ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Δυναμικός συμβολικός έλεγχος με χρήση πολλαπλών SMT επιλυτών

Διπλωματική εργασία

Κωνσταντίνος Ρακτιβάν

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

Αθήνα, Νοέμβριος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Δυναμικός συμβολικός έλεγχος με χρήση πολλαπλών SMT επιλυτών

Διπλωματική εργασία

Κωνσταντίνος Ρακτιβάν

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Νοεμβρίου 2017.

---	---	---
Κωνσταντίνος Σαