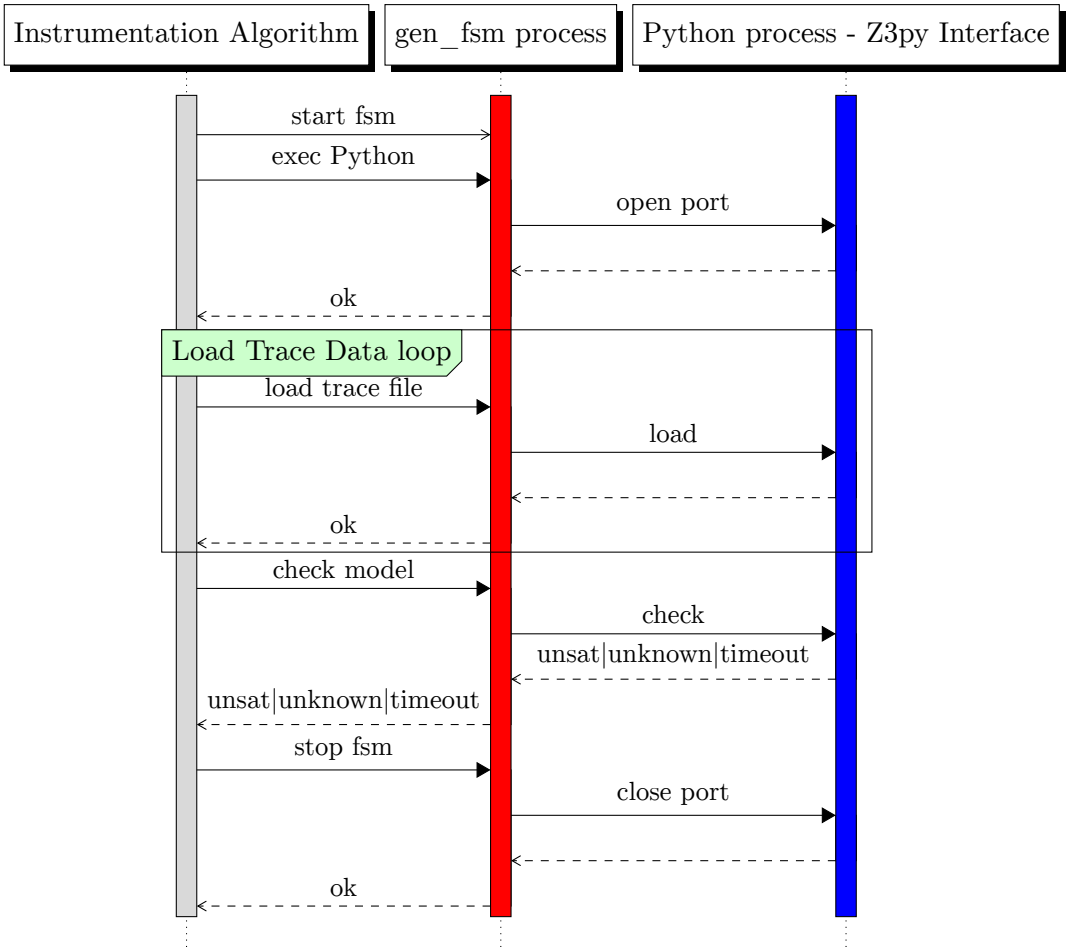

Figure 4.6: Sequence diagram for the scenario where Z3 cannot solve the constraints

4.3 Instrumentation Algorithm

In order to explore all the feasible execution paths, we do a *Bread-First Search* in the computation tree. The goal is to visit as many different execution paths as fast as possible. There are exponentially many paths to explore thus an exhaustive search is infeasible for non-trivial programs. The BFS algorithm enables us to enumerate all the possible paths that differ on the i first conditions and then continue with enumerating all the possible paths that differ on the $i + 1$ first conditions. In this way, we can set an upper limit to our search depending on our computational resources.

4.3.1 Concurrent and Distributed Programs

In sequential programs, the program's path predicate has a linear representation as it consists of the symbolic constraints of only one process. Should we want to visit a path that diverges on the i-th constraint, we only need to pass the $i - 1$ constraints and the logical negation of the i-th constraint to the SMT solver and request a model that satisfies them.

However, in concurrent and distributed programs, the program's path predicate has a tree representation as it consists of the linear path predicates of each process. Now, it is not so obvious which constraints to pass to the solver in order to generate a variant of the input that will explore a specific path. Let's assume that we want to explore a path that will diverge on the i-th constraint of process j. Of course, we still need to pass the $i-1$ constraints and the logical negation of the i-th constraint of process j to the solver. However, we need to find which process spawned j and also pass the constraints of that process till the moment it spawned j. The same thing applies to all the processes that sent a message to j and wrote in the shared memory.

In order to serialize the path predicate, we need to perform an offline trace analysis. We did not implement it for this thesis and this is the reason why CutEr currently only supports sequential Erlang programs.

4.3.2 Presentation of the Algorithm

Let's assume that we are given an Erlang program P, a seed input x and the limit l of the breadth first search. Our goal is to return a list R of all the erroneous executions and their respective inputs.

1. The first step has to do with initializations. We perform the concolic execution of program P with x as input. If the the result is erroneous then we add it to R. The execution also yields the path predicate p_1. We add the pair $\{p_1, 1\}$ to our empty queue Q. This means that at a later step we will try to generate a new input that will lead to an execution that will diverge from p_1 on the 1st constraint.

2. We can now move to the main loop of the algorithm, which executes for as long as there are path predicates in queue Q.

3. We take the first pair from Q, let's say $\{p, i\}$. This means that will try to diverge from p on the i-th constraint. We pass the $i-1$ constraints and the negated i-th constraint to our SMT solver and check for the satisfiability of the model.

4. If the model is satisfiable, we recover the interpretation y that satisfies it. We then perform the concolic execution of P with y as input. This will yield the result r and the path predicate p'. If r is erroneous, we add it to R. Since p and p' cover all the possible outcomes of the i-th constraint, we add the pairs $\{p', i+1\}$ and $\{p, i+1\}$ to Q if $i+1$ is smaller than l.

5. If the model is unsatisfiable or the solver cannot respond, we simply add the pair $\{p, i+1\}$ to Q.

6. When Q becomes empty, we report the list R of the erroneous executions.

We can see the pseudocode of the algorithm in Figure 4.7.

Require: An Erlang program P, a seed input x and the limit l of the breadth first search.

1: $R \leftarrow []$ {list of the program errors we found}
2: $\{r_1, p_1\} \leftarrow \text{run}(P, x)$ {run P with input x, return the result and the path constraint}
3: $Q \leftarrow [\{p_1, 1\}]$ {queue of path predicates to negate}
4: **if** $r_1 \in$ error **then**
5: $R \leftarrow R \vee \{r_1, x\}$
6: **end if**
7: **while** $Q \neq \emptyset$ **do**
8: $\{p, i\} \leftarrow \text{dequeue}(Q)$
9: $L \leftarrow p[1] \wedge p[2] \wedge ... \wedge p[i-1] \wedge \neg p[i]$
10: **if** z3_check$(L) =$ sat **then**
11: $y \leftarrow$ z3_model(L)
12: $\{r, p'\} \leftarrow \text{run}(P, y)$
13: **if** length$(p') \geq i + 1$ **then**
14: enqueue$(Q, \{p', i+1\})$
15: **end if**
16: **if** $r \in$ error **then**
17: $R \leftarrow R \vee \{r, x\}$
18: **end if**
19: **end if**
20: **if** length$(p) \geq i + 1$ **then**
21: enqueue$(Q, \{p, i+1\})$
22: **end if**
23: **end while**

Figure 4.7: Implementation of the breadth-first search in the computation tree.

Chapter 5

Examples

In this chapter, we will present three examples of using CutEr to test sequential Erlang programs.

5.1 A Toy Example

Let's consider the example we described in Section 3.1.1. It consists of a very simple function which is ideal for showing how CutEr tests programs.

We can see the resulting Core Erlang code in Listing 5.1.

```
1  foo(X, Y) ->
2    let Z = 'erlang':'*'(2, Y)
3    in let K =
4      case <> of
5        <> when call 'erlang':'=:='(X, 100000) -> call 'erlang':'<'(X, Z)
6        <> when 'true' -> 'false'
7      end
8    in case K of
9      'false' when 'true' -> 'ok'
10     'true' when 'true' -> call 'erlang':'error'('assertion')
11   end
```

Listing 5.1: Core Erlang code of the example in Section 3.1.1

This function has three possible execution paths so we should need exactly three executions to explore all of them. Let's say that our seed input is $X \rightarrow 1$ and $Y \rightarrow 1$.

The first execution with the seed input will produce the concrete result *ok*. In addition, there will be only one non-trivial constraint (a constraint that involves at least one symbolic variable). That constraint will be the guard expression at line 5 which compares X to 100000. In this case, the condition will evaluate to $false$. This is the output we got from CutEr for this first execution as we can see in Listing 5.2. The actual symbolic constraint that is generated is $(X =:= 100000) = false$.

CutEr will try to negate this one constraint and will produce a testcase where $X \rightarrow 100000$. In this case, it generated the input $X \rightarrow 100000$ and $Y \rightarrow 19$. This input will produce the

```
1  Input: 1, 1
2   Result: ok
3   Path Vertex: ''F''
4  -------------------------------------------------
5  Input: 100000, 19
6   Result: ok
7   Path Vertex: ''TT''
8  -------------------------------------------------
9  Input: 100000, 50001
10  Runtime Error: assertion
11  Path Vertex: ''TFT''
```

Listing 5.2: CutEr output when testing they toy function

concrete result *ok* and will generate two non-trivial constraints. The first one is again the guard expression in line 5 that will generate the constraint $(X =:= 100000) = true$. The second one is the first clause of the case statement in line 9. This will try to match the result of the comparison $X < Z$ to $false$. Z is $2 * Y \rightarrow 38 < 100000$ so the comparison will return $false$ and the match will occur resulting in a $true$ condition. The generated constraint will be $(X < 2 * Y) = false$.

We now have two symbolic constraints and the CutEr will try to negate the second one for the third execution. The solver will receive the model

$$(X =:= 100000) = true \wedge \neg(X < 2 * Y) = false$$

and will produce a testcase where $X \rightarrow 100000$ and $Y > 50000$. In this case it generates the input $X \rightarrow 100000$ and $Y \rightarrow 50001$. In this execution there will be three non-trivial constraints. The first one is again the guard expression in line 5 that will generate the constraint $(X =:= 100000) = true$. The second one will be the case clause in line 8 that will no longer match and will generate the constraint $(X < 2 * Y) = true$. The third one will be the case clause in line 10 that will now match and produce the constraint $(X < 2 * Y) = true$.

CutEr will try to perform a fourth execution by negating the third constraint. It will pass the model

$$(X =:= 100000) = true \wedge (X < 2 * Y) = false \wedge \neg(X < 2 * Y) = true$$

to the solver which is unsatisfiable. CutEr will then deduce that it has explored all the execution paths and will terminate.

5.2 A Depth-First Search Example

In this example we will see how CutEr can generate inputs that match custom data types (Listing 5.3).

