ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Περιορισμός του χώρου αναζήτησης του concolic testing μέσω στατικής ανάλυσης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΔΙΟΝΥΣΙΟΥ ΣΠΗΛΙΟΠΟΥΛΟΥ

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

Αθήνα, 16 Μαρτίου 2022

NATION TECHNICAL UNIVERSITY OF ATHENS
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Reducing the search space of concolic testing via static analysis

SENIOR THESIS

of

DIONISIOS SPILIOPOULOS

Supervisor : Konstantinos Sagonas
Assoc. Professor NTUA

Athens, 16th March 2022

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

Περιορισμός του χώρου αναζήτησης του **concolic testing** μέσω στατικής ανάλυσης

Διπλωματική Εργασία

του

ΔΙΟΝΥΣΙΟΥ ΣΠΗΛΙΟΠΟΥΛΟΥ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16/03/2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Κωνσταντίνος Σαγώνας	Άρης Παγουρτζής	Δημήτριος Φωτάκης
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, 16 Μαρτίου 2022

(Υπογραφή)

....

Διονυσιος Σπηλιοπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο έλεγχος προγραμμάτων είναι κρίσιμος για την ανάπτυξη αξιόπιστων εφαρμογών. Το concolic testing είναι μία τεχνική White-Box testing, η οποία επιδιώκει την αυτόματη δημιουργία πολλαπλών εισόδων για ένα πρόγραμμα, προκειμένου να εντοπίσει σφάλματα κατά την εκτέλεση. Αυτή η τεχνική αντιμετωπίζει το πρόβλημα ενός πολύ μεγάλου χώρου αναζήτησης από μονοπάτια εκτέλεσης, του οποίου η εξερεύνηση είναι χρονικά απαιτητική. Σε αυτή την διπλωματική εργασία αναπτύχθηκε μια μέθοδος για τη μείωση του χώρου αναζήτησης κάνοντας χρήση της στατικής πληροφορίας ενός προγράμματος για την αφαίρεση των κλαδιών που δεν παράγουν σφάλματα. Ως στατική πληροφορία χρησιμοποιούμε το αφηρημένο συντακτικό δέντρο του προγράμματος, καθώς και τους τύπους που παρέχονται σε αυτό. Η μέθοδος αυτή αναπτύχθηκε στο CutEr, ένα εργαλείο για concolic testing της γλώσσας προγραμματισμού Erlang. Ακόμα, η μέθοδος ελέγχθηκε με ήδη υπάρχοντα προγράμματα, καθώς και νέα ειδικά σχεδιασμένα για τον σκοπό αυτόν, προκειμένου να αναδειχθεί η χρησιμότητά της. Τέλος, παρατίθενται τα αποτελέσματα του ελέγχου αυτού, όπου φαίνεται ότι οι χρονικές απαιτήσεις του concolic testing μπορούν να μειωθούν ραγδαία με τη χρήση αυτής της μεθόδου αφαίρεσης κλάδων.

Λέξεις Κλειδιά

Concolic Testing, CutEr, Erlang, Έλεγχος προγραμμάτων, White-Box Testing, Μείωση χώρου αναζήτησης

Abstract

Software testing is crucial for developing any reliable application. Concolic testing is a White-Box testing technique, which tries to create various inputs for a program automatically, in order to locate runtime errors. This technique though suffers from a vast search space of execution paths that is inefficient and time-consuming to fully traverse. In our work, we propose a method to reduce this search space by exploiting the static information in a program and pruning branches which will certainly not produce errors. As static information, we use the program's abstract syntax tree, as well as the type annotations provided. This method was implemented in CutEr, a concolic testing tool for the Erlang programming language. We have also tested our method with custom programs and real world code. Finally, we report the results of our testing and show that the time performance of concolic testing can be immensely increased by using our method for safe branch pruning.

Keywords

Concolic Testing, CutEr, Erlang, Software Testing, White-Box Testing, Search Space Reduction

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον Κωστή Σαγώνα για την καθοδήγησή του και το ενδιαφέρον του, τόσο σε θέματα που αφορούν στην διπλωματική μου, όσο και σε στην προσωπική μου εξέλιξη. Τον ευχαριστώ που προσπάθησε ειλικρινά για κάθε ζήτημα να επιλυθεί με τον πλέον ορθό τρόπο. Θα ήθελα ακόμα να ευχαριστήσω τον Νίκο Παπασπύρου που μέσω της άριστης διδασκαλίας του με ενέπνευσε να ασχοληθώ με τον συγκεκριμένο τομέα και για την ικανότητα και την διάθεσή του να δίνει λύση σε κάθε πρόβλημα χρησιμοποιώντας την βαθειά αντίληψή του. Αισθάνομαι τυχερός που είχα την ευκαιρία να συνεργαστώ και με τους δύο.

Ευχαριστώ τον Άγγελο Γιάντσιο για όλη του την ανιδιοτελή βοήθεια και στήριξη.

Τέλος, ευχαριστώ πολύ την οικογένειά μου και τους φίλους μου, που με διαμόρφωσαν στον άνθρωπο που είμαι σήμερα και που ήταν δίπλα μου σε κάθε στιγμή της ζωής μου. Ιδιαίτερα θα ήθελα να ευχαριστήσω την Δανάη που με υποστήριξε όλα αυτά τα χρόνια.

Διονύσιος Σπηλιόπουλος
16 Μαρτίου 2022

Contents

List of Figures

List of Tables

Εκτενής Περίληψη στα Ελληνικά

Σε αυτό το κεφάλαιο παρουσιάζεται η διπλωματική εργασία που εκπονήθηκε μέσω μιας εκτεταμένης περίληψης στα ελληνικά.

Εισαγωγή

Τα συστήματα υπολογιστών αποτελούν σήμερα αναπόσπαστο μέρος κάθε ανθρώπινης δραστηριότητας. Σχεδόν όλες οι διαδικασίες της ανθρώπινης επικοινωνίας, των μεταφορών, των ανταλλαγών, της υγείας, της ψυχαγωγίας, της χαλάρωσης κ.λπ. εξαρτώνται από τα συστήματα υπολογιστών για την υλοποίησή τους. Η λειτουργία αυτών των συστημάτων καθορίζεται από ένα σύνολο οδηγιών, το **λογισμικό**. Το λογισμικό περιλαμβάνει το σύνολο των προγραμμάτων, των διαδικασιών και των ρουτινών που σχετίζονται με τη λειτουργία ενός συστήματος υπολογιστών [1]. Αυτό το λογισμικό αναπτύσσεται από **προγραμματιστές**, οι οποίοι γράφουν το σύνολο των εντολών ως **κώδικα** εκφρασμένο σε μία ή περισσότερες **γλώσσες προγραμματισμού** και εξασφαλίζουν ότι εκτελεί την επιθυμητή λειτουργία. Για να επιτευχθεί αυτό, είναι σημαντικό το προϊόν λογισμικού όχι μόνο να ανταποκρίνεται στις απαιτήσεις, αλλά και να είναι **defect free**, δηλαδή να μην υπάρχουν **σφάλματα** στον κώδικα που θα οδηγούσαν σε απρόβλεπτη συμπεριφορά του προϊόντος.
Σε αυτό το σημείο γίνεται χρήσιμος ο **έλεγχος προγραμμάτων**. Ο έλεγχος προγραμμάτων χρησιμοποιείται για να αυξηθεί η πιθανότητα να μην υπάρχουν λάθη στο λογισμικό, ώστε αυτό να είναι αξιόπιστο και ασφαλές. Ο έλεγχος προγραμμάτων μπορεί να διαχωριστεί σε δύο κατηγορίες: **Black Box Testing** και **White Box Testing** [2, 3]. Το Black Box Testing μπορεί να χρησιμοποιηθεί ακόμη και όταν δεν υπάρχουν πληροφορίες σχετικά με την εσωτερική δομή του κώδικα, τις λεπτομέρειες υλοποίησης ή τις εσωτερικές διαδρομές. Στο White Box Testing η εσωτερική δομή, ο σχεδιασμός και η κωδικοποίηση του λογισμικού είναι γνωστά και χρησιμοποιούνται για τον εντοπισμό σφαλμάτων. Στην παρούσα εργασία, εστιάζουμε μόνο στο White Box Testing και στους τρόπους βελτίωσης των υφιστάμενων μεθόδων ελέγχου αυτής της κατηγορίας.
Το **concrete execution** ενός προγράμματος αναφέρεται στην πλήρη εκτέλεσή του με κάποια καλά καθορισμένη (concrete) είσοδο και στην παρατήρηση της συμπεριφοράς του. Η **συμβολική εκτέλεση** [4] είναι ένας τρόπος ελέγχου ενός προγράμματος μέσω του οποίου προσδιορίζονται οι είσοδοι που προκαλούν την εκτέλεση κάθε εντολής του προγράμματος. Η συμβολική Εκτέλεση χρησιμοποιεί έναν **διερμηνευτή**, ένα πρόγραμμα που εκτελεί όλες τις

εντολές του κώδικα χωρίς καμία προηγούμενη μετατροπή του κώδικα σε εντολές γλώσσας μηχανής. Αυτός ο διερμηνευτής ακολουθεί την ροή του προγράμματος και υποθέτει συμβολικές τιμές για τις εισόδους του, χωρίς να έχει πληροφορίες για τις πραγματικές τιμές εισόδου. Στη συνέχεια, συμπεραίνει εκφράσεις για τις πράξεις και τις μεταβλητές του προγράμματος, καθώς και περιορισμούς για τα πιθανά αποτελέσματα κάθε διακλάδωσης. Επιλύοντας αυτούς τους περιορισμούς, μπορούμε να καθορίσουμε ποιες είσοδοι είναι σε θέση να ακολουθήσουν κάθε κλάδο. Το **concolic testing** [5, 6] είναι μια τεχνική που ερμηνεύει το υπό εξέταση πρόγραμμα, πραγματοποιώντας και concrete και συμβολική εκτέλεση ταυτόχρονα, και καταγράφει το μονοπάτι εκτέλεσης, δηλαδή την ακολουθία των διακλαδώσεων που ακολουθήθηκαν κατά την εκτέλεση. Βλέποντας πώς η είσοδος επηρέασε αυτό το μονοπάτι εκτέλεσης, δημιουργεί έναν συζευκτικό λογικό τύπο, ο οποίος, αν ικανοποιηθεί από οποιαδήποτε είσοδο, το πρόγραμμα θα ακολουθήσει το ίδιο μονοπάτι. Τέλος, χρησιμοποιώντας έναν επιλυτή περιορισμών, αναιρεί έναν όρο αυτού του τύπου για να δημιουργήσει μια είσοδο που θα ακολουθήσει διαφορετικό μονοπάτι εκτέλεσης. Κάνοντας το αυτό συστηματικά, η τεχνική θα διερευνήσει κάθε πιθανό μονοπάτι εκτέλεσης του προγράμματος και θα καταγράψει ποιες είσοδοι παράγουν σφάλμα εκτέλεσης.

Η τεχνική αυτή εφαρμόζεται για πολλές γλώσσες προγραμματισμού κυρίως προστακτικές και αντικειμενοστραφείς, χρησιμοποιώντας μία χαμηλού επιπέδου αναπαράσταση αυτών, όπως η assembly ή το LLVM. Το **CutEr** [7, 8] είναι ένα εργαλείο που υλοποιεί αυτήν την τεχνική για την Erlang, η οποία είναι μια συναρτησιακή γλώσσα προγραμματισμού. Το CutEr δεν ενεργεί στο bytecode που δημιουργεί ο μεταγλωττιστής της Erlang, αλλά στο Abstract Syntax Tree (AST), το οποία είναι μία ενδιάμεση αναπαράσταση του προγράμματος κατά τη διάρκεια της μεταγλώττισης.

Η παρούσα εργασία διερευνά τρόπους αύξησης της απόδοσης ενός τέτοιου concolic testing εργαλείου.

Θεωρητικό Υπόβαθρο

Erlang

Η Erlang είναι μια γλώσσα προγραμματισμού, η οποία σχεδιάστηκε για την ανάπτυξη παράλληλων προγραμμάτων που ¨τρέχουν για πάντα' [9]. Η Erlang δημιουργήθηκε το 1986 στο Εργαστήριο Πληροφορικής της Ericsson, προκειμένου να αντιμετωπιστούν τα άτυπα προβλήματα που υπήρχαν στις εφαρμογές τηλεφωνίας, λόγω της εγγενούς τους παράλληλης λειτουργίας. Λόγω της σπουδαιότητας των σταθερών τηλεπικοινωνιών, ήταν σημαντικό για την Erlang να παρέχει υψηλή **διαθεσιμότητα**, **ανοχή σε σφάλματα** και **ενημέρωση του προγράμματος κατά την λειτουργία**. Οι λειτουργίες έπρεπε επίσης να είναι 'soft real-time', που σημαίνει ότι ενώ κάποιες θα έπρεπε να διακόπτονται αν δεν τελείωναν σε ένα συγκεκριμένο χρονικό διάστημα, άλλες μπορεί να χρειαζόταν να επαναληφθούν σε μια τέτοια περίπτωση.

Μια εφαρμογή Erlang αποτελείται από **modules**. Κάθε module μεταγλωττίζεται ξεχωριστά και αργότερα φορτώνεται στο σύστημα χρόνου εκτέλεσης της Erlang. Αυτό το σύστημα χρόνου εκτέλεσης χρησιμοποιεί ένα register based virtual machine που ονομάζεται **BEAM**. Ο μεταγλωττιστής μετατρέπει κάθε module σε BEAM bytecode και ο φορτωτής με την σειρά του το μετατρέπει σε threaded code.

Για να δημιουργήσει το BEAM bytecode, ο μεταγλωττιστής Erlang πρώτα αναλύει το κείμενο του προγράμματος και δημιουργεί το Abstract Syntax Tree (AST) του module. Δεδομένου ότι η Erlang περιέχει syntactic sugar, το AST υφίσταται έναν ενδιάμεσο μετασχηματισμό που παράγει το ισοδύναμο συντακτικό δέντρο Core Erlang [10]. Τέλος, αυτό το δέντρο μετασχηματίζεται σε BEAM bytecode.

Η Core Erlang [10, 11] χρησιμεύει ως μια ενδιάμεση αναπαράσταση της γλώσσας προγραμματισμού Erlang. Περιέχει ένα μικρό σύνολο δομών, προσφέρει καθαρή σημασιολογία και η μετατροπή από την Erlang σε αυτήν την αναπαράσταση καθώς και η μετατροπή από την Core Erlang σε bytecode είναι αρκετά απλή. Είναι δυνατόν να εκτυπωθεί σε κείμενο που είναι εύκολο να διαβαστεί και να επεξεργαστεί, λόγω της απλής γραμματικής της. Η σημαντικότερη χρήση της είναι η δυνατότητα ανάπτυξης εργαλείων που δρουν στην αναπαράσταση του πηγαίου κώδικα (όπως αναλυτές τύπων [12, 13, 14], στατικοί αναλυτές για παράλληλη μετάδοση μηνυμάτων [15, 16, 17], εργαλεία δοκιμών [18, 19, 20, 21], εργαλεία αναδιοργάνωσης κώδικα [22, 23, 24, 25], κ.λπ.), αποφεύγοντας την πολυπλοκότητα της πλήρους Erlang και της χαμηλού επιπέδου αναπαράστασης του BEAM bytecode. Το εργαλείο CutEr [7, 8], εκτελεί concolic testing στο Core Erlang AST.

Η Erlang είναι dynamically typed και δεν απαιτεί την παροχή type annotations από τον χρήστη. Ωστόσο, οι πληροφορίες των τύπων των δεδομένων υπάρχουν στο χρόνο εκτέλεσης και ελέγχονται σε κάθε λειτουργία για να διασφαλιστεί το type safety. Έτσι, υπάρχουν primitive types στην Erlang. Ο προγραμματιστής μπορεί να προσθέσει type annotations για λόγους σαφήνειας του προγράμματος. Επιπλέον, αυτά τα annotations μπορούν να χρησιμοποιηθούν από εξωτερικά προγράμματα για να αναλύσουν τον κώδικα σε ένα πιο συγκεκριμένο πλαίσιο. Ο προγραμματιστής μπορεί να δηλώσει και να ορίσει καινούριους τύπους χρησιμοποιώντας το annotation `type` ή `opaque` και να ορίσει records με το annotation `record`. Επιπλέον, οι υπογραφές συναρτήσεων μπορούν να καθοριστούν με το annotation `spec`. Όλα αυτά τα annotations δεν θα ληφθούν υπόψη από τον μεταγλωττιστή και υπάρχουν για να κάνουν την ανάπτυξη του κώδικα πιο δομημένη και τεκμηριωμένη, καθώς και για να διευκολύνουν τη διαδικασία αποσφαλμάτωσης μέσω της χρήσης εξωτερικών εργαλείων.

Concolic Testing

Η λέξη 'concolic' είναι αποτέλεσμα της συγχώνευσης των λέξεων 'concrete' και 'symbolic' που αντιπροσωπεύουν τη διπλή αναπαράσταση των τιμών κατά τη διάρκεια της διερμήνευσης του προγράμματος. Στο concolic testing, εκτελείται το πρόγραμμα και με συμβολικό και με concrete τρόπο. Το συμβολικό μέρος της εκτέλεσης είναι απαραίτητο για τη δημιουργία των νέων εισόδων μέσω της επίλυσης περιορισμών και η concrete εκτέλεση είναι απαραίτητη για να καθοδηγήσει τη συμβολική εκτέλεση σε ένα συγκεκριμένο μονοπάτι [26, 5, 7, 8, 27, 28].

Κατά τη διάρκεια της εκτέλεσης, διατηρείται τόσο ένα συμβολικό όσο και ένα concrete περιβάλλον. Στο concrete περιβάλλον έχουμε αντιστοιχίσεις από τις μεταβλητές στις concrete τιμές τους, ενώ στο συμβολικό περιβάλλον έχουμε αντιστοιχίσεις από τις μεταβλητές στις συμβολικές εκφράσεις τους. Για να ξεκινήσει η διαδικασία, απαιτείται μια αρχική είσοδος, είτε τυχαία παραγόμενη είτε παρεχόμενη από τον χρήστη. Στη συνέχεια, το πρόγραμμα διερμηνεύεται με αυτήν την είσοδο παράγοντας μια διαδρομή εκτέλεσης, διατηρώντας παράλληλα τους περιορισμούς από το συμβολικό περιβάλλον σε κάθε δήλωση διακλάδωσης. Η σύζευξη όλων αυτών των περιορισμών σχηματίζει μια λογική έκφραση, η οποία θα οδηγήσει κάθε είσοδο που την ικανοποιεί στο ίδιο μονοπάτι εκτέλεσης. Έτσι, εάν οποιοσδήποτε όρος αυτού του τύπου που αντιστοιχεί σε μια διακλάδωση στο πρόγραμμα αντιστραφεί, μια είσοδος που ικανοποιεί τον νέο τύπο θα εξερευνήσει ένα διαφορετικό μονοπάτι, αποκλίνοντας από το προηγούμενο σε αυτήν την διακλάδωση. Η διατήρηση όλων των μονοπατιών εκτέλεσης και η αντιστροφή όλων των πιθανών όρων που αντιστοιχούν σε όλους τους κλάδους σε αυτά τα μονοπάτια εκτέλεσης θα καλύψει αναπόφευκτα ολόκληρο το χώρο αναζήτησης των πιθανών εισόδων. Προκειμένου να βρεθεί μια είσοδος που να ικανοποιεί κάθε νέο κατηγόρημα που δημιουργείται, χρησιμοποιείται ένας επιλυτής περιορισμών, όπως το lp_solve, το ECLiPSe [29] ή το Z3 [30].

Το concolic testing επιτρέπει ενδελεχή έλεγχο του προγράμματος και εκμεταλλεύεται την εσωτερική δομή του για την αποτελεσματική εύρεση εισόδων που παράγουν σφάλματα κατά τη διάρκεια εκτέλεσης. Ωστόσο, υπάρχουν διάφοροι περιορισμοί σε αυτή την τεχνική και πολλές προτεινόμενες μέθοδοι για την αντιμετώπισή τους.

Η πιο δυσεπίλυτη από αυτές τις προκλήσεις είναι η μη πληρότητα της επίλυσης των περιορισμών. Ανάλογα με το πρόγραμμα εισόδου, ένας περιορισμός που παράγεται από ένα μονοπάτι εκτέλεσης μπορεί να απαιτεί πολύ χρόνο για να επιλυθεί από τον επιλυτή περιορισμών, ο οποίος μπορεί ακόμη και να μην είναι σε θέση να τον επιλύσει. Αυτό καθιστά την διερεύνηση ορισμένων προγραμμάτων αδύνατη, καθώς δεν θα είναι εφικτή η δημιουργία νέων εισόδων [6, 5].

Μια άλλη σημαντική πρόκληση, η οποία αποτελεί το κύριο αντικείμενο αυτής της εργασίας, είναι το εύρος του χώρου αναζήτησης. Ανεξάρτητα από το μέγεθος του προγράμματος, τα δεδομένα που θα παραχθούν κατά την εκτέλεση του είναι πιθανόν να είναι πολλά. Ακόμη, ορισμένα μονοπάτια μπορεί να είναι πολύ μεγάλα και να δημιουργούν εκτενείς εκφράσεις περιορισμών, οι οποίοι είναι δύσκολο να αντιμετωπιστούν από τον επιλυτή περιορισμών. Παρόλα αυτά, είναι σημαντικό να βρεθούν τα πιθανά σφάλματα εκτέλεσης σε ένα λογικό χρονικό διάστημα, οπότε πρέπει να κατευθυνθεί η concolic εκτέλεση σε συγκεκριμένα μονοπάτια χρησιμοποιώντας διάφορες ευρετικές μεθόδους [31, 32, 33, 34, 35]. Αυτές δεν μειώνουν τον χώρο αναζήτησης αλλά επιχειρούν να κατευθύνουν την αναζήτηση με τέτοιο τρόπο ώστε να βρεθούν τα σφάλματα γρηγορότερα. Το CutEr κάνει αναζήτηση breadth first με οριοθετημένο βάθος.

Τέλος, πολλά προγράμματα μπορεί να παρουσιάζουν μια μη ντετερμινιστική συμπεριφορά (π.χ. χρησιμοποιώντας τυχαίους αριθμούς). Αυτό αποτελεί πρόβλημα, καθώς το μονοπάτι εκτέλεσης μπορεί να μην σχετίζεται αποκλειστικά με τη φόρμουλα που παράγεται από τη συμβολική εκτέλεση, και έτσι, να εμποδίζει τον αλγόριθμο να δημιουργήσει εισόδους για την εξερεύνηση νέων επιλεγμένων κλάδων.

Περιορισμός Χώρου Αναζήτησης

Η μέθοδος που ακολουθήθηκε σε αυτήν την εργασία εστιάζει στον περιορισμό του χώρου αναζήτησης του concolic testing αποτρέποντάς το από το να ακολουθήσει ορισμένους κλάδους του προγράμματος που έχουν προκαθοριστεί ως 'ασφαλείς' μέσω μιας στατικής ανάλυσης. Με τον όρο ασφαλείς εδώ εννοείται η βεβαιότητα ότι ο συγκεκριμένος κλάδος δεν θα καταλήξει σε σφάλμα ανεξάρτητα από την είσοδο του προγράμματος. Η μέθοδος αυτή υλοποιήθηκε στο εργαλείο CutEr [7, 8]. Η στατική ανάλυση γίνεται στο συντακτικό δέντρο του προγράμματος, προσθέτωντας σε κάθε κόμβο συγκεκριμένες επισημειώσεις. Έπειτα, κατά την εκτέλεση του concolic testing εκτελείται ένας δυναμικός έλεγχος χρησιμοποιώντας αυτές τις επισημειώσεις, ο οποίος αποκλείει τους ασφαλείς κλάδους από μέλλουσα εξερεύνηση. Ο αποκλεισμός αυτός γίνεται δυνατός σταματώντας την αποθήκευση των συμβολικών περιορισμών στις επερχόμενες διακλαδώσεις, καθώς χωρίς αυτές το CutEr δεν θα προσπαθήσει να παραγάγει εισόδους προς εξερεύνησή τους.
Η στατική ανάλυση είναι υπεύθυνη για την προσθήκη των εξής επισημειώσεων στους κόμβους:

- `maybe_error`: Έχει τρεις δυνατές τιμές. Όταν έχει την τιμή `true`, το υποδέντρο με ρίζα τον συγκεκριμένο κόμβο μπορεί να παραγάγει σφάλματα. Όταν έχει την τιμή `false`, το υποδέντρο με ρίζα τον συγκεκριμένο κόμβο δεν μπορεί να παραγάγει σφάλματα, οποιαδήποτε και αν είναι η είσοδος του προγράμματος. Όταν έχει την τιμή `type_dependent`, το υποδέντρο με ρίζα τον συγκεκριμένο κόμβο δεν μπορεί να παραγάγει σφάλματα, δεδομένου του ότι η είσοδος ικανοποιεί την υπογραφή που έχει οριστεί για την συνάρτηση που περιέχει αυτόν τον κόμβο.

- `type_dependent_unreachable`: Επισημείωση που μπορεί να βρίσκεται μόνο σε `clause` κόμβους ενός `case`. Υποδηλώνει ότι αυτά τα `clauses` δεν μπορούν να επιλεχθούν με κανέναν τρόπο αν η είσοδος της συνάρτησης ικανοποιεί την υπογραφή της.

- `force_constraint_logging`: Boolean τιμή. Όταν είναι αληθής, δεν πρέπει να απορρίψουμε επερχόμενους κλάδους, όποια και αν είναι η `maybe_error` επισημείωσή τους.

- `distrust_type_dependent`: Boolean τιμή. Όταν είναι αληθής, θεωρούμε ότι οι κόμβοι με επισημείωση `maybe_error` ίση με `type_dependent` λειτουργούν σαν να είχαν την αντίστοιχη επισημείωση `false`.

Παρακάτω παρουσιάζεται πρώτα ο δυναμικός έλεγχος, που χρησιμοποιεί αυτές τις επισημειώσεις, και ύστερα η στατική ανάλυση που τις υπολογίζει.

Δυναμικός Έλεγχος

Ο δυναμικός έλεγχος είναι μέρος του διερμηνευτή του CutEr. Σε αυτόν χρησιμοποιούνται τρείς Boolean μεταβλητές μέσω των οποίων κρίνεται το κατά πόσον θα αποθηκευτούν οι επερχόμενοι περιορισμοί στο πρόγραμμα. Αν δεν αποθηκευτούν, το CutEr δεν θα εξερευνήσει διαφορετικές διαδρομές, δεδομένου του ότι ο επιλυτής περιορισμών δεν θα έχει πρόσβαση στους περιορισμούς που αντιπροσωπεύουν τις επερχόμενες διακλαδώσεις. Οι μεταβλητές αυτές είναι οι ακόλουθες:

1. `constraintLogging` αρχικοποιημένο σε `true`
2. `isForced` αρχικοποιημένο σε `false`
3. `distrustTypeDependent` αρχικοποιημένο σε `false`

Οι συναρτήσεις που αποθηκεύουν τους περιορισμούς λαμβάνουν υπ᾽ όψιν τις δύο πρώτες μεταβλητές για να αποφασίσουν αν θα λειτουργήσουν. Αν η `isForced` είναι αληθής, τότε οι περιορισμοί αποθηκεύονται ανεξάρτητα από τις τιμές των υπολοίπων μεταβλητών. Στην αντίθετη περίπτωση, οι περιορισμοί αποθηκεύονται αν και μόνον αν η `constraintLogging` είναι αληθής. Η μεταβλητή `distrustTypeDependent` υπάρχει για να επηρεάζει το πώς διαχειρίζεται ο δυναμικός έλεγχος τις επισημειώσεις `maybe_error` σύμφωνα με τον Πίνακα 4.1.
Η λογική που υλοποιήθηκε για τον δυναμικό έλεγχο συνοψίζεται στις παρακάτω περιπτώσεις, που αντιστοιχούν σε κόμβους που διαχειρίζεται ο διερμηνευτής.

- `apply`/`call`: Αν ο κόμβος έχει την επισημείωση `distrust_type_dependent`, η μεταβλητή `distrustTypeDependent` λαμβάνει την τιμή `true`.
- `let`: Αν ο κόμβος έχει την επισημείωση `force_constraint_logging`, η μεταβλητή `isForced` λαμβάνει την τιμή `true`.
- `case`: Αν η μεταβλητή `isForced` είναι αληθής, ή η επισημείωση `maybe_error` (επηρεασμένη από την μεταβλητή `distrustTypeDependent`) είναι αληθής, το όρισμα του `case` καλείται με την μεταβλητή `isForced` ως αληθή. Ύστερα, εντοπίζεται το `clause` που θα τρέξει. Τέλος, αν η μεταβλητή `isForced` είναι `false` και η επισημείωση `maybe_error` του συγκεκριμένου `clause` είναι επίσης `false`, η μεταβλητή `constraintLogging` λαμβάνει την τιμή `false`.

Αξίζει να σημειωθεί ότι η καταγραφή των περιορισμών μπορεί να διακοπεί μόνο σε κόμβο `case`, όπου επιλέγεται ένα ασφαλές `clause` ενώ περιέχει και μη ασφαλή `clauses`.

Στατική Ανάλυση

Η στατική ανάλυση χωρίζεται σε τέσσερα στάδια που φαίνονται στο διάγραμμα ροής στην Εικόνα 4.1 τα οποία αναλύονται ξεχωριστά στην συνέχεια.

Contract Gathering

Στην Erlang οι υπογραφές των συναρτήσεων που όρισε ο χρήστης ονομάζονται contracts. Αυτό το στάδιο της στατικής ανάλυσης μετατρέπει τα contracts των συναρτήσεων σε κάποια μορφή που να υποστηρίζει διάφορους χειρισμούς τύπων. Η μορφή που επιλέχθηκε είναι αυτή που ορίζεται στο module `erl_types`. Διαθέτει πλήρη αναπαράσταση των τύπων της Erlang και υποστηρίζει πράξεις σε τύπους, όπως η εύρεση του ελάχιστου δυνατού υπερτύπου κάποιων τύπων και η αφαίρεση τύπων.

Στο στάδιο αυτό, μετατρέπονται πρώτα οι ορισμένοι από τον χρήστη τύποι σε `erl_types` και έπειτα οι υπογραφές. Αυτό γίνεται με έναν υπολογισμό σταθερού σημείου, καθώς κάποιοι τύποι ή υπογραφές μπορεί να ορίζονται πριν από άλλους τους οποίους χρησιμοποιούν. Γι' αυτόν τον λόγο, σε αυτό το στάδιο χρησιμοποιείται ένα σύνολο από τύπους που δεν έχουν μετατραπεί ακόμα (αρχικά περιέχον όλους τους τύπους) και ο αλγόριθμος μετατροπής εκτελείται μέχρι το σύνολο αυτό να παραμείνει σταθερό.

Callgraph Processing

Σε αυτό το στάδιο υπολογίζεται το callgraph του προγράμματος με ρίζα την συνάρτηση που καλείται να αναλύσει το CutEr. Με τον όρο callgraph εννοούμε το γράφημα που ως κόμβους έχει συναρτήσεις και ως ακμές τις κλήσεις μεταξύ συναρτήσεων. Το callgraph είναι κατευθυνόμενο γράφημα. Ο υπολογισμός του γίνεται με χρήση του `xref` το οποίο είναι ένα cross reference εργαλείο για την Erlang. Το callgraph που δημιουργείται μπορεί να περιέχει κύκλους καθώς η Erlang υποστηρίζει αναδρομή και αμοιβαία αναδρομή. Οι κύκλοι όμως δημιουργούν πρόβλημα, διότι για το τελευταίο τμήμα της στατικής ανάλυσης χρειάζεται η γνώση των επισημειώσεων όλων των συναρτήσεων, που καλούνται από μία υπό εξέταση συνάρτηση για να επισημειωθεί και αυτή ορθώς. Γι' αυτόν τον λόγο, αυτό το τμήμα της στατικής ανάλυσης μετατρέπει το callgraph σε ακυκλικό κατευθυνόμενο γράφημα βρίσκοντας όλους τους περιεχόμενους κύκλους και αντικαθιστώντας τους από έναν κόμβο. Μετά από την μετατροπή, το callgraph περιέχει κόμβους που αντιστοιχούν σε μία συνάρτηση και κόμβους που αντιστοιχούν σε ένα σύνολο συναρτήσεων με κυκλικές σχέσεις. Το τελευταίο τμήμα της ανάλυσης μπορεί να επισημειώσει όλες τις συναρτήσεις στο καινούριο callgraph ξεκινώντας από τα φύλλα και πηγαίνοντας προς την ρίζα του γραφήματος σε σειρά depth first. Όταν ο κόμβος που αναλύεται είναι απλή συνάρτηση, η επισημείωση της είναι απλή καθώς είναι γνωστές οι επισημειώσεις για όλες τις συναρτήσεις που αυτή καλεί. Όταν ο κόμβος που αναλύεται είναι σύνολο από συναρτήσεις με κυκλική σχέση, επισημειώνεται κάθε μία από τις συναρτήσεις στο σύνολο αγνοώντας τις κλήσεις συναρτήσεων που δεν έχουν επισημειωθεί ακόμα. Αν υπάρξει αλλαγή στις επισημειώσεις οποιασδήποτε από τις συναρτήσεις σε αυτό το σύνολο, η διαδικασία επαναλαμβάνεται για αυτό το σύνολο. Ένα παράδειγμα αυτής της μετατροπής του callgraph δίνεται στην Εικόνα 4.6 που αντιστοιχεί στο module που φαίνεται στην εικόνα 4.5.

Type Annotation

Σε αυτό το στάδιο, κάθε κόμβος του συντακτικού δέντρου, επισημειώνεται με τον τύπο που αναμένεται να έχει κατά την εκτέλεση αν η είσοδος κάθε συνάρτησης υπακούει στην υπογραφή που παρείχε ο προγραμματιστής. Αυτά τα δεδομένα τύπων είναι πολύ σημαντικά για το τελευταίο τμήμα της ανάλυσης, καθώς, χωρίς αυτά, τα περισσότερα προγράμματα θα έπρεπε να καταγραφούν ως ανασφαλή. Για παράδειγμα, η πρόσθεση δύο τιμών στο πρόγραμμα δεν είναι ασφαλής αν δεν είναι γνωστό ότι οι τιμές αυτές είναι αριθμητικές.

Για την επισημείωση των τύπων των κόμβων μίας συνάρτησης, αρχικοποιείται ένα περιβάλλον στο οποίο προστίθενται τα ορίσματα με τους τύπους που ορίστηκαν στην υπογραφή της. Ύστερα διατρέχει το συντακτικό δέντρο της συνάρτησης σε depth first σειρά και συμπεραίνει τον τύπο κάθε κόμβου με βάση τους τύπους των παιδιών του. Επειδή η στατική ανάλυση πρέπει να κατηγοριοποιεί τους κόμβους για τους οποίους δεν έχει αρκετά δεδομένα ως μη ασφαλείς, το παρόν τμήμα επισημειώνει τους κόμβους για τους οποίους δεν μπορούσε να εξαχθεί ο τύπος με βεβαιότητα, ώστε να μην τους λάβει υπ' όψιν το επόμενο τμήμα.

Μία σημαντική επιπρόσθετη λειτουργία αυτού του τμήματος είναι ο εντοπισμός των `clauses` τα οποία δεν υπάρχει τρόπος να επιλεχθούν σε κάποιο `case` δεδομένης της πληροφορίας που έχουμε για τους τύπους. Τέτοια `clauses` υπάρχουν αρκετά συχνά καθώς, αν ο προγραμματιστής δεν βάλει σε κάποιο `case` ένα `clause` το οποίο να καλύπτει όλες τις περιπτώσεις, το προσθέτει ο μεταγλωττιστής αυτόματα. Σε αυτό το `clause` παράγεται ένα σφάλμα εκτέλεσης κάτι που καθιστά το `clause` μη ασφαλές, και με την σειρά του όλο το δέντρο από εκεί και πάνω. Ο τρόπος με τον οποίο εντοπίζονται αυτά τα `clauses` χρησιμοποιεί τον τύπο του ορίσματος του `case`. Ξεκινώντας από τον τύπο του ορίσματος, αφαιρείται ο τύπος που αντιπροσωπεύει κάθε `clause` από αυτόν. Αν αυτός ο τύπος καταλήξει να είναι `none` τότε δεν υπάρχει περίπτωση να επιλεχθεί επόμενο `clause`.

Τέλος, το στάδιο αυτό παράγει, όπου είναι δυνατόν, τις υπογραφές των συναρτήσεων που δεν δίνονται από τον προγραμματιστή ώστε να επισημειώσει και αυτές. Το να μην παρέχεται υπογραφή για κάποια συνάρτηση δεν είναι σπάνιο, ειδικά για συναρτήσεις που δεν είναι `exported` και επομένως η χρήση τους είναι σαφής, καθώς περιορίζεται μέσα στο `module`. Αυτό σημαίνει ότι κάποιες υπογραφές δεν είναι γνωστές στην αρχή και, όταν βρεθούν, μπορεί να επηρεαστούν οι συναρτήσεις που καλούν μια συνάρτηση με την νέα υπογραφή. Συνεπώς, αυτό το στάδιο είναι επίσης ένας υπολογισμός σταθερού σημείου και επαναλαμβάνεται έως ότου δεν υπάρχει αλλαγή στις επισημειώσεις τύπων των συναρτήσεων και δεν υπάρχει αλλαγή στις υπογραφές που παράγονται.

Error Annotation

Σε αυτό το στάδιο υπολογίζονται οι επισημειώσεις που απαιτούνται από τον δυναμικό έλεγχο για την απόρριψη κλάδων. Δεδομένης της σειράς με την οποία εξετάζεται κάθε συνάρτηση όπως ορίστηκε στο τμήμα της δημιουργίας του callgraph, θεωρούνται για κάθε συνάρτηση γνωστές οι επισημειώσεις των συναρτήσεων τις οποίες καλεί. Η επισημείωση γίνεται σε depth first σειρά ξεκινώντας από τα φύλλα του συντακτικού δέντρου της συνάρτησης και πηγαίνοντας προς την ρίζα. Συνεπώς, για κάθε κόμβο του συντακτικού δέντρου θα είναι γνωστές οι επισημειώσεις όλων των παιδιών του. Στην γενική περίπτωση για κάθε κόμβο, η `maybe_error` επισημείωση του κόμβου είναι ο συνδυασμός των `maybe_error` επισημειώσεων των παιδιών του με την πράξη που περιγράφεται στον Πίνακα 4.2, η οποία μοιάζει με την πράξη της σύζευξης στην λογική με τρείς τιμές.

Η επισημείωση `force_constraint_logging` μπαίνει σε κόμβους τύπου `let`, οι οποίοι εισάγουν μεταβλητές που θα χρησιμοποιηθούν στο όρισμα κάποιου μη ασφαλούς `case`. Αυτοί οι `let` κόμβοι εντοπίζονται ως εξής: με ένα σύνολο A (αρχικοποιημένο ως κενό), κάθε μεταβλητή που βρίσκεται στο όρισμα ενός μη ασφαλούς `case` κόμβου προστίθεται στο A. Όταν αναλύεται ένας `let` κόμβος, αφού αναδρομικά αναλυθούν τα παιδιά του, εξετάζεται αν κάποια από τις μεταβλητές που εισαγάγει υπάρχει στο σύνολο A. Αν ναι, τότε επισημειώνεται με `force_constraint_logging`.

Η επισημείωση `distrust_type_dependent` προστίθεται σε κόμβους τύπου `call` ή `apply` που καλούνται με ορίσματα τουλάχιστον ένα εκ των οποίων δεν είναι ασφαλές. Το όρισμα αυτό μπορεί να είναι συνάρτηση άρα έχει νόημα να χαρακτηριστεί ασφαλές ή μη. Αυτή η επισημείωση υπάρχει ώστε να είναι δυνατός ο χαρακτηρισμός των higher order συναρτήσεων ως ασφαλείς ή μη. Όταν αναλύεται μια higher order συνάρτηση θεωρούμε ότι τα ορίσματα-συναρτήσεις που λαμβάνει είναι ασφαλή με βάση την υπογραφή της συνάρτησης. Η επισημείωση `distrust_type_dependent` υπάρχει ώστε να αναιρέσει αυτήν την υπόθεση όταν βρεθεί μια κλήση σε μία higher order συνάρτηση με κάποιο όρισμα-συνάρτηση, που είτε δεν είναι ασφαλές, είτε δεν υπακούει στην υπογραφή της καλουμένης συνάρτησης.

Πειραματικά Αποτελέσματα και Συμπεράσματα

Αφού υλοποιήθηκε η συγκεκριμένη μέθοδος στο εργαλείο CutEr έγιναν δοκιμές της σε διάφορα προγράμματα, κάποια για να αναδείξουν τις ιδιότητες τις μεθόδου, και κάποια από την βιβλιοθήκη της Erlang. Ένα πρόγραμμα που αναδεικνύει την βελτίωση που επιτυγχάνεται από την στατική ανάλυση φαίνεται στην Εικόνα 4.25. Αυτό το πρόγραμμα περιέχει μια αναδρομική συνάρτηση κάτι που θα δημιουργήσει μονοπάτια εκτέλεσης αποτελούμενα από απροσδιόριστο αριθμό διακλαδώσεων. Όλες αυτές όμως οι διακλαδώσεις αντιστοιχούν στατικά στους ίδιους κόμβους του συντακτικού δέντρου, άρα, αν αυτοί χαρακτηριστούν ως ασφαλείς, το CutEr θα καταλήξει αμέσως στο συμπέρασμα ότι δεν υπάρχουν είσοδοι που παράγουν σφάλματα. Αυτό είναι και το πραγματικό αποτέλεσμα, όταν εκτελείται το CutEr με αυτό το πρόγραμμα ως είσοδο. Με την προσθήκη της στατικής ανάλυσης προσπαθεί να δημιουργήσει τρεις διαφορετικές εισόδους και τερματίζει ανεξάρτητα από το βάθος της εξερεύνησης, ενώ, χωρίς την στατική ανάλυση, προσπαθεί να δημιουργήσει εισόδους που αυξάνονται σε αριθμό, σύμφωνα με το βάθος της εξερεύνησης. Στην περίπτωση που το βάθος είναι στην προδιαγεγραμμένη του τιμή (δηλαδή 25) προσπαθεί να δημιουργήσει 15 διαφορετικές εισόδους.

Εκτός αυτού, η μέθοδος δοκιμάστηκε σε διάφορες συναρτήσεις από το module `lists` του standard library της Erlang και φάνηκε βελτίωση σε αρκετές από αυτές η οποία συνοψίζεται στον Πίνακα 5.1. Να σημειωθεί ότι δεν είναι εύκολο να καθοριστεί η τάξη της χρονικής βελτίωσης, καθώς, σε όλα αυτά τα παραδείγματα, η προσθήκη της στατικής ανάλυσης προκαλεί την εξερεύνηση του χώρου σε σταθερό χρόνο, ενώ, χωρίς αυτήν, ο αριθμός προσπαθειών εξαρτάται από το βάθος της εξερεύνησης.

Το κύριο συμπέρασμα που προκύπτει από αυτήν την εργασία είναι ότι η στατική ανάλυση μπορεί να βελτιώσει την χρονική επίδοση του concolic testing, καθώς εκμεταλλεύεται την στατική πληροφορία του προγράμματος που δεν είναι διαθέσιμη κατά την διάρκεια του concolic testing. Επίσης, η συγκεκριμένη μέθοδος μπορεί να λειτουργήσει ταυτόχρονα με μεθόδους που αλλάζουν την σειρά εξερεύνησης του χώρου αναζήτησης [31, 32, 33, 34, 35] και δεν τις αντικαθιστά.

Chapter 1

Introduction

Computer systems nowadays are an integral part of every human activity; almost all procedures in human communication, transport, exchange, health, entertainment, relaxation, etc. depend on computer systems for their realization. The operation of such systems is defined by a set of instructions, namely the **software**. Software comprises the entire set of programs, procedures, and routines associated with the operation of a computer system [1]. This software is developed by **programmers**, who write the set of instructions as **code** expressed in one or more **programming languages** and ensure that it executes the desired functionality. In order to achieve this, it is important that the software product not only matches expected requirements, but also is **defect free**, meaning that no **bugs** or **errors** exist in the code that would result in unpredictable behavior of the product.

This is the point where **software testing** becomes useful. Software testing is used to increase confidence that no mistakes exist in the software, so that it is reliable and secure. Software testing can be separated into two categories: **Black Box Testing** and **White Box Testing** [2, 3]. Black Box testing can be used even when there exists no information about the internal code structure, implementation details or internal paths. In White Box testing the internal structure, design and coding of the software are known and used locating errors. In this work, we only focus on White Box Testing and ways to improve existing testing methods of this category.

Concrete execution of a program refers to fully executing it with some well defined (concrete) input and observing its behavior. **Symbolic execution** [4] is a testing means of analyzing a program to determine the inputs that cause each program instruction to execute. Symbolic execution uses an **interpreter**, a program that executes all code instructions without any previous transformation of the code into machine language instructions. This interpreter follows the program and assumes symbolic values for the program inputs, without having information about the actual input values. It then deduces unique expressions for operations and variables in the program, as well as constraints for the possible outcomes of each conditional branch. By solving these constraints, we can determine which inputs are able to follow each branch. **Concolic testing** [5, 6] is a technique that interprets the program under examination, symbolically and concretely simultaneously, and records the execution path, which is the sequence of branches that were followed during the execution. Seeing how the input affected this execution path, it creates a conjunctive logical formula which, if satisfied by any input, the program will follow the same path. Finally, using a constraint solver, it negates a term of this formula to create an input that will follow a different execution path. Doing this systematically will cause the technique to explore every conceivable execution path of the program, and record which inputs produce a runtime error.

This technique is implemented for many programming languages covering imperative and object-oriented ones, using the low level representation of these languages such as assembly or LLVM. **CutEr** [7, 8] is a tool that implements this technique for Erlang, which is a functional programming language. CutEr does not act on the low level bytecode created by the Erlang compiler, but the higher level Abstract Syntax Tree (AST) representation, which is created as an intermediate representation of the program during the compilation. This thesis explores ways to increase the performance of such a concolic testing tool.

1.1 Problem Statement

The most important challenge any concolic testing tool faces is the large number of the execution paths. Concolic testing generates the next input by selecting a branch of a previous execution path, trying to follow the opposite branch direction. This implies that the search space is exponential with respect to the number of branches. Additionally, the number of branches may be virtually infinite in case of recursive calls, since the execution path is recorded dynamically. This deems an exhaustive search impossible to be performed. To traverse this search space, many search strategies have been proposed [31, 32, 33, 35, 34]. These have all been implemented on concolic testing tools acting on the low level representation of the program. They essentially introduce a way to prioritize branches to be chosen for exploration in order to reach points of interest in the program faster, or accelerate the program statement coverage of the execution paths. Note that they do not reduce the search space, they are just determining the order of its traversal. In CutEr however, we are provided with a higher level representation of the program, which enables us to extract more information statically.

We propose a strategy that performs a static analysis on the program, which a priori excludes branches that are found to be safe from the concolic execution. Safe, in this context, means that the program will not produce any runtime errors if this branch is followed, regardless of the input. To do that, it exploits the type data that are present in such a high level representation. In order to not divert the concolic execution from branches that could potentially produce errors, and this way compromise the tool's **reliability**, this analysis is pessimistic. By reliability, we are referring to the tool's ability to locate runtime errors. This analysis should not prevent the tool from finding any errors that it could previously find. As a result, when not enough information exists for a branch to be characterized as safe, it is not excluded from the search. This strategy is implemented and tested on CutEr.

1.2 Thesis Organization

In Chapter 2, we inspect the characteristics of the Erlang/OTP system. We also explain how Erlang code is represented in its intermediate AST form, which will be used by CutEr, and how the type information is present in the AST. In Chapter 3 we explain how concolic testing works and its challenges and limitations. We also describe how CutEr works in order to understand how it can be improved by our proposed method. In Chapter 4 we present the algorithm used for safe-branch pruning. In Chapter 5 we gather some experimental results. Finally, in Chapter 6 we give some conclusions and ideas for future research.

Chapter 2

The Erlang/OTP System

2.1 The Erlang Language

Erlang is a programming language, which was designed for developing concurrent programs that "run forever" [9]. Erlang was created in 1986 at the Ericsson Computer Science Laboratory in order to deal with the atypical problems that existed in telephony applications, due to their inherit concurrency. Due to the importance of stable telecommunications, it was important for Erlang to provide high **availability**, **fault-tolerance** and **on-the-fly update** operations. Operations also needed to be "soft real-time", meaning that while some would need to abort if they were not finished in a certain time interval, others might need to be repeated in such case.

Processes in Erlang are more lightweight that standard operating systems (OS) processes and their creation is fast. Process communication is realized asynchronously through messages which are traceable and safely transported.

Fault-tolerance in the case of Erlang does not mean that there will be no processes that will fail, or no data will be lost. Instead, it means that even if that happens, the whole system will not crash and will keep being operational. While this happens, processes may start again and their code itself may be altered on the fly. This ensures that the system will be always on and at the same time able to be updated.

Erlang is a strict and dynamically-typed programming language. This means that the programmer does not have to provide type information statically in the program. However, type safety is ensured in the runtime of the program by environment and, when a type error occurs, a corresponding exception is raised. This behavior allows the programmer to write code with more flexibility but retains the fault-tolerance that characterizes Erlang.

The Open Telecom Platform (OTP) is a middleware platform designed to run on top of an OS and includes important tools for running Erlang, such as compilers, development tools and libraries. It is based on certain design principles for processes, modules and directories, in order to achieve the desired scalability [36].

2.2 The Erlang Compiler

An Erlang application consists of **modules**. Each module is independently compiled and later loaded to the Erlang runtime system. This runtime system uses a register-based virtual machine called **BEAM**. The compiler converts each module into BEAM bytecode and the loader converts this bytecode into threaded code upon loading it.
To create BEAM bytecode, the Erlang compiler first parses the text of the program and creates the Abstract Syntax Tree (AST) of the module. Since Erlang contains a lot of syntactic sugar, the AST undergoes an intermediate transformation which produces the equivalent Core Erlang Syntax Tree [10]. Finally, this tree is transformed into BEAM bytecode.

2.3 Core Erlang

Core Erlang [10, 11] serves as an intermediate representation of the Erlang programming language. It contains a small set of constructs, offers clean semantics, and the conversion from Erlang to this representation as well as the transformation from Core Erlang to bytecode is quite straight forward. It can be printed in a textual representation that is easy to read and to edit, due to its simple grammar. Its most important use is the ability to develop tools acting on the source code representation (such as type analyzers [12, 13, 14], static analyzers for message-passing concurrency [15, 16, 17], testing tools [18, 19, 20, 21], code refactoring tools [22, 23, 24, 25], purity analysis tools [37], etc.), avoiding the complexity of full Erlang and its low-level BEAM bytecode representation.
The Erlang compiler itself can export a Core Erlang textual representation of the program. This can be inspected, edited and given back to the compiler for the creation of the BEAM bytecode. The recompilation of the edited Core Erlang program is necessary, since an interpreter of Core Erlang does not exist. However, there exists an interpreter that can run the ASTs stored in the debugging information of the BEAM representation of a module, given that this module has been compiled with the option to include such information. CutEr [7, 8], performs the concolic testing execution on the Core Erlang AST so we will explore its structure more.
The Core Erlang Abstract Syntax Tree is represented in the Erlang compiler both through records (in `/lib/compiler/src/core_parse.hrl`) and through abstract data types (in `/lib/compiler/src/cerl.erl`). Both representations are equivalent and can be used interchangeably. Moreover, there is an API defined in the `cerl` module, which enables the manipulation of an AST. The grammar, the lexical analysis of Erlang and the properties and significance of each AST node have been well documented in the literature [10, 38]. However, a general description of the Core Erlang AST is presented here too, as it will be useful when presenting the algorithm. Note that when referring to an **AST** from now on, we will mean the tree representation of a Core Erlang code.
Each AST comprises nodes, each of which represents an Erlang construct. We will use the

`module` node as an example (Fig. 2.1), which is the root node of a module's AST. Each node is a record with name `c_<node_name>` (e.g. `c_module`). The first field of every node record is its annotations. These may be populated by the compiler, but are extremely useful as they can be manipulated by external programs too, without altering the behavior of the program. In the case of the module node, the rest fields are:

- `name`: Contains the name of the module. It is a `literal` node.
- `exports`: Contains a list with the exported types and functions of the module.
- `attrs`: Contains the attributes added by the user in the module. They can contain type and signature definitions, as well as other arbitrary attributes. It is a list of two element tuples containing the name of the attribute and the attribute itself.
- `defs`: Containins the function definitions of the module.

```
-record(c_module, {anno = [], % annotation list
                   name,      % module name
                   exports,   % module expors
                   attrs,     % module attributes
                   defs}).    % module definitions
```

Figure 2.1: Module node definition.

Other important nodes for this work are the `values` node, the `let` node, the `letrec` node, the `case` node, the `clause` node and the `call` and `apply` nodes, which are all explained below.
The `values` node is essentially a collection of other nodes. Its definition is presented in Fig. 2.2.

```
-record(c_values, {anno=[]
                   es}).      % List of values (each one is a node)
```

Figure 2.2: Values node definition.

The `let` node (Fig. 2.3) has a `vars` field, which is a list containing the names of the variables bound in this let construct. The `arg` field represents the objects bound to those variables and it may be a `values` node, if this `let` introduces more than one variable. Last, the `body` field contains what is executed after the binding of the variables.

```
-record(c_let, {anno=[],
                vars,         % let variables
                arg,          % let argument
                body}).       % let body
```

Figure 2.3: Let node definition.

The `letrec` node (Fig. 2.4) is essentially similar to the `let` node. Its purpose is to introduce closures and bind them to a name. It contains a `defs` field, which is a list of two-valued tuples. Each tuple holds the name of the closure introduced and its definition. The `body` field contains the operations performed after the closures are introduced.

```
-record(c_letrec, {anno=[],
                   defs,      % function names with
                              % function definitions
                   body}).    % letrec body
```

Figure 2.4: Letrec node definition.

The `case` node (Fig. 2.5), apart from its annotations, consists of the case argument and the case clauses. The case argument can be a node representing some value, or a `values` node. In the latter case, each clause will have a number of patterns equal to the size of this `values` node, each one corresponding to a value of the `values` node.

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

Figure 2.5: Case node definition.

The `clause` node (Fig. 2.6), holds a list of patterns (in the field `pats`), a guard containing some assertions, and the `body` which holds all operations that will be executed if this clause is chosen. Note that not all Erlang expressions are permitted to act as a guard, but only a limited subset of them. The most common ones are various comparisons (e.g. `>`, `=:=`, etc.) and type assertions (eg. `is_integer(X)`).

```
-record(c_clause, {anno=[],
                   pats,      % clause patterns
                   guard,     % clause guard
                   body}).    % clause body
```

Figure 2.6: Clause node definition.

The `call` node (Fig. 2.7) represents a function call. It specifies the function to be called specifying the module it is defined in, its name, and its arguments.

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

Figure 2.7: Call node definition.

The `apply` node (Fig. 2.8) also represents a function call. In this case though, we are given the name of the function which can be a variable bound to a closure or a function defined in the same module. The node also holds the arguments of the function application.

```
-record(c_apply, {anno=[],
                  op,          % apply opperation
                  args}).      % apply arguments
```

Figure 2.8: Apply node definition.

2.4 Erlang Types

Erlang is a dynamically typed language and does not require any type annotations to be provided by the user. However, the type information of data exists in the runtime and is checked at each operation to ensure type safety. The programmer can give type annotations for documentation and clarity reasons. Moreover, these annotations can be used by external programs to analyze the code with a more specific context. The programmer can declare and define user defined types using the `type` or `opaque` attribute and define records with the `record` attribute. Furthermore, function signatures can be specified with the `spec` attribute. All these attributes will not be taken into account by the compiler, and exist to make the code development more structured and documented, as well as ease the debugging procedure through static analysis tools such as Dialyzer [12]. We present briefly the types in Erlang and how the user defined types and the function signatures can appear in the AST.

2.5 Built-in Types

Everything in Erlang is a term. A term can be of any type, predefined or user defined. The programmer can annotate something as a term using the `term()` or `any()` types. The predefined types are summed up in the following listing.

- *Integer*: It represents a whole number. To declare an integer type, we use `integer()` for any integer, `neg_integer()` for negative integers, `non_neg_integer()` for non-negative integers, `pos_integer()` for positive integers, the `<Lo>...<Hi>` notation for specifying a range of integers and just a constant for specifying a particular integer.

- *Float*: It represents a floating point number. To declare a float type, we use `float()`.
- *Atom*: An atom is a named constant. To declare an atom type, we use `atom()` or a specific constant atom.
- *Bitstring and Binary*: A bit string is a consecutive number of bits stored in an area of memory. In the case where this number is divisible by eight, it represents a binary. To declare a bistring or binary type, we use the `bitstring()` notation for any bistring and `binary()` notation for any binary. Additionally, the `<<>>` notation can be used, inside which there can be specifications concerning the size of the bistring.
- *Reference*: It represents a unique term in an Erlang runtime system. To declare a reference we use `reference()`.
- *Fun*: It represents a function type. To declare such a type, we use the `fun()` notation for any function. Additionally, we can specify the return type with `fun(...) -> Type` and furthermore, the argument types with `fun(Tlist) -> Type`.
- *Port identifier*: It is used to identify an Erlang port.
- *Pid*: A pid of a process. It uniquely identifies an Erlang process.
- *Tuple*: It is an ordered, fixed in size collection of types. To declare a tuple we use `tuple()` for any tuple or `{Type1, ..., TypeN}` to specify the types of the fields of the tuple.
- *List*: It is a compound data type with a variable number of elements. The elements can be of different types. It can be declared using `list()` for any list, `nonempty_list()` for non-empty lists, `[Type1|...|TypeN]` for constraining the types of the terms the list contains or `[Type1|...|TypeN, ...]` to further constrain the list to be non-empty.

Apart from these types, some additional types have been defined for syntactic sugar. These are:

- `number()`: Either float or integer.
- `char()`: Representing a character. It is an integer range from 0 to 1114111.
- `string()`: List of chars.
- `nonempty_string()`: A non-empty list of chars.

- *Record*: A record is a tuple with an atom as its first element. This atom is considered the name of the record and the rest elements are considered the fields of the record. A record can be declared using the `record` attribute and will be replaced by the corresponding tuple in the compilation.
- `boolean()`: It consists of the atoms `true` and `false`.
- `byte()`: An integer in the range 0 to 255.
- `node()`: An atom that represents the name of an Erlang virtual machine node.
- `module()`: An atom that represents an Erlang module.
- `mfa()`: A tuple representing an Erlang MFA which stands for Module, Function, Arity. It uniquely identifies a function defined in that module.

All types, if used in a user-defined type or a signature will appear in the AST in a specific form. These forms are gathered in Table 2.1. In those type forms, the line in the file on which the types were defined is present, is represented with `Ln`.
We add the miscellaneous types and the built-in compound types in Table 2.2, for clarity.

2.5.1 User-defined Types

The programmer can define types using the primitive ones as attributes in the Erlang program. The syntax for defining a type is:

```
-type <name>(<arguments>) :: <definition>.
```

The definition may consist of any of the primitive types explained above, and unions of them using the operator `|`. The arguments of a type are variables and can be used to constrain the definition of the type. For example, here we define a tree type with its nodes containing data of an arbitrary type.

```
-type tree(X) :: {X, tree(X), tree(X)} | nil.
```

The AST representation of a user defined type is an attribute in the module with the following form:

```
{{c_literal,[Ln],type},
 {c_literal,[Ln],[{<Tname>, <Definition>, [<Variables>]}]}}
```

where `<Tname>` corresponds to the type name, `<Definition>` to its definition and `[<Variables>]` to the names of the variables that the type uses. The `tree` type example has this corresponding representation in the AST:

```
{{c_literal,[Ln],type},
      {c_literal,
          [Ln],
          [{tree,
               {type,Ln,union,
                   [{type,Ln,tuple,
                        [{var,Ln,'X'},
                         {user_type,Ln,tree,[{var,Ln,'X'}]},
                         {user_type,Ln,tree,[{var,Ln,'X'}]}]},
                    {atom,Ln,nil}]},
               [{var,Ln,'X'}]}]}}
```

We see that the variable `X` is used in the definition with the form `{var,Ln,'X'}`. This type can then be used in another type or signature by specifying the type that the tree nodes will contain (e.g. `tree(integer())`). The form through which a user-defined type is specified is seen in the above example too and is `{user_type,Ln,<Tname>,[<Variables>]}`. Instead of the `type` attribute, the `opaque` attribute can be used. The only difference between those two attributes is that the `opaque` one is meant for type definitions of terms whose internal structure is not supposed to be known and inspected outside of their defining module.

2.5.2 Function Signatures

The programmer can declare the signature of a function via the `spec` attribute with this syntax:

```
-spec <Name>(<Args>) -> <Ret>.
```

where `<Name>` is the name of the function, `<Args>` the types of its arguments and `<Ret>` the return type. There is also an option, mainly for documentation purpose, to introduce constraints through variables in such a signature. An example of such a constraint is:

```
-spec f(N) -> N when
    N :: integer().
```

where we constraint the value of the variable N to be an integer.

Class	Type	Form
Numeric	`integer()`	`{type, Ln, integer, []}`
	`<Int>`	`{integer, Ln, <Int>}`
	`<L>...<H>`	`{type, Ln, range, [L, H]}` where `L :: {integer, Ln, <L>}`, `H :: {integer, Ln, <H>}`
	`neg_integer()`	`{type, Ln, neg_integer, []}`
	`non_neg_integer()`	`{type, Ln, non_neg_integer, []}`
	`pos_integer()`	`{type, Ln, pos_integer, []}`
	`float()`	`{type, Ln, float, []}`
Tuples	`tuple()`	`{type, Ln, tuple, any}`
	`{}`	`{type, Ln, tuple, []}`
	`{Type1, ... TypeN}`	`{type, Ln, tuple, [T1,...,Tn]}` where `Ti` is a valid type expression
Lists	`list()`	`{type, Ln, list, []}`
	`nonempty_list()`	`{type, Ln, nonempty_list, []}`
	`[]`	`{type, Ln, nil, []}`
	`[Type]`	`{type, Ln, list, [T]}` where `T` is a valid type expression corresponding to `Type`
	`[Type, ...]`	`{type, Ln, nonempty_list, [T]}` where `T` is a valid type expression corresponding to `Type`
Bitstrings	`bistring()`	`{type, Ln, bistring, []}`
	`binary()`	`{type, Ln, binary, []}`
	`<<_:M>>`	`{type, Ln, binary, <M>, <N>}` where `<M> :: {integer, Ln, M}`, `<N> :: {integer, Ln, 0}`,
	`<<_:_:*N>>`	`{type, Ln, binary, <M>, <N>}` where `<M> :: {integer, Ln, 0}`, `<N> :: {integer, Ln, N}`,
	`<<_:M,_:_:*N>>`	`{type, Ln, binary, <M>, <N>}` where `<M> :: {integer, Ln, M}`, `<N> :: {integer, Ln, N}`,
Atoms	`atom()`	`{type, Ln, atom, []}`
	`<Atom>`	`{atom, Ln, <Atom>}`

Table 2.1: Types with their corresponding AST representations. `Ln` corresponds to the line in the file where the type is declared.

Type	Form
`term()`	`{type, Ln, term, []}`
`any()`	`{type, Ln, any, []}`
`none()`	`{type, Ln, none, []}`
`no_return()`	`{type, Ln, no_return, []}`
`Type1 | ... | TypeN`	`{type, Ln, union, [T1,...,Tn]}` where `Ti` is a valid type expression
`reference()`	`{type, Ln, reference, []}`
`port()`	`{type, Ln, port, []}`
`pid()`	`{type, Ln, pid, []}`
`number()`	`{type, Ln, number, []}`
`char()`	`{type, Ln, char, []}`
`string()`	`{type, Ln, string, []}`
`nonempty_string()`	`{type, Ln, nonempty_string, []}`
`boolean()`	`{type, Ln, boolean, []}`
`byte()`	`{type, Ln, byte, []}`
`node()`	`{type, Ln, node, []}`
`module()`	`{type, Ln, module, []}`
`mfa()`	`{type, Ln, mfa, []}`

Table 2.2: Miscellaneous types and aliases of types with their corresponding AST representations. `Ln` corresponds to the line in the file where the type is declared.

Chapter 3

Concolic Testing

3.1 The Basics of Concolic Testing

The word “concolic” is a result of merging the words “concrete” and “symbolic” representing the dual representation of values during the interpretation. In concolic testing, both concrete and symbolic execution is performed for a high path coverage. The symbolic part of the execution is necessary to create the new inputs via constraint solving and the concrete execution is needed to guide the symbolic execution on a concrete path [26, 5, 7, 8, 27, 28]. During execution, both a symbolic and a concrete state is kept. In the concrete environment, we have mappings from variables to their concrete values, whereas in the symbolic environment, we have mappings from variables to their symbolic values. For the process to start, a seed input is needed, either randomly generated or user provided. Then, the program is interpreted with this seed input producing an execution path, while keeping the constraints from the symbolic environment at each branching statement. The conjunction of all those constraints forms a logical expression, which will lead any input satisfying it along the same execution path. Thus, if any term of this formula corresponding to a conditional statement is negated, an input satisfying the new formula will explore a different path, diverging from the previous one at this conditional statement. Keeping all the execution paths and negating all possible terms corresponding to all the branches in those execution paths will inevitably cover the whole search space of possible inputs. In order to find an input satisfying any new predicate created, a constraint solver is used such as lp_solve, ECLiPSe [29] or Z3 [30].

3.2 Challenges and Limitations

Concolic testing allows for thorough testing and exploits the internal view of the program in order to efficiently find inputs producing runtime errors. However there are various limitations to this technique and many proposed methods to tackle them.
The most intractable of these challenges is the incompleteness of the constraint solving. Depending on the input program, a constraint generated from an execution path may

require a lot of time to be solved by the constraint solver, which may even be unable to solve it. This renders some programs impossible to explore, as new inputs will not be able to be created even for some feasible execution paths [6, 5].

Another important challenge is the search space explosion, which is the main focus of this thesis. Regardless of the program size, the trace data that will be produced can likely be too many. Some paths can even be very long and create large predicates which are hard to be handled by the constraint solver. Despite this, we need to reach the possible runtime errors in a reasonable amount of time, so we need to steer the concolic execution down certain paths using various heuristics, with some of them presented in Section 3.3.

Last, many programs may have a non-deterministic behavior (eg. using random numbers). This is a problem, because the execution path may not relate solely to the predicate produced by the symbolic execution, and thus, prevent the algorithm from creating inputs to explore the branches that it chooses.

Specifically in Erlang, which is a language aimed towards concurrent applications, concolic execution can be more complex [39].

3.3 Search Space Exploration

To tackle the problem of the search space explosion in concolic testing, many approaches have been proposed. The idea is to guide the symbolic execution in branches in a certain order which will reach the bug in the program faster. The basic approach is to do a bounded DFS or BFS search in the branches encountered which will inevitably find all runtime errors but has to explore the whole search space. More advanced heuristics are presented in the following subsections.

Random Search

One of the simpler heuristic to traverse the search space is the Random Branch Search heuristic [31]. This algorithm selects for inversion a random branch from the previous execution path. Despite its simplicity, it performs better than DFS or BFS. It even surprisingly performs better, in certain cases, than other more advanced heuristics [32].

Control Flow Directed Search

Control-Flow Directed Search [31] uses the structure of the program to guide the dynamic search towards paths that haven't been previously discovered. It uses a weighted call and control flow graph of the program. The weights are adjusted while uncovering new paths, giving smaller weights to edges leading to unexplored branches. Then, the condition that leads to the closest unexplored branch is selected to be inverted. This method performs a better branch coverage per iteration than the ones presented so far.

CarFast

CarFast [33] is a strategy exploiting coverage information to try and reach as many branches as possible in a greedy manner. It always chooses for inversion a branch whose alternative hasn't been explored yet. Additionally, it prioritizes branches depending on a score value, which is computed based on the number of statements that will be coverable through those branches.

Context-Guided Search

Context-Guided Search [34] is in its core a BFS search on the execution tree. During the search, it excludes branches whose "contexts" have already been explored. The context of a branch is, in this case, defined as a sequence of preceding branches in the execution path containing the branch in question. For a new depth d to be explored, all branches of this depth are gathered. The algorithm continues the search for each candidate branch if their context is not already considered. Then it proceeds with the nodes of depth $d + 1$.

Generational Search

Generational Search is a strategy used in SAGE [40], an Automated Whitebox Fuzz Testing tool based on symbolic execution [35]. It uses an "incremental coverage gain" to guide the concolic execution. Additionally, it does not select one branch at a time, but a whole path and creates a new "generation" of branches consisting of all the opposite branches of the selected path. Inputs are generated to search them, creating a number of new paths. The path with the greatest "coverage gain", which depends on the newly discovered statements through this path's execution, is selected to create the next generation of inputs.

3.4 CutEr

The method proposed in this work is built upon CutEr [7, 8, 38]. CutEr is a concolic testing tool for Erlang. It acts upon the Core Erlang AST with a custom concolic (both concrete and symbolic) interpreter and uses Z3 [30] as a constraint solver.
Most concolic testing tools act on a low level representation of the program. For instance, KLEE [41] acts on LLVM code while SAGE [40] on x86 bytecode. This method certainly has its benefits, as this low level representation is optimized by compilers and it is efficient for languages like C and Java whose type system is similar on both the original code and the compiled one. Erlang's BEAM format loses the rich type information that the constraint solver needs and this is why CutEr uses core Erlang AST.
The choice of the Z3 constraint solver was based upon its extra ability to handle recursive data types natively. Apart from that, it is a powerful SMT solver that supports various datatypes, extensional arrays, fixed-size bit-vectors, quantifiers and arithmetic.
CutEr's interpreter uses four types of processes during the execution [7]:

- Interpreter processes, where the execution takes place.
- Supervisor processes, that oversee the execution.
- Code servers, where the core Erlang ASTs are computed and stored.
- Monitor servers, which monitor interpreter processes and gather occurred exceptions.

Regarding the heuristic with which the search space is explored, CutEr uses by default a bounded BFS. The depth of the search can be set by the user. Additionally, the search strategy is implemented as a separate module in a way that easily enables the creation and use of a new heuristic.
Our method is implemented in two of these processes. The static analysis part is run by the code server process, when obtaining the ASTs of the modules reachable by the entry point of the concolic testing. The dynamic check is developed in the interpreter process.

Chapter 4

Search Space Reduction via Safe Branch Pruning

4.1 Overview

Safe branch pruning is a technique that is used to prevent the concolic testing tool from exploring unneeded paths. The algorithm that performs this task consists of a static analysis and a dynamic check performed during concolic execution. The static analysis annotates each node of the abstract syntax tree (AST) of all the functions that constitute the unit as safe or unsafe and the dynamic check uses these annotations to decide which branches should be excluded from the search. A simple flowchart of the static analysis is presented in Fig. 4.1. Each part of the diagram will be explained in Section 4.3.
The annotated ASTs of the functions contain in each node an annotation called `maybe_error` with values `true`, `false` or `type_dependent`. The value `true` means that running the subtree of the AST rooted by this node can possibly produce an error. The value `false` means that no error can happen regardless of input. Lastly, the value `type_dependent` means that if the function is called with arguments conforming to its signature, then it will not produce a runtime error in this subtree. Additionally, the following annotations might exist in a node:

- `type_dependent_unreachable`: Present only in `clause` nodes that will not be reached if the function is called with arguments satisfying its signature.
- `force_constraint_logging`: Boolean value. When true, regardless of the `maybe_error` annotation, one must explore this subtree.
- `distrust_type_dependent`: Boolean value. When true, we do not trust the nodes with `maybe_error` value of `type_dependent`.

Note that the analysis tries to be *pessimistic*, meaning that, when it is not sure about a node, it will be marked with the `maybe_error` annotation as true. This way the ability of the tool to locate all runtime errors is not hindered.

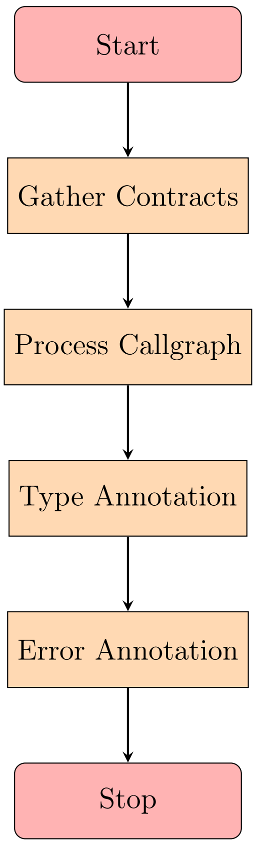

Figure 4.1: Static Analysis Flowchart

The dynamic check will be presented first to show the function of each of these annotations and how they behave. After that, we will present each of the steps of Fig. 4.1.

4.2 Dynamic Check

The dynamic part of the algorithm is added in the interpreter. It introduces some options to guide the constraint logging of the various branches through an extra parameter map. Stopping the constraint logging at some point of the execution path, will not let the constraint solver consider any proceeding branches to be explored. This is how the safe branch pruning is performed. Whenever we reach a branch that we want to prune, we stop the constraint logging for the rest of the execution. The parameter map contains three boolean flags:

1. `constraintLogging` initialized to `true`
2. `isForced` initialized to `false`
3. `distrustTypeDependent` initialized to `false`

The logging functions take the first two of those flags under consideration before logging anything. If `isForced` is true then everything else is ignored and logging proceeds normally.

On the opposite case, only if constraintLogging is true then logging proceeds. The distrustTypedependent flag alters the behavior of retrieving the maybe_error annotation of a node according to Table 4.1.

maybe_error / distrustTypedependent	**true**	**type_dependent**	**false**
true	true	false	false
false	true	true	false

Table 4.1: distrustTypedependent effect on maybe_error. The dynamic check will need a boolean value of the maybe_error annotation, which originally can take 3 values (true, false, type_dependent). The distrustTypedependent flag decides whether the type_dependent value will be perceived as false or true.

Essentially, when this flag is true, the type_dependent value of maybe_error will be considered by the dynamic check as true, else as false.
The rest of the dynamic check takes place when encountering certain Core Erlang nodes, where we adjust the values of those flags:

- apply/call: If the node is marked with the distrust_type_dependent annotation, we set distrustTypeDependent flag to true.
- let: If the node is marked with the force_constraint_logging annotation, we set isForced flag to true.
- case: If isForced is true, or maybe_error (depending on distrustTypeDependent) is true, the argument of the case is called with the option isForced as true else as false. After that, the clause that should be chosen is identified. Lastly if isForced is false and maybe_error of the clause is also false, then it sets the option constraintLogging to false.

Note that the only place that we disable the constraint logging happens in a case node, where we can have such a node that can produce an error, but the currently chosen branch is safe for execution.
The logging function now takes under consideration the constraintLogging flag and only logs when it is true. This way, the safe branches will not be considered for exploration.

4.3 Static Analysis

4.3.1 Contract Gathering

In Erlang, contracts are the signatures of functions that can be defined by the programmer using the -spec attribute [42]. The analysis needs the function contracts in some form supporting various type manipulations. For this reason, the module erl_types was chosen.

It has a full Erlang type representation and supports operations on types, such as finding the least possible supertype of some types and type subtraction.

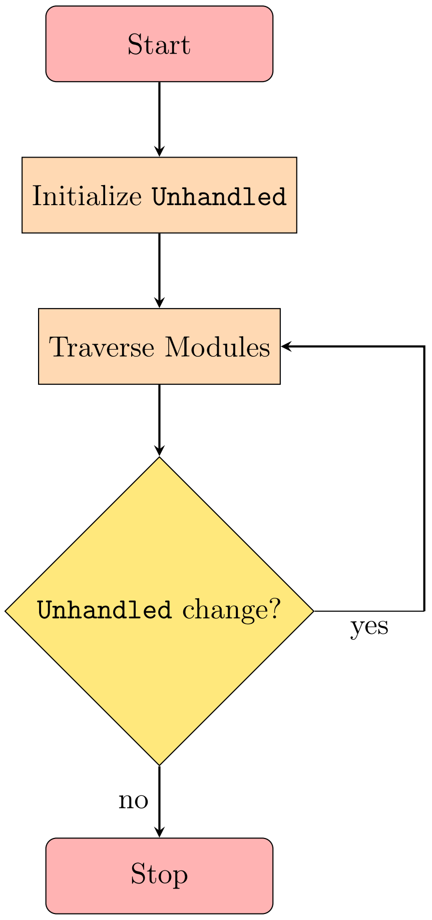

Figure 4.2: Contract Gathering Flowchart. `Unhandled` is a set initialized to hold all functions defined in the modules that are involved in the callgraph. Traversing the modules will convert signatures to `erl_type` form and remove these functions from the `Unhandled` set.

Function signatures have a certain form in the AST of the module, which is different than the `erl_types` one. Thus, there is a need for a conversion from the one form to the other. The `erl_types` API provides a function `erl_types:t_from_form/6` which converts such a form to an `erl_type`. It needs to be provided with various data apart the form itself, such as the exported types of all the modules needed by this form and the user defined types of the module. If it does not have enough information, it fails to produce an `erl_type`. To use it, we start by gathering the modules that will be reached by the entry point. We also initialize a set called `Unhandled` with all the MFAs in those modules. For each module traversed, we first try to convert its user defined types and records. Secondly, we try and convert the contracts and, if the conversion succeeds, we remove them from the `Unhandled` set. This is done until `Unhandled` remains unchanged in a whole traversal of these modules. This way, we can handle types whose dependencies are defined later in a

module or in different modules.
Two other forms that `erl_types:t_from_form/6` can't handle are records and types with constraints declared using `when` like the one in Fig. 4.3. Records though are not really present as entities in the ASTs of functions. A record is replaced by the compiler with a tuple having as values the record's name and its fields. In order to handle records, we replace the aforementioned form with a temporary type declaration before giving it as input in `erl_types:t_from_form/6` and keep it stored. When encountering any type or contract that refers to this record, the reference gets replaced by its equivalent temporary type. For types with guards, we run a preprocessing step in the form which transforms this type to an equivalent one with no guards and then it can be given to `erl_types:t_from_forms/6`.

```
-spec keyfind(Key, N, TupleList) -> Tuple | false when
      Key :: term(),
      N :: pos_integer(),
      TupleList :: [Tuple],
      Tuple :: tuple().
```

Figure 4.3: Signature with constraints. It is the signature of the `lists:keyfind/3` function of stdlib.

4.3.2 Callgraph Processing

In order to annotate a function successfully, we have to know whether the calls to other functions can produce errors. This creates the need to annotate them in a specific order. This order requires a callgraph to be computed. The callgraph is a directed graph that has nodes corresponding to functions and edges corresponding to function calls pointing from the caller to the called function. Only the functions reachable via a sequence of calls originating from the entry point of concolic testing constitute the callgraph. This callgraph is computed with the use of the `xref` module which is the cross reference tool of Erlang, and can be used for finding various dependencies between functions and modules. We start by adding to the `xref` server module of the entry point and ask for the edges of the entry point to other functions. We then continue this process recursively for their modules. When we don't have any more functions to explore, we have created the callgraph as we defined it.
In the simple case where this graph is acyclic, and thus represents a tree, we can annotate each function starting from the leaves of the tree in a depth first search (DFS) order. This way, we can know for each function whether any other called function is error-free or not, since the subtree starting from this function is already annotated. However, this is not the case for most callgraphs, especially in Erlang, which is a functional programming language and uses recursion for many computations. A recursive function alone is a cycle containing one node which points to itself. Larger cycles may exist through mutual recursion. More-

over, cycles may have common nodes creating a strongly connected component (SCC) in the callgraph.

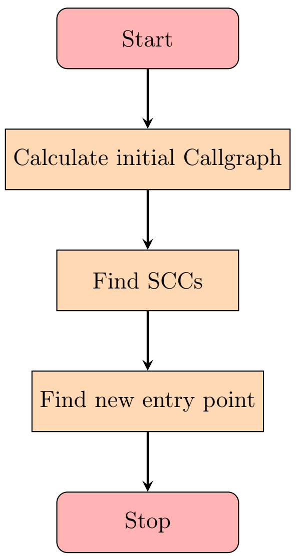

Figure 4.4: Graph Processing Flowchart. Initial Callgraph is a directed graph that possibly contains SCCs. We find the SCCs and merge the ones that have common nodes. If the entry point is part of an SCC then the whole SCC becomes the new entry point.

To tackle this problem we first define what will be done when the functions that we want to annotate all belong to an SCC. In this case, we run a fix-point computation and in every function annotation we exclude the calls to functions that we have not already processed but do belong to the SCC. While doing this, we check if any annotations have changed or any unexplored functions in the SCC have been encountered. If this is true for any of the functions in the SCC, we run the analysis again to the whole SCC until fixpoint. If two SCCs in the callgraph have at least one common node, then we have to run this fix-point computation for the nodes of both SCCs simultaneously like they belonged to the same SCC.

Knowing how to handle SCCs, we run a preprocessing step on the graph where all the SCCs are identified and replaced by single nodes. In the new graph, the nodes are of the form `{function, <function name>}` or `{scc, [<list of functions in SCC>]}`. Each edge that previously pointed to any of the functions in the SCC, now points to the `SCC` node and each edge that had one of these functions as a starting point, now has the `SCC` as its starting point. This algorithm also returns the new entry point node that might have changed due to belonging to an SCC. This procedure is depicted in the flowchart of Fig. 4.4.

The new graph is a directed acyclic graph, and will be processed in a DFS order like previously starting from the leaves and going towards the root. If the node processed is just a function, we run the default error annotation (Section 4.3.4) and if it is an `SCC` we run the fix-point computation.

The module in Fig. 4.5 poses a good example of this graph transformation.

```erlang
-module(funs1).
-export([f5/1]).

f1(X) -> % error-free
  case X of
    3 -> f2(2);
    2 -> f2(1);
    _ -> 1
  end.

f2(X) -> % error-free
  case X of
    3 -> f1(2);
    2 -> f1(1);
    _ -> 1
  end.

f3(X) -> % not error-free
  case X of
    3 -> f4(2);
    2 -> f4(1);
    1 -> 1
  end.

f4(X) -> % not error-free
  case X of
    3 -> f4(2);
    2 -> f3(1);
    1 -> 1
  end.

f5(X) ->
  case f1(X) of
    1 -> f3(X);
    _ -> 1
  end.
```

Figure 4.5: Callgraph with SCCs example. `f3/1` and `f4/1` belong to two SCCs which are merged into one. `f3/1` and `f4/1` constitute another SCC.

Here we have two mutually recursive functions, where one of them additionally calls itself. The entry point is not part of an SCC. The simple SCCs here are:

- `[funs1:f1/1, funs1:f2/1]`
- `[funs1:f3/1, funs1:f4/1]`
- `[funs1:f4/1]`

The two latter SCCs, which have a common function, get merged and belong to the same node in the resulting graph. The starting and the final callgraph are shown in Fig. 4.6.

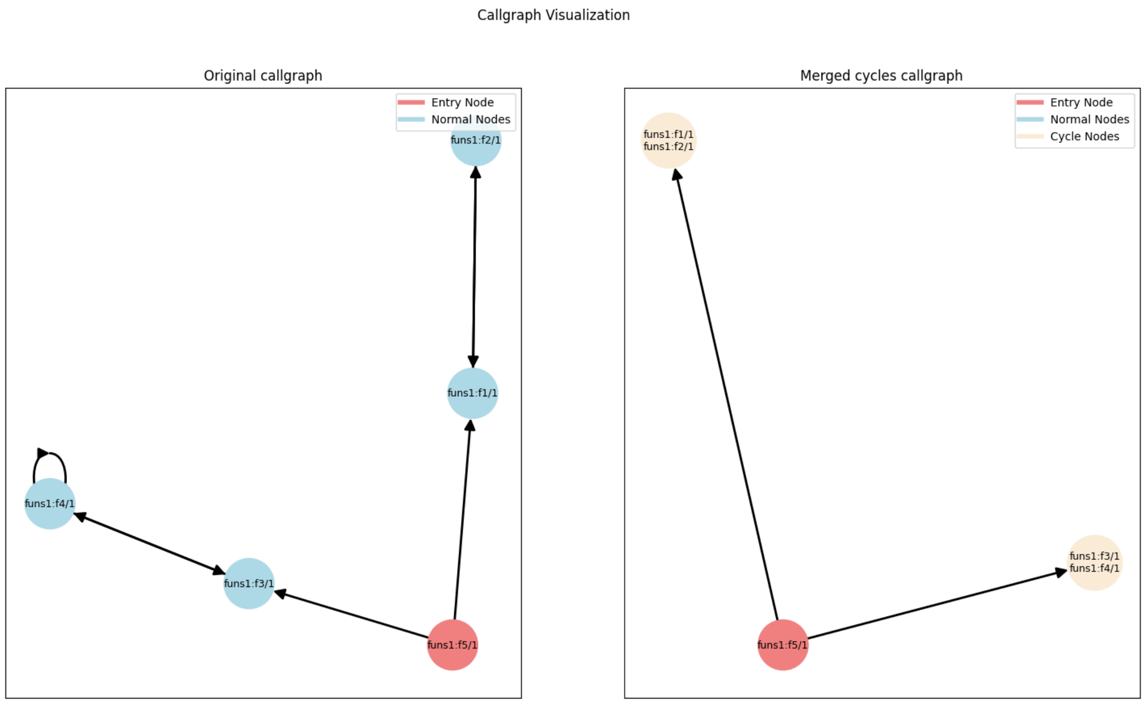

Figure 4.6: Callgraph Processing. On the **left** figure we present the original callgraph. On the **right** figure we present the resulting callgraph after merging the SCCs.

4.3.3 Type Annotation

This type annotation process has the purpose of finding the type of each AST function node if its arguments satisfy its contract. This is performed in order to leverage the type data and further improve branch pruning. Without considering type information, our ability to classify an Erlang program as error-free would be very limited. For example, a simple addition cannot be expected to be error-free without knowing the types of its arguments. However, modern Erlang uses type and function signature attributes in the program which, enables the use of tools (such as the Dialyzer) to point out type discrepancies in them. A programmer using these tools, can make sure that the function signatures provided are correct. Thus, we can make our analysis find more safe parts of the program to prune given that we trust the function contracts. A very simple example is depicted in Fig. 4.7, where function `f/1` adds its argument to the constant integer 1. Without considering the types of the arguments of `f/1` the input `[]` produces an error. If we trust the signature of this function though, we can prove that this function would never fail.

```
-spec f(integer()) -> integer().
f(X) ->
  X + 1.
```

Figure 4.7: Simple Addition

The type annotation of each AST node is happening by traversing the function ASTs and propagating the node's specified types. This can be achieved by using a table, namely the symbol table, containing the signatures of other functions and the variables introduced as arguments or in the function's body.
The algorithm traverses the AST in DFS order again and starts annotating the leaves of the AST with their type. Of course, this can not be deduced for every part of the AST nor for every possible function. To be conservative, we ignore those nodes and don't annotate them at all in order for error annotation algorithm (Section 4.3.4) to ignore the type data for them and mark them safely as error-free or not. This traversal in some cases cannot happen only once, because not all functions have a contract. This does not mean that we cannot know their signature though. If we inspect the code of `lists:last/1` (Fig. 4.8), we see that `lists:last/1`, which has a contract, calls lists:last/2 which does not. Such a pattern is common in functional programming languages such as Erlang. However, `lists:last/2` not only is not exported, but is only called by `lists:last/1`. This means that if we know the types of the arguments of `lists:last/2` when it is called by `lists:last/1`, we can use this as its contract and analyze it normally after that.

```
-spec last(List) -> Last when
      List :: [T,...],
      Last :: T,
      T :: term().

last([E|Es]) -> last(E, Es).

last(_, [E|Es]) -> last(E, Es);
last(E, []) -> E.
```

Figure 4.8: lists:last

To do this, we start by initializing a set called `NoSpec` containing all those functions in the callgraph that do not have a signature. We also initialize an open set in the form of a queue that contains all the functions that do have a signature. While the open set is not empty, we take the first element and apply the type traversal on it. During that, if we encounter any calls to a function in `NoSpec`, we keep that call's argument types. When the traversal is over, we create the signature of the called functions by finding the supremum of their argument types. This is done because a function may be called with different types in each call instance. In Fig. 4.9, `reverse/2` is called (1) by `reverse/1` with an empty list as a second argument and (2) by itself with a non-empty list. In this case we have to handle `reverse/2` as if it would take a non constrained list as an argument. If the newly created signature is different than the previous one (which does not exist initially) then we put this called function at the end of the open set, and the caller after that. This way we will traverse the called function with its new signature first and then its caller. Additionally, if we encounter any change in the type annotations of any node in a function, we put it in the open set too. Lastly, if the function we were traversing is part of the `NoSpec` set, then we check if we were able to deduce the type returned by the function and if yes, we update its signature.

```
-spec reverse(List1) -> List2 when
      List1 :: [T],
      List2 :: [T],
      T :: term().
reverse(L) ->
  reverse(L, []).

reverse([], Acc) -> Acc;
reverse([H|T], Acc) ->
  reverse(T, [H|Acc]).
```

Figure 4.9: Different argument types at call

This fix-point computation is presented in the flowchart of Fig. 4.11.

We also have to take closures under consideration. Closures are introduced via a `letrec` node in the AST and are bound to a variable. Defining closures is really common and is even done automatically in case we have a list comprehension. In Fig. 4.10 we use a simple list comprehension to add 1 to each element of a given list.

```
-spec f([number()]) -> [number()].
f(X) ->
  [Y + 1 || Y <- X].
```

Figure 4.10: List Comprehension

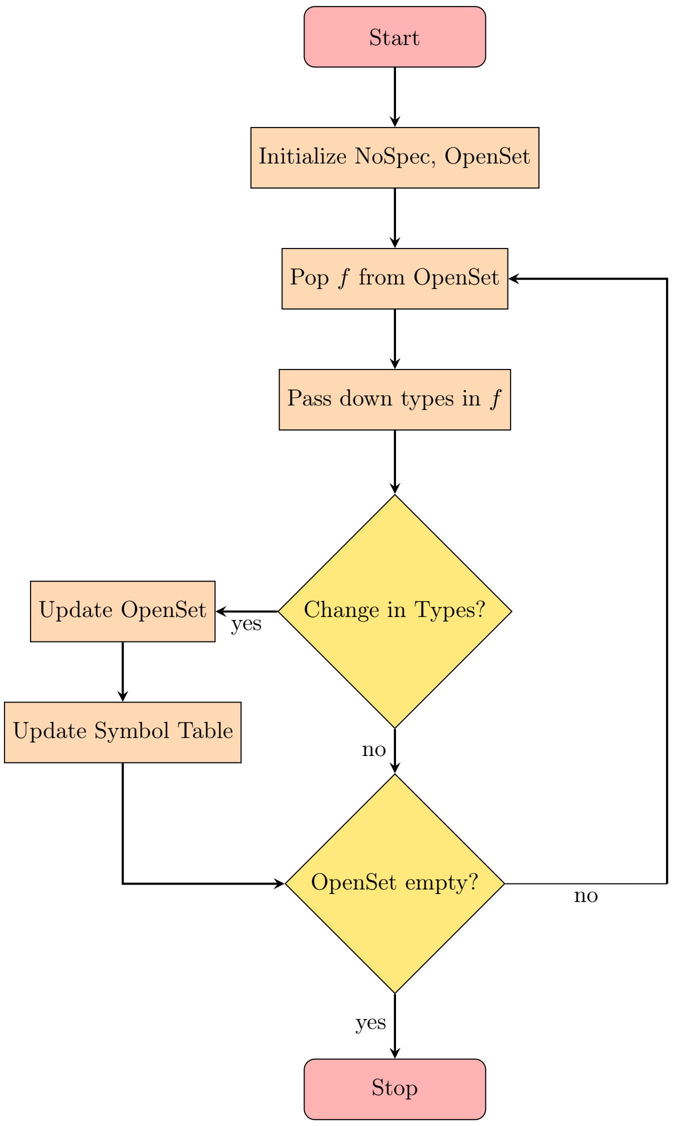

Figure 4.11: Type annotation algorithm flowchart. NoSpec is a set containing all functions with no user provided signature. OpenSet is initialized to hold all functions with a user defined signature. The symbol table contains all the signatures of functions and can contain signatures computed for functions in the NoSpec set. A change in types might be an encounter of a call to function in the NoSpec set with argument types different than the so far computed for this function or a change in any node type computed. Updating the open set includes pushing the f in the open set and any functions belonging to NoSpec that are called and have to get an updated signature (done in process "Update Symbol Table")

In the AST of the function shown in Fig. 4.12 we see that it this done by introducing a closure in the `letrec` node in line 4. This closure is also recursive as it calls itself in line 17. As we do not know its signature yet, we have to create it similarly to how we found the signatures of functions with no signatures.

```
 1  {c_fun,
 2    [{function,{f,1}},5],
 3    [{c_var,[],0}],
 4    {c_letrec,[],
 5      [{{c_var,[],{'lc$^0',1}},
 6          {c_fun,[],[{c_var,[],3}],
 7            {c_case,[],{c_var,[],3},
 8              [{c_clause,[],[{c_cons,[],{c_var,[],'Y'},{c_var,[],2}}],
 9                  {c_literal,[],true},
10                  {c_let,[],[{c_var,[],5}],
11                    {c_call,[],
12                      {c_literal,[],erlang},
13                      {c_literal,[],'+'},
14                      [{c_var,[],'Y'},
15                        {c_literal,[],1}]},
16                    {c_let,[],[{c_var,[],6}],
17                      {c_apply,[],
18                        {c_var,[],{'lc$^0',1}},[{c_var,[],2}]},
19                      {c_cons,[],{c_var,[],5},{c_var,[],6}}}}},
20                {c_clause,[],[{c_literal,[],[]}],
21                  {c_literal,[],true},
22                  {c_literal,[],[]}},
23                {c_clause,[compiler_generated],[{c_var,[],4}],
24                  {c_literal,[],true},
25                  {c_primop,[],{c_literal,[],match_fail},
26                    [{c_tuple,[],
27                        [{c_literal,[],function_clause},
28                          {c_var,[],4}]}]}}]}}}],
29      {c_apply,[],
30        {c_var,[],{'lc$^0',1}},[{c_var,[{function,{f,1}},5],0}]}}}
```

Figure 4.12: List Comprehension AST. In line 5 the closure `{'lc\$^0',1}` is introduced which is recursive. It calls itself in the `apply` node in line 17.

To do this, we perform a fix-point computation when we encounter a `letrec` node with its own `NoSpec` set, called `LetRecSpec`. Inside it, when we find a call to something in the `LetRecSpec` we keep the types of its arguments of that call to process them the same way. When we return to the `letrec` node, we check whether any of these encountered calls to functions in the `LetRecSpec` correspond to a closure defined in this specific `letrec` node and if yes, we update their signatures and traverse them again. This check is provoked by nested `letrec` nodes. When we finally find the signature of the defined closure, we keep it in a table which will be passed in the once again traversed function by the outer fix-point computation. This is done so that when we reach the same `letrec` node, we will know the signature of the closure and thus be able to compute the type of some nodes again with types that don't match the already existing ones. This results in a change that puts the function in the open set again producing an infinite loop.

This type-propagation algorithm also tries to find `case` nodes with unreachable clauses given the signature of a function. This is important because the Erlang compiler adds a "catch all" clause to all `case` nodes that do not already have one, in which it calls the `match_fail` primop. An example illustrating this is presented in Fig. 4.13. Since `f/1` accepts a `boolean` as an argument, there is no input conforming to this signature that will not reach one of the two `case` clauses in line 3. It will be added by the compiler s

```
1  -spec f(boolean()) -> atom().
2  f(X) ->
3    case X of
4      true -> ok;
5      false -> notok
6    end.
```

Figure 4.13: In this example, an unreachable clause will be added by the compiler, since it doesn't consider that `X` will be of type `boolean` and thus will have values `true` or `false`.

However, the Erlang compiler, since it does not consider signatures, adds a "catch all" `clause` seen in line 29 in Fig. 4.14 to raise an error in case the argument does not match some of the previous clauses. If we let the error analysis to annotate this AST, it will mark the whole `case` as unsafe because it contains this unsafe `clause`.

To resolve this, we mark the unreachable clauses during the type analysis using the annotation `type_dependent_unreachable` in order for the next analysis to not take them under consideration.

The algorithm processes the type t initialized as the `case` argument type. For every `clause` it checks whether t is `none()`. If it is, then this clause will not be reached and it is marked. After this, the new t is constructed by subtracting the type represented by the pattern and guards of the `clause`. If the union of the first n types of the first n `clause` nodes are a supertype of the type of the `case` argument, then there can't be any input to this `case` that will reach the next clauses.

There remains the problem of deducing the type to be subtracted given the patterns and guards of each clause. The type of the argument of the `case` node can be just a variable or a `values` node. In the latter case, there exist several patterns in each clause, where each one of them corresponds to a variable in the `case` argument like in Fig. 4.15 with its AST shown in Fig. 4.16. We will examine the simpler case where the argument is just a variable.

```
 1  {c_fun,
 2      [{function,{f,1}},5],
 3      [{c_var,[5],0}],
 4      {c_let,[],
 5          [{c_var,[],957}],
 6          {c_var,[{function,{f,1}},5],0},
 7          {c_letrec,[],
 8              [{{c_var,[],{match_472,0}},
 9                {c_fun,[],[],
10                    {c_let,[],
11                        [{c_var,[],1}],
12                        {c_var,[],957},
13                        {c_primop,
14                            [6],
15                            {c_literal,[],match_fail},
16                            [{c_tuple,[],
17                                 [{c_literal,[],case_clause},
18                                  {c_var,[],1}]}]}}}}],
19              {c_case,[],
20                  {c_var,[],957},
21                  [{c_clause,[],
22                       [{c_literal,[],true}],
23                       {c_literal,[],true},
24                       {c_literal,[8],ok}},
25                   {c_clause,[],
26                       [{c_literal,[],false}],
27                       {c_literal,[],true},
28                       {c_literal,[10],notok}},
29                   {c_clause,[], % compiler generated
30                       [{c_var,[],600}], % matches everything
31                       {c_literal,[],true},
32                       {c_apply,[],{c_var,[],{match_472,0}},[]}}]}}}}
```

Figure 4.14: Unreachable clause AST. The unreachable clause is in line 29, added by the compiler.

```
-spec f(boolean(), boolean()) -> atom().
f(true, true) -> true;
f(false, true) -> false.
```

Figure 4.15: Values as argument in case

```
 1  {c_fun,
 2      [{function,{f,2}},5],
 3      [{c_var,[5],0},{c_var,[5],1}],
 4      {c_case,
 5          [{function,{f,2}},5],
 6          {c_values,
 7              [5],
 8              [{c_var,[5],0},{c_var,[5],1}]},
 9          [{c_clause,
10               [5],
11               [{c_literal,[5],true},
12                {c_literal,[5],true}],
13               {c_literal,[],true},
14               {c_literal,[5],true}},
15           {c_clause,
16               [6],
17               [{c_literal,[6],false},
18                {c_literal,[6],true}],
19               {c_literal,[],true},
20               {c_literal,[6],false}},
21           {c_clause,
22               [compiler_generated],
23               [{c_var,[],3},{c_var,[],2}],
24               {c_literal,[],true},
25               {c_primop,
26                   [5],
27                   {c_literal,[],match_fail},
28                   [{c_tuple,
29                        [5],
30                        [{c_literal,[],function_clause},
31                         {c_var,[],3},
32                         {c_var,[],2}]}]}]}]}}
```

Figure 4.16: Values as argument in case. The case statement in line 4 has as argument a `values` node (line 6) which contains the two arguments of the function, `0` and `1`.

In this case we will have one pattern and a guard in each `clause`. Not all guards can be considered by this analysis. Currently are supported only guards of the form `has_type(X)` (e.g.`is_integer(X)`), where we bound a variable to some type. For each `clause` we will consider firstly the guard and secondly the pattern. If we encounter a guard that is not supported then we assume that this clause represents the type `none()`. If it is supported, the guard will be of the form `has_type(X)` and may be referring directly to the argument of the case or in a variable found in the pattern of the `clause`. In the first case, we assume that this `clause` corresponds to the type that the guard is constraining the argument to be. In the latter case, we add the variable that the guard is referring to with the type that the function is constraining it to be, in a separate symbol table. Then we consider the pattern of the `clause` and convert it to an `erl_type`. In this conversion, all constructs, such as lists and tuples, are preserved and the only extra logic applies on the encounter of variables in the pattern. When a variable is encountered, we check whether it exists in the symbol table of the analysis. If it does not, then we check whether it exists in the secondary symbol table created when examining the guard. If it does then it is replaced by the type in that symbol table. Else, it must represent an arbitrary type and gets replaced by the type `any()`. However if it exists in the analysis symbol table, it represents the type `none()`. This decision is made because the `case` will be of the form represented in Fig. 4.17. Here, `Y` is of type `integer()` but the pattern `Y` can't represent the `integer()` type because `Y` is just one integer value. Since we do not know this value, we cannot replace the pattern with it and, being overall pessimistic, we replace it with `none()`.

```
-spec f(integer(), integer()) -> boolean().
f(X, Y) ->
  case X of
    Y -> true;
    _ -> false
  end.
```

Figure 4.17: Known variable in pattern

Being able to handle a `case` node with no `values` node as an argument, we extend the algorithm to support those too. Here, we assume that the type of the argument is a tuple type with its elements being the types of the values in the `values` node. Then, in each `clause`, we deduce the type represented for each of the values with the above algorithm, given the guard and patterns, and create a tuple out of these types to be subtracted from the `case` argument type.
Another thing to note about this type annotation phase is its behavior when encountering a `call` node. The symbol table used throughout this step is initialized containing various built-in functions as described in Section 4.3.4. For each `call` node, we firstly gather the types of the arguments. If any of them has an invalid type, then the `call` node has an invalid type too. If all of them do, we fetch the signature of the function from the symbol

table. What is fetched from the symbol table will in reality be a list of signatures, where the more specific signatures precede the more general ones. The type of the `call` node is the range of the first signature that is satisfied by the argument types. If no signatures match, then the `call` node has an invalid type. This prioritization of signatures enables us to specify the node type as much as possible.
Lastly, we have to examine how we determine the types of variables introduced in patterns of `clause` nodes. For that to happen, the guard of the clause is analyzed just like in the unreachable `clause` detection and the constraints of variables are added to a symbol table. Then, we try to unify the pattern with the `clause` argument type. In the simpler case, both the argument type and the pattern will be of the same type (e.g.list) and we can recursively unify their parts. When we encounter a variable in the pattern, the type that is left is bound to that variable (given that this variable is not in the symbol table). However the types do not always match. For example, in Fig. 4.18, the argument type `T` is a union of a tuple of two numbers and a single number. The first pattern is just a tuple of two variables.

```
-spec f({number(), number()} | number()) -> number().
f(X) ->
  case X of
    {A, B} -> A + B;
    A -> A + 1
  end.
```

Figure 4.18: Case argument has the type of `{number, number} | number`. The first pattern is just a tuple. We have to use the tuple part of the union, not the whole union.

To handle this situation, the unification algorithm creates the type `T1`, which is produced by the subtraction of the type `tuple(any(), any())` from `T`. `T1` in this case is `number()` and is subtracted from `T`. The result is the tuple part of the union, with which we want to unify the pattern. If it were impossible to subtract a tuple from `T`, `T1` would be equal to `T`, the resulting type would be `none()` and we would be sure that we cannot continue with the unification.

4.3.4 Error Annotation

This part of the algorithm is responsible for deducing the `maybe_error` annotation for all the nodes in the ASTs of functions as well as the `force_constraint_logging` and `distrust_type_dependent` flags. For clarity, we will use the term "*error-free*" for a node that can't produce an error and has to be marked with the `maybe_error` annotation false. We will start with a simple function as an example shown in Fig. 4.19. We can see that the only part of the program that will produce a runtime error is the clause in line 4 where the program calls the function `error`.

```
f(X) ->
  case X of
    1 -> ok;
    _ -> error(error) % runtime error
  end.
```

Figure 4.19: Simple function that produces an error when its argument is not 1

We provide the corresponding AST of this example in Fig. 4.20. We observe the call to the `error` function in lines 16-20. In this example, we can't prune any branches that contain `case` nodes in order to reduce the search space of the program.

```
1  {c_fun,
2      [{function,{f,1}},5],
3      [{c_var,[5],0}],
4      {c_case,
5          [6],
6          {c_var,[{function,{f,1}},5],0},
7          [{c_clause,
8               [7],
9               [{c_literal,[7],1}],
10              {c_literal,[],true},
11              {c_literal,[7],ok}},
12          {c_clause,
13              [8],
14              [{c_var,[],3}],
15              {c_literal,[],true},
16              {c_call,
17                  [8],
18                  {c_literal,[8],erlang},
19                  {c_literal,[8],error},
20                  [{c_literal,[8],error}]}}]}}
```

Figure 4.20: AST of function in Fig. 4.19. The argument of the function is the variable with name `0` and the call to `error` is done in line 16.

This annotation will be made in a DFS order. Each node in the AST will decide the value of its `maybe_error` annotation depending on the `maybe_error` annotations of its children. When we need to combine two `maybe_error` annotations, the result is summed up in Table 4.2. This operation is equivalent to the conjunction operation in ternary logic, but our `true` and `false` states are swapped. To avoid the confusion we will denote this operation with the $\otimes$ symbol. However, it shares the associativity and commutativity of $\wedge$. Thus when having the `maybe_error` annotations $X_1, X_2, ..., X_n$ of the n children of a node, the annotation of this node will be $X_1 \otimes X_2 \otimes ... \otimes X_n$.

$A \otimes B$		B		
		true	**type_dependent**	**false**
A	**true**	true	true	true
	type_dependent	true	type_dependent	type_dependent
	false	true	type_dependent	false

Table 4.2: Result $A \otimes B$ of combining two `maybe_error` annotations A and B.

Naturally, in an AST of a function, calls to other functions can be found. With this emerges the need to track which function calls can produce errors. This is why the analysis introduces a symbol table in the form of a dictionary with the names of the functions as keys and their `maybe_error` annotation as values. Furthermore, the functions must be processed in a specific order if we want to have them in the symbol table when we encounter them. The order of the annotation of the functions is described in Section 4.3.2. When we examine a function we will assume that we know everything about the functions that it calls. In Fig. 4.21 function `example2:f/1` calls `example2:g/1`. `example2:g/1` can't produce an error, because regardless of the input, it will return a number. On the other hand, `example2:f/1` can produce an error, as it calls `error` depending on the value of the function's argument. Moreover `example2:g/1` should be annotated first (and will be) and added to the symbol table in order for `example2:f/1` to recognize its `maybe_error_annotation` as `false` and mark the node of this call with the annotation `false` in line 6. The last thing to note about this example is that the clause in line 6, since it can't produce errors, can be pruned, meaning that we don't need to create different inputs in order to obtain different results of `g(X)`. As a result, in this example, in contrast to the previous one, we would get a lower number of tries from CutEr in order to explore the program.
In Fig. 4.22 the need for a dynamic check (Section 4.2) is obvious. In line 6 we have a call to an error-free function that can be pruned just like in Fig. 4.21. In line 5 we have a similar call to the same error-free function. However, no branches in this call can be pruned because the result of this call is the argument of a non error-free `case` construct. The problem occurs because the interpreter in both calls will end up traversing the AST of `g/1` without any knowledge of the context of the call. We have to enforce the evaluation and logging of the call in line 5 which is done in Section 4.2.
However, if we consider the AST of `example3:f/2` provided in Fig. 4.24, we see that the argument of the `case` node in line 11 is just a variable bound to the result of `g/1` through a `let` node. Now the interpreter won't be able to know that this `let` node in line 4 will bind a variable used in a non error-free `case` argument. Thus, the static analysis must annotate this `let` node with the `force_constraint_logging` flag. These are identified by introducing a set of variables s. When a variable in an argument of a non error-free `case`

```
1  -module(example2).
2  -export([f/1]).
3
4  f(X) -> % not error free
5    case X of
6      1 -> g(X);
7      _ -> error("error") % runtime error point
8    end.
9
10 g(X) -> % error free
11   case X of
12     1 -> 1;
13     2 -> 2;
14     _ -> 1
15   end.
```

Figure 4.21: Branches in the call of `g/1` in line 6 can be pruned since paths from that point cannot produce an error.

node is used, it is added to s. When a `let` node is encountered in the analysis, firstly it analyzes the body of the `let` statement. This allows any non error-free `case` nodes to add the variables they use to s. Then, it checks if the variables that it introduces are members of s. If there is at least one such variable, then it is flagged to be logged, despite it being error-free.

Having the type annotations from Section 4.3.3 we can handle calls to functions that are error-free depending on their signature. They will have their `maybe_error` annotation set to `type_dependent`. When encountering a `call`, we check whether the called function is error-free. If it is, then the call is error-free too. If its `maybe_error` annotations is `type_dependent` and the `call` node has been marked with a valid type, the node's `maybe_error` annotation is also `type_dependent`. If it is not error-free, the call is not error-free too.

Having the type annotated functions is not enough for this step. We need to start with not just an empty symbol table, but rather with one that contains built-in functions which are error-free or `type_dependent`. For this reason, many built-in functions with their corresponding signatures have been gathered to initialize this symbol table. It is important to mention that the signatures have been altered to cover a larger range of programs as `type_dependent`. This is explained using Fig. 4.23.

Here, a function `f/1` which accepts an `integer()` and returns an `integer()`, calls a function `g/1` having the same signature. The signature that the function `erlang:+/2` needs in order to be error-free is just `(number(), number()) -> number()`. This is not enough though to mark this program as error-free because the call to `g/1` at line 3 accepts as

```
1   -module(example3).
2   -export([f/2]).
3
4   f(X, Y) -> % not error free
5     case g(X) of % has to be searched
6       1 -> g(Y); % has to be pruned
7       _ -> error("error")
8     end.
9
10  g(X) -> % error free
11    case X of
12      1 -> 1;
13      2 -> 2;
14      _ -> 1
15    end.
```

Figure 4.22: Branches in the call of `g/1` in line 6 can be pruned since it cannot produce an error. Branches in the call of `g/1` in line 5 can not since they have to be explored in order to lead to the erroneous clause in line 7.

an argument `X + 1`. Here both `X` and `1` are integers and thus subtypes of `number()`, so they respect the signature of `erlang:+/2`. Thus, this `call` will have the type of the range of `erlang:+/2` which is `number()`. This creates a complication, since `g/1` accepts an `integer()` and this call to `g/1` will be marked as non error-free. To solve this, we just add `integer() -> integer()` to the signature list of `erlang:+/2`. By adding more signatures, we can use the nature of the functions to further specify the type of their range. Another example is the function `erlang:div/2` which is not even safe, because with 0 as a second argument it produces a runtime error. If it is called with a positive integer as a second argument though, then it is safe and can be used in the analysis. So the signature `(integer(), pos_integer()) -> integer()` is put in the initial symbol table. This symbol table is the one also used by Section 4.3.3 to derive the types of `call` nodes.

```
-spec f(integer()) -> integer().
f(X) -> g(X + 1). % X + 1 should type to integer

-spec g(integer()) -> integer().
g(X) -> X + 1.
```

Figure 4.23: If the signature of `erlang:+/2` remains `(number(), number()) -> number()` then the expression `X + 1` will be annotated with the type `number()`. This will result in the type of the call to `g/1` being invalid since `number()` is not a subtype of `integer()`.

Regarding higher order functions, they introduce an issue when it comes to calling their arguments. The function arguments can have their `maybe_error` annotation as `true`, `false` or `type_dependent`. To further explain this, consider a higher order function `f`. If we try to be pessimistic and mark `f`'s arguments a priori with `true`, then we cannot classify it as error-free. This would be very restrictive. If we flag them with `false`, then we will blindly consider every call to their arguments as error-free, which not the case in many scenarios. Lastly, if we flag `f`'s arguments with `type_dependent`, we encounter the same problem as before in case another function calls `f` with a non error-free function that satisfies `f`'s signature. To tackle this, we consider the function arguments of `f` to be `type_dependent`. When we encounter a call node to `f`, we check whether the arguments of the call are error-free. If they are not, we mark this `call` node with the `distrust_type_dependent` annotation. This way, the dynamic check (Section 4.2) will start handling `type_dependent` nodes as non error-free and thus explore `f`. It is worth noticing that, instead of the `distrust_type_dependent` flag, we could add the `force_constraint_logging` flag and forcefully explore every branch, but this would explore even the strictly error-free paths and prune less branches.

As far as `letrec` nodes are concerned, their handling requires the implementation of extra logic, just like in Section 4.3.3. In the case of a list comprehension (recall Fig. 4.10 with its AST in Fig. 4.12), a recursive closure is introduced. Since it is recursive, it should be annotated just as if it was an SCC, as described in Section 4.3.2. Thus, it should ignore calls to itself until it has been introduced in the symbol table. Note that, in contrast to Section 4.3.3, we do not need any persistence in this case.

4.4 Search Space Reduction

The safe branch pruning algorithm is not meant to replace the algorithms presented in Section 3.3. The strategy, through which the next branch to be explored is selected, is independent. Regardless of that strategy though, safe branch pruning can massively reduce which branches are candidates for search especially in cases of recursion. Consider the function `f/1` in Fig. 4.25. It calls `f/2` which is a recursive function. This means that the concolic interpreter will encounter an arbitrarily long sequence of branches when traversing `f/2`, when all of them correspond to the `case` statements in lines 7,10 and 12.

Analyzing this program statically, we don't have the same constraint, and tagging this function as error free, steers the search away from all these paths. In this specific example, CutEr tries to invert 15 different branches (for a default BFS search depth of 25), where 12 of them need unsatisfiable conditions to be met. On the other hand, with safe branch pruning it will just try two satisfiable inversions and terminate, regardless of the search depth. In conclusion, improvement will be really noticeable in programs that contain recursive functions that can not produce errors, something not uncommon, especially in a functional programming language.

```
1  {c_fun,
2      [{function,{f,2}},4],
3      [{c_var,[4],0},{c_var,[4],1}],
4      {c_let,[],
5          [{c_var,[],957}],
6          {c_apply,
7              [5],
8              {c_var,[5],{g,1}},
9              [{c_var,[4],0}]},
10         {c_case,[],
11             {c_var,[],957},
12             [{c_clause,[],
13                  [{c_literal,[],1}],
14                  {c_literal,[],true},
15                  {c_apply,
16                      [6],
17                      {c_var,[6],{g,1}},
18                      [{c_var,[4],1}]}},
19              {c_clause,[],
20                  [{c_var,[],472}],
21                  {c_literal,[],true},
22                  {c_let,[],
23                      [{c_var,[],5}],
24                      {c_var,[],957},
25                      {c_call,
26                          [7],
27                          {c_literal,[7],erlang},
28                          {c_literal,[7],error},
29                          [{c_literal,[7],"error"}]}}}]}}}
```

Figure 4.24: The variable introduced in the `let` node in line 5 is the argument of the `case` node in line 11. This means that we should not prune the `call` to the function `g/1` in line 8 even if it is error-free, because we wouldn't be able to create inputs that explore all the clauses of the `case` node.

```erlang
-spec f(integer()) -> boolean().
f(X) -> % error free
  f(X, []).

-spec f(integer(), [integer()]) -> boolean().
f(X, Found) ->
  case X of
    1 -> true;
    _ ->
      case lists:member(X, Found) of
        false ->
          case X rem 2 of
            0 ->
              f(X div 2, [X|Found]);
            _ ->
              f(3 * X + 1, [X|Found])
          end;
        true ->
          false
      end
  end.
```

Figure 4.25: Program verifying the Collatz conjecture for a number.