Let's assume a NxN map and a robot that wants to move from point *Start* to point *End*. The robot can move up, right, down and left and to a point that is inside the map bounds

```erlang
%% Macro definition
-define(MAP_SIZE, 8).

%% Type definitions
-type map_size() :: 1..?MAP_SIZE.
-type coordinate() :: 1..?MAP_SIZE.
-type point() :: {coordinate(), coordinate()}.
-type obstacles() :: [point()].

-spec solve(map_size(), point(), point(), obstacles()) -> boolean().
solve(N, Start, End, Obstacles) -> dfs(N, Start, End, Obstacles, [], []).

dfs(_N, _End, _End, _Obstacles, _Queue, _Visited) -> true;
dfs(N, Curr, End, Obstacles, Queue, Visited) ->
  case expand(N, Curr, Obstacles, Queue, Visited) of
    [] -> false;
    [M | Ms] ->
      _ = validate_move(M, N),
      dfs(N, M, End, Obstacles, Ms, [M|Visited])
  end.

%% Ensure that the current position is valid
validate_move({X, Y}, N) when X > 0, Y > 0, X =< N, Y =< N -> ok;
validate_move(Point, _N) -> exit({invalid_point, Point}). %% Catch the BUG here

%% Expand the current state and generate possible moves
expand(N, {X, Y}, Obstacles, Pending, Visited) ->
  Up = {X, Y+1},
  Right = {X+1, Y},
  Down = {X, Y-1},
  Left = {X-1, Y},
  L = lists:foldl(
    fun(M, Acc) ->
      case can_move(N, M, Obstacles) andalso (not member(M, Visited)) of
        true -> [M|Acc];
        false -> Acc
      end
    end,
    Pending,
    [Left, Down, Right, Up]
  ),
  L.

%% Check if a move is valid
can_move(N, {X, Y}, _) when X > N; Y > N -> false;
can_move(_, {_, Y}, _) when Y < 1 -> false; %% BUG: Not checking if X < 1
can_move(_, _Point, [_Point|_]) -> false;
can_move(_, _, []) -> true;
can_move(_, Point, Obstacles) -> not member(Point, Obstacles).

%% An implementation of lists:member/2 that is not a BIF
member(_, []) -> false;
member(X, [X|_]) -> true;
member(X, [_|L]) -> member(X, L).
```

Listing 5.3: Depth-First Search example - robot.erl

and not occupied by an obstacle. Our strategy is to first go up, then right, then down and then left in a depth-first search with backtracking.

When we expand a state and consider the possible moves, we only add to our stack the valid moves. In addition, when we actually move to a point we check if this move is valid. If it is not, we throw an exception. If we properly add the valid moves to the stack, then this validation will be redundant. However, we included a bug in our code where we do not prune moves that take the robot to a point that outside the left bounds of the map.

Our entry point to this program is the $solve/4$ function. This function takes four arguments.

1. N: The size of the map. Its type is $map\_size()$ which a range from 1 to 8. The bottom left point is $\{1, 1\}$ and the top right point is $\{N, N\}$
2. $Start$: The starting point of the robot. Its type is $\{coordinate(), coordinate()\}$ where $coordinate()$ is a range from 1 to 8. (This is actually a dependent type that depends on N but Erlang specs are not so expressive as to actually declare this dependence).
3. End: The ending point of the robot. Its type is the $\{coordinate(), coordinate()\}$, just like $Start$.
4. $Obstacles$: The list of the positions of the all obstacles in the map. Its type is as expected $[\{coordinate(), coordinate()\}]$.

If a valid path from $Start$ to End exists then $solve/4$ will return $true$, else it will return $false$.

Let's say that the seed input is $N \to 8$, $Start \to \{6, 7\}$, $End \to \{7, 3\}$ and $Obstacles \to [\{5, 5\}, \{6, 5\}, \{7, 5\}]$. In Listing 5.4, we see the output that CutEr generated for the first three executions.

```
Input: 8, {6,7}, {7,3}, [{5,5},{6,5},{7,5}]
 Result: true
 Path Vertex: ''FTFFTFFFTFTFTFTFTFTTTTFFTFFFTFTFTFTFTFTTTTFFTFFFTF''
-------------------------------------------------
Input: 1, {7,6}, {7,6}, [{8,6},{6,6},{3,5},{6,5},{7,2},{2,2},{2,6},{6,1},{1,4}]
 Result: true
 Path Vertex: ''T''
-------------------------------------------------
Input: 6, {5,7}, {8,7}, []
 Runtime Error: {invalid_point,{0,1}}
 Path Vertex: ''FTTFFFTTTTTFFFFFTFTFFFTFTFFFTFTTTFFFFFTFTFTFFFTFTTT''
```

Listing 5.4: CutEr output when testing the DFS example

It managed to generate a testcase that would find the bug on the third execution but as we can see the input is completely invalid. We have $N \to 1$ and $Start \to \{7, 6\}$. However, this input is valid as far as the type signature of $solve/4$ is concerned. The problem lies in the fact that we could not convey the dependency between the coordinates and the map size.

The solution we use, is to to try and alleviate the dependency by setting N to a fixed value, let's say $N \to 8$. With this setting, the generated testcases make sense and CutEr is still able to locate the bug on the eighth execution, as we see in Listing 5.5.

```
Input: 8, {6,7}, {7,3}, [{5,5},{6,5},{7,5}]
 Result: true
 Path Vertex: ''FTFFTFFFTFTFTFTFTFTTTTFFTFFFFTFTFTFTFTFTTTTFFTFFFTF''
-------------------------------------------------
Input: 8, {1,8}, {1,8}, [{7,6},{8,6},{6,6},{3,5},{6,5},{7,2},{2,2},{2,6},{6,1}]
 Result: true
 Path Vertex: ''T''
-------------------------------------------------
Input: 8, {6,7}, {7,6}, []
 Result: true
 Path Vertex: ''FTFFFTTFFFTTFFFTTFFFTTTFFFFTFTFFFTFTFFFTFTTTFFFFTF''
-------------------------------------------------
Input: 8, {6,3}, {7,6}, [{8,1}]
 Result: true
 Path Vertex: ''FTFFTFFFTFTTTTFFTFFFTFTTTTFFTFFFTFTTTTFFTFFFTFTTTT''
-------------------------------------------------
Input: 8, {4,8}, {1,6}, [{8,2},{3,4}]
 Result: true
 Path Vertex: ''FTFFTFFFTFTFTFTTTTFFTFFFTFTFTFTTTTFFTFFFTFTFTFTTTT''
-------------------------------------------------
Input: 8, {7,1}, {6,3}, [{5,4}]
 Result: true
 Path Vertex: ''FTFFTFFFTFTTTTFTFFTFFFTFTTTTFFTFFFTFTTTTFFTFFFTF''
-------------------------------------------------
Input: 8, {8,1}, {7,6}, [{3,5}]
 Result: true
 Path Vertex: ''FTFFTFFFTFTTTTFTFFTFFFTFTTTTFFTFFFTFTTTTFFTFFFF''
-------------------------------------------------
Input: 8, {7,1}, {1,4}, [{6,2},{5,5},{1,6},{8,7}]
 Runtime Error: {invalid_point,{0,7}}
 Path Vertex: ''FTFFTFFFTFTFTFTFTFTFTFTTTTFTFFFTFFFTFTFTFTFTFTFTT''
```

Listing 5.5: CutEr output when testing the DFS example with fixed $N \to 8$

5.3 An Abstract Datatype Example

This goal of this example is to show how CutEr handles abstract datatypes that consist of more the one clauses and are generally recursive.

Let's consider a language of untyped arithmetic expressions [18]. This language only has expressions. An expression e can be:

- a boolean
- a natural number
- an if-then-else
- a predecessor of a natural number (we define that the predecessor of zero is zero)

- a successor of a natural number
- a check if a natural number is zero

In order to evaluate such expressions, we should define a type e that describes them. The definition of e in EBNF Notation [1] and in Erlang is in Listing 5.6.

```
%% EBNF Notation
e = true
  | false
  | 0
  | if e the e else e
  | succ e
  | pred e
  | iszero e

%% Erlang Type Definition
-type e() :: e_true | e_false | e_0
           | {e_if, e(), e(), e()}
           | {e_succ, e()} | {e_pred, e()} | {e_iszero, e()}.
```

Listing 5.6: Definition of untyped arithmetic expressions

Our program takes an expression of this language and evaluates it (Listing 5.7).

```
-spec eval(e()) -> boolean() | non_neg_integer().
eval(e_true) -> true;
eval(e_false) -> false;
eval(e_0) -> 0;
eval({e_if, true, T, _}) -> eval(T);
eval({e_if, false, _, T}) -> eval(T);
eval({e_if, C, T, F}) ->
  case eval(C) of
    true -> eval(T);
    false -> eval(F)
  end;
eval({e_pred, T}) ->
  case eval(T) of
    0 -> 0;
    N when is_integer(N) -> N - 1
  end;
eval({e_succ, T}) ->
  case eval(T) of
    N when is_integer(N) -> N + 1
  end;
eval({e_iszero, T}) ->
  case eval(T) of
    0 -> true;
    N when is_integer(N), N > 0 -> false
  end.
```

Listing 5.7: Abstract Datatype example - airth.erl

The entry point to our program is $eval/1$. The type specification of this function is $e() \rightarrow boolean()|non\_neg\_integer()$. $e()$ is a recursive type so if we want our solver to

reason over it we should declare it and generate some axioms that would constraint the generated testcases. The problem is that we cannon exhaustively represent with any of the base types and Z3 does not have the notion of subtyping ($e()$ is actually a subtype of $any()$). In order to overcome this limitation, we should unroll the type and set a bound to recursion level. At this point, this operation is not supported by CutEr so the programmer should do it manually.