Chapter 5

Experimental Results

The proposed method was tested on real world code, as well as custom programs created to validate its capabilities. These programs highlight the performance increase of the tool, especially in cases of recursive functions with safe branches like in Fig. 4.25. We will discuss about the strengths and limitations of this method in the following section, and then present the results from testing it in code from Erlang's standard library.

5.1 Strengths and Limitations

The main strength of this method is that it enables CutEr to verify programs that previously seemed intractable. What that means is that, even though CutEr efficiently finds bugs within its reach, depending on the depth of the bounded BFS, recursive functions can produce execution paths longer than any predetermined search depth. Additionally, this method does not add any considerable overhead to the tool, as it just performs some passes through the AST of the functions comprising the program.

The static nature of the method also has its limitations. Erlang is a dynamic programming language, which makes the static analysis unable to predict the behavior of some programs. Consider for example the program in Fig. 5.1, where taking `module:h/0` as an entry point it is difficult even to find that the callgraph contains `module:g1/1` and `module:g2/2`. Since operations such as the one performed by `module:f/1` are permitted, it is even impossible in many cases to find the callgraph.

Apart from that, even though it is common practice to provide signatures for functions, it is not mandatory and regardless of signature existence, the compilation and execution of the program will proceed normally. If the programmer does not provide signatures, then the ability of the static analysis to find safe branches will be extremely limited. The correctness of the signatures is also not checked by the compiler so in the case where they are faulty, the analysis may deduce false annotations for each branch. Fortunately, there are programs such as the `dialyzer` that help the programmer ensure the correctness of such signatures.

Finally, the determination of the type and safety of every node in the AST is undecidable

```erlang
-module(module).
-export([h/0]).

f(M, F) ->
  M:F(4).

g1(X) -> X + 1.

g2(X) -> X + 2.

h() ->
  Funs = [{module, g1}, {module, g2}],
  [f(M, F) || {M, F} <- Funs].
```

Figure 5.1: It is very difficult to calculate the callgraph of this program.

and thus, although the method can be improved, it will always consider some safe programs to be unsafe and let CutEr completely explore them.

5.2 Erlang Standard Library Code Samples

To test the method on already existing code, the module `lists` from the Erlang standard library was chosen. Each function was given as an entry point to CutEr and we gathered the ones where the method improved the efficiency of the tool. In all other cases the algorithm didn't prune any branches, so the results remain unchanged. The ones where pruning occurred are gathered in Table 5.1. Note that this improvement can't be quantified in this case, because all the improved entry points were found to be completely safe by the analysis. This means that CutEr found no errors with the analysis enabled in constant time. When, on the other hand, the analysis is disabled, CutEr will perform tests exponentially dependent on the chosen depth of the BFS.
We observe that all those functions are recursive, and that is why CutEr will have to cover a larger search space depending on the bounded BFS depth. One other positive result is that when these functions are higher order and thus inherently more difficult to be provided with valid inputs, this analysis handles them correctly. These are 16 functions out of the 52 exported functions in the lists module.

Entry point	Solved/Unsolved	
	With safe branch pruning	Without safe branch pruning
`lists:sum/1`	1/1	4/11
`lists:append/1`	2/3	72/216
`lists:map/2`	1/10	15/123
`lists:all/2`	1/10	15/72
`lists:any/2`	1/10	15/72
`lists:flatmap/2`	1/10	25/143
`lists:foldl/3`	1/10	15/123
`lists:foldr/3`	1/10	15/119
`lists:filtermap/2`	1/10	363/1195
`lists:foreach/2`	1/10	15/123
`lists:mapfoldl/3`	1/10	8/74
`lists:mapfoldr/3`	1/10	15/120
`lists:takewhile/2`	1/10	15/72
`lists:unzip/1`	1/2	8/65
`lists:unzip3/1`	1/2	8/73
`lists:last/1`	0/1	14/15

Table 5.1: Table consisting of the entry points from `lists` where the analysis improved the performance of CutEr. For each entry point, we report the number of solved and unsolved models with the analysis enabled and disabled. All the tests were done with depth for the CutEr BFS of 15 because some wouldn't terminate in reasonable time with the default depth of 25.

Chapter 6

Conclusion and Future Work

What we achieved with this thesis is to reduce the search space of concolic testing, which is dynamic in nature, using the static information of a program. Although it was developed for Erlang, this method could be implemented for other languages as well, given that the concolic testing algorithm acts on some high level representation of the program. Moreover, our method is not meant to replace all the search heuristics created for the same reason and can work alongside them for even better results.
Also, the type information provided in the code is not only helpful for this static analysis, but also crucial to cover a decent amount of programs. Without it, the only programs that we could explore more efficiently would never be actually created for real applications.
As we saw in the experimental results, in many cases, while the search space can exponentially grow with respect to the depth of the search, the safe branch pruning is able to reduce it to some constant size, especially in cases of recursive computations.
As future work, we aim to improve this method in order for it to be able to handle even more programs. Firstly, this could be realized by improving the algorithm used for detecting the unreachable clauses in a `case` construct. The use of an SMT solver should be considered in this case, since the constraints provided by the patterns and guards in the clauses are complex. Secondly, while in Erlang it is possible to provide many signatures for a function, this method does not consider functions with more than one signatures. This could be improved by running the analysis for each signature separately and then merging the resulting annotated ASTs produced. Lastly, right now our method does not support bitstrings, which is something that could be implemented in the future.

Bibliography

[1] The Editors of Encyclopaedia Britannica. “software (computing)”. In: *Encyclopedia Britannica*. URL: https://www.britannica.com/technology/software.

[2] Mohd Khan. “Different Forms of Software Testing Techniques for Finding Errors”. In: *International Journal of Computer Science Issues* 7 (May 2010).

[3] Mohd Ehmer and Farmeena Khan. “A Comparative Study of White Box, Black Box and Grey Box Testing Techniques”. In: *International Journal of Advanced Computer Science and Applications* 3 (June 2012). DOI: 10.14569/IJACSA.2012.030603.

[4] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. “A Formal System for Testing and Debugging Programs by Symbolic Execution”. In: *Proceedings of the International Conference on Reliable Software*. Los Angeles, California: ACM, 1975. ISBN: 9781450373852. DOI: 10.1145/800027.808445. URL: https://doi.org/10.1145/800027.808445.

[5] Koushik Sen, Darko Marinov, and Gul 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. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1595930140. DOI: 10.1145/1081706.1081750. URL: https://doi.org/10.1145/1081706.1081750.

[6] Koushik Sen and Gul Agha. “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools”. In: *18th International Conference on Computer Aided Verification (CAV’06)*. Vol. 4144. LNCS. Springer, 2006, pp. 419–423. DOI: 10.1007/11817963_38. URL: http://dx.doi.org/10.1007/11817963_38.

[7] Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. “Concolic Testing of Functional Languages”. In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: ACM, 2015, pp. 137–148. DOI: 10.1145/2790449.2790519. URL: http://doi.acm.org/10.1145/2790449.2790519.

[8] Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. “Concolic Testing of Functional Languages”. In: *Sci. Comput. Program.* 147 (2017), pp. 109–134. DOI: 10.1016/j.scico.2017.04.008. URL: https://doi.org/10.1016/j.scico.2017.04.008.

[9] Joe Armstrong. “A History of Erlang”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pp. 6-1-6–26. ISBN: 9781595937667. DOI: 10.1145/1238844.1238850. URL: https://doi.org/10.1145/1238844.1238850.

[10] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. “Core Erlang 1.0 language specification”. In: *Technical Report 2000-030*. Department of Information Technology, Uppsala University, Sweden, 2000.

[11] Richard Carlsson. “An introduction to Core Erlang”. In: *Proceedings of the PLI’ 01 Erlang Workshop*. 2001.

[12] Tobias Lindahl and Konstantinos Sagonas. “Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story”. In: *Programming Languages and Systems: Proceedings of the Second Asian Symposium*. Ed. by Chin Wei-Ngan. Vol. 3302. LNCS. Berlin, Germany: Springer, 2004, pp. 91–106. DOI: 10.1007/978-3-540-30477-7_7. URL: https://doi.org/10.1007/978-3-540-30477-7%5C_7.

[13] Tobias Lindahl and Konstantinos Sagonas. “Practical Type Inference Based on Success Typings”. In: *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. Venice, Italy: ACM Press, 2006, pp. 167–178. DOI: 10.1145/1140335.1140356. URL: http://doi.acm.org/10.1145/1140335.1140356.

[14] Tobias Lindahl and Konstantinos Sagonas. “TypEr: A Type Annotator of Erlang Code”. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*. Tallinn, Estonia: ACM, 2005, pp. 17–25. DOI: 10.1145/1088361.1088366. URL: http://doi.acm.org/10.1145/1088361.1088366.

[15] Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. “Message analysis for concurrent programs using message passing”. In: *ACM Trans. Program. Lang. Syst.* 28.4 (2006), pp. 715–746. DOI: 10.1145/1146813. URL: http://doi.acm.org/10.1145/1146813.

[16] Maria Christakis and Konstantinos Sagonas. “Static Detection of Race Conditions in Erlang”. In: *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Proceedings*. Ed. by Manuel Carro and Ricardo Peña. Vol. 5937. LNCS. Madrid, Spain: Springer, Jan. 2010, pp. 119–133. DOI: 10.1007/978-3-642-11503-5_11. URL: https://doi.org/10.1007/978-3-642-11503-5%5C_11.

[17] Maria Christakis and Konstantinos Sagonas. “Detection of Asynchronous Message Passing Errors Using Static Analysis”. In: *Practical Aspects of Declarative Languages, 13th International Symposium, PADL 2011, Proceedings*. Ed. by Ricardo Rocha and John Launchbury. Vol. 6539. LNCS. Austin, TX, USA: Springer, Jan. 2011, pp. 5–18.

DOI: 10.1007/978-3-642-18378-2_3. URL: https://doi.org/10.1007/978-3-642-18378-2%5C_3.

[18] Tamas Nagy and Aniko Nagyne Vig. "Erlang Testing and Tools Survey". In: *Proceedings of the 2008 ACM SIGPLAN Workshop on Erlang*. Victoria, Canada: ACM, Sept. 2008, pp. 21–28. DOI: 10.1145/1411273.1411277. URL: https://doi.org/10.1145/1411273.1411277.

[19] Manolis Papadakis and Konstantinos Sagonas. "A PropEr Integration of Types and Function Specifications with Property-based Testing". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. Tokyo, Japan: ACM, 2011, pp. 39–50. DOI: 10.1145/2034654.2034663. URL: http://doi.acm.org/10.1145/2034654.2034663.

[20] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. "Test-Driven Development of Concurrent Programs using Concuerror". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. New York, NY, USA: ACM, Sept. 2011, pp. 51–61. DOI: 10.1145/2034654.2034664. URL: https://doi.org/10.1145/2034654.2034664.

[21] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. "Systematic Testing for Detecting Concurrency Errors in Erlang Programs". In: *Sixth IEEE International Conference on Software Testing, Verification and Validation*. ICST 2013. Luxembourg: IEEE, 2013, pp. 154–163. DOI: 10.1109/ICST.2013.50. URL: https://doi.org/10.1109/ICST.2013.50.

[22] Huiqing Li and Simon Thompson. "Tool Support for Refactoring Functional Programs". In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. New York, NY, USA: ACM, Jan. 2008, pp. 199–203. DOI: 10.1145/1328408.1328437. URL: https://doi.org/10.1145/1328408.1328437.

[23] Huiqing Li and Simon Thompson. "Clone Detection and Removal for Erlang/OTP within a Refactoring Environment". In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. New York, NY, USA: ACM, Jan. 2009, pp. 169–177. DOI: 10.1145/1480945.1480971. URL: https://doi.org/10.1145/1480945.1480971.

[24] Thanassis Avgerinos and Konstantinos Sagonas. "Cleaning up Erlang Code is a Dirty Job but Somebody's Gotta Do It". In: *Proceedings of the 8th ACM SIGPLAN Erlang Workshop*. New York, NY, USA: ACM, Sept. 2009, pp. 1–10. DOI: 10.1145/1596600.1596602. URL: https://doi.org/10.1145/1596600.1596602.

[25] Konstantinos Sagonas and Thanassis Avgerinos. "Automatic Refactoring of Erlang Programs". In: *Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: ACM,

Sept. 2009, pp. 13–24. DOI: 10.1145/1599410.1599414. URL: https://doi.org/10.1145/1599410.1599414.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[27] Nicky Williams, Bruno Marre, and Patricia Mouy. "On-the-fly generation of k-path tests for C functions". In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering.* ASE '04. USA: IEEE Computer Society, 2004, pp. 290–297. DOI: 10.1109/ASE.2004.1342749.

[28] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. "PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis". In: *Dependable Computing - EDCC 5.* Ed. by Mario Dal Cin, Mohamed Kaâniche, and András Pataricza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 281–292. ISBN: 978-3-540-32019-7.

[29] Mark Wallace, Stefano Novello, and Joachim Schimpf. "ECLiPSe: a Platform for Constraint Logic Programming". In: *ICL Systems Journal* 12 (Oct. 1997).

[30] Leonardo de Moura and Nikolaj Björner. "Z3: an efficient SMT solver". In: *Proceedings of the Theory and Practice of Software, 14th International Tools and Algorithms for the Construction and Analysis of Systems.* Vol. 4963. LNCS. Berlin, Heidelberg: Springer-Verlag, Apr. 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[31] Jacob Burnim and Koushik Sen. "Heuristics for Scalable Dynamic Test Generation". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[32] Sooyoung Cha, Seongjoon Hong, Jiseong Bak, Jingyoung Kim, Junhee Lee, and Hakjoo Oh. "Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics". In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3101870.

[33] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. "CarFast: Achieving Higher Statement Coverage Faster". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* FSE '12. Cary, North Carolina: ACM, 2012. ISBN: 9781450316149. DOI: 10.1145/2393596.2393636. URL: https://doi.org/10.1145/2393596.2393636.

[34] Hyunmin Seo and Sunghun Kim. “How We Get There: A Context-Guided Search Strategy in Concolic Testing”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 413–424. ISBN: 9781450330565. DOI: 10.1145/2635868.2635872. URL: https://doi.org/10.1145/2635868.2635872.

[35] Patrice Godefroid, Michael Y. Levin, and David A Molnar. “Automated Whitebox Fuzz Testing”. In: *Network Distributed Security Symposium (NDSS)*. Internet Society. 2008. URL: http://www.truststc.org/pubs/499.html.

[36] Ericsson AB. *OTP Design Principles*. 2021. URL: https://www.erlang.org/doc/design_principles/des_princ.html.

[37] Mihalis Pitidis and Konstantinos Sagonas. “Purity in Erlang.” In: vol. 6647. Sept. 2010, pp. 137–152. DOI: 10.1007/978-3-642-24276-2_9.

[38] Aggelos Giantsios. “Program testing by combining symbolic and concrete execution with automatic generation of inputs”. MA thesis. National Technical University of Athens, School of Electrical and Computer Engineering, Greece, 2014. URL: http://artemis-new.cslab.ece.ntua.gr:8080/jspui/handle/123456789/16798.

[39] Niloofar Razavi, Franjo Ivančić, Vineet Kahlon, and Aarti Gupta. “Concurrent test generation using concolic multi-trace analysis”. In: *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Proceedings*. LNCS. 2012, pp. 239–255. ISBN: 9783642351815. DOI: 10.1007/978-3-642-35182-2_17.

[40] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft.” In: *Queue* 10.1 (Jan. 2012), pp. 20–27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. URL: https://doi.org/10.1145/2090147.2094081.

[41] Cristian Cadar, Daniel Dunbar, and Dawson 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. San Diego, California: USENIX Association, 2008, pp. 209–224.

[42] Miguel Jimenez, Tobias Lindahl, and Konstantinos Sagonas. “A Language for Specifying Type Contracts in Erlang and Its Interaction with Success Typings”. In: *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*. ERLANG ’07. Freiburg, Germany: ACM, 2007. ISBN: 9781595936752. DOI: 10.1145/1292520.1292523. URL: https://doi.org/10.1145/1292520.1292523.