For example, we unrolled the type only once and produced the spec shown in Listing 5.8.

```
1 -type e1() :: e_true | e_false | e_0
2             | {e_if, any(), any(), any()}
3             | {e_succ, any()} | {e_pred, any()} | {e_iszero, any()}.
4
5 -type e() :: e_true | e_false | e_0
6            | {e_if, e1(), e1(), e1()}
7            | {e_succ, e1()} | {e_pred, e1()} | {e_iszero, e1()}.
```

Listing 5.8: Unrolling $e()$ one time to remove recursion

We run CutEr with the seed input $\{e\_iszero, \{e\_pred, \{e\_succ, \{e\_succ, e\_0\}\}\}\}$ and we get the output shown in Listing 5.9.

CutEr found a bug in our code on the seventh execution when it tried to evaluate $\{e\_pred, e\_true\}$. This is an expression in our language that cannot be evaluated and our interpreter cannot handle it so it crashes. However, some of the other testcases that are produced, like $\{e\_if, e\_true, e\_0, \{e\_iszero, 0\}\}$, are invalid expressions but the solver produces them since we approximated the recursive type $e()$.

```
Input: {e_iszero,{e_pred,{e_succ,{e_succ,e_0}}}}
 Result: false
 Path Vertex: ''FFFFFFTFTFTTFFFFFFTTFFFFFFTFTT''
-------------------------------------------------
Input: e_true
 Result: true
 Path Vertex: ''T''
-------------------------------------------------
Input: e_false
 Result: false
 Path Vertex: ''FT''
-------------------------------------------------
Input: e_0
 Result: 0
 Path Vertex: ''FFT''
-------------------------------------------------
Input: {e_if,e_true,e_0,{e_iszero,0}}
 Result: 0
 Path Vertex: ''FFFTTFTTFTTTFFT''
-------------------------------------------------
Input: {e_pred,{e_iszero,0}}
 Runtime Error: {badmatch,{function_clause,0}}
 Path Vertex: ''FFFFFFTTFFFFFFTFTFTTFFFFFFFFF''
-------------------------------------------------
Input: {e_pred,e_true}
 Runtime Error: {badmatch,{case_clause,true}}
 Path Vertex: ''FFFFFFTTTF''
-------------------------------------------------
Input: {e_succ,{e_iszero,0}}
 Runtime Error: {badmatch,{function_clause,0}}
 Path Vertex: ''FFFFFFTFTTFFFFFFTFTFTTFFFFFFFF''
-------------------------------------------------
Input: {e_pred,e_false}
 Runtime Error: {badmatch,{case_clause,false}}
 Path Vertex: ''FFFFFFTTFTF''
-------------------------------------------------
Input: {e_succ,e_true}
 Runtime Error: {badmatch,{case_clause,true}}
 Path Vertex: ''FFFFFFTFTTTF''
-------------------------------------------------
Input: {e_pred,e_0}
 Result: 0
 Path Vertex: ''FFFFFFTTFFT''
-------------------------------------------------
Input: {e_succ,e_false}
 Runtime Error: {badmatch,{case_clause,false}}
 Path Vertex: ''FFFFFFTFTTFTF''
-------------------------------------------------
Input: {e_pred,{e_if,2,0,1}}
 Runtime Error: {badmatch,{function_clause,2}}
 Path Vertex: ''FFFFFFTTFFFTTFTTFTTFFFFFFFF''
-------------------------------------------------
Input: {e_if,e_false,e_true,{e_iszero,0}}
 Runtime Error: {badmatch,{function_clause,0}}
 Path Vertex: ''FFFTTFTTFTTFTFFFFFFTFTFTTFFFFF''
-------------------------------------------------
Input: {e_iszero,e_true}
 Runtime Error: {badmatch,{case_clause,true}}
 Path Vertex: ''FFFFFFTFTFTTTF''
-------------------------------------------------
Input: {e_succ,e_0}
 Result: 1
 Path Vertex: ''FFFFFFTFTTFFTT''
```

Listing 5.9: CutEr output when testing the evaluation of untyped arithmetic expressions

Chapter 6

Conclusion and Future Work

In this thesis we presented CutEr, a testing tool for Erlang that implements a white-box technique that combines concrete and symbolic execution, namely concolic testing. We have described the architecture and general implementation of the tool and experimented on some basic sequential Erlang programs.

Our experiments revealed that there are a lot of things that need to be addressed in order for CutEr to work successfully and efficiently. The most important have to do with supporting concurrency and reason for more Erlang types, since in its current state it can test a limited amount of programs.

Therefore, we have compiled a list of tasks for the future:

- *Encoding of Erlang types and operations in Z3.* It is imperative that we need to find a more efficient representation of the Erlang type system for Z3 that will at least include binaries and improper lists. This will help CutEr reason for a larger number of programs since it is very common to use files as input which is equivalent to a stream of binary data. In addition, we would want to better address the issue of encoding recursive types.
- *Use more expressive type signatures.* We want the type signatures of the functions to be more expressive and denote the dependencies between their arguments in order to properly generate testcases for them.
- *Serialize path conditions.* The only reason why CutEr cannot reason over concurrent and distributed programs is the fact that these program generate a trace file for each process and the correlation between them can be represented with a tree structure. So in order to reverse a specific constraint, all the correlated constraints should be taken into account, which is not a trivial task. We need to perform a trace analysis and serialize the path predicate.
- *Simplify complex constraints.* When the solver cannot respond whether a model is satisfiable or not, we should assist it by simplifying some constraints by using the concrete values of some variables instead of the symbolic ones. We need to come up with an efficient strategy that will perform this action.
- *Improve the instrumentation algorithm.* Our goal is to be able to target specific assertions in a program's code and try to instrument the search in the execution paths towards those assertions in order to violate them.

- *Make CutEr run concurrently.* Currently, many instances of the interpreter can be spawned and run concurrently. The same applies to the FSM processes that communicate with Z3. So we just need to add the ability to spawn multiple interpreters and FSM processes to the instrumentation algorithm.

Bibliography

[1] Information technology - syntactic metalanguage - extended bnf. ISO/IEC 14977, International Organization for Standardization, 2001.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec. 2008.

[5] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 2000-030, Department of Information Technology, Uppsala University, Nov. 2000.

[6] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[8] Erlang/OTP. http://www.erlang.org/.

[9] Erlang External Term Format. http://www.erlang.org/doc/reference_manual/typespec.html.

[10] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] X. Ge, K. Taneja, T. Xie, and N. Tillmann. Dyta: Dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 992–994, New York, NY, USA, 2011. ACM.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. Technical Report 1, New York, NY, USA, Jan. 2012.

[13] K. Jamrozik, G. Fraser, N. Tillman, and J. Halleux. Generating test suites with augmented dynamic symbolic execution. In M. Veanes and L. Viganò, editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2013.

[14] E. Larson and T. Austin. High coverage detection of Input-Related security faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, Washington, D.C., U.S.A., Aug. 2003. USENIX Association.

[15] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: Concolic verification and test generation for GPUs. *SIGPLAN Not.*, 47(8):215–224, Feb. 2012.

[16] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[17] M. Papadakis. Automatic Random Testing of Function Properties from Specifications. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, October 2010. http://artemis.cslab.ntua.gr/el_thesis/artemis.ntua.ece/DT2010-0295/DT2010-0295.pdf.

[18] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[19] N. Razavi, F. Ivančić, V. Kahlon, and A. Gupta. Concurrent test generation using concolic multi-trace analysis. In R. Jhala and A. Igarashi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 239–255. Springer Berlin Heidelberg, 2012.

[20] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, Champaign, IL, USA, 2006. AAI3242987.

[21] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, FASE'06, pages 339–356, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[24] N. Tillmann and J. De Halleux. Pex: White box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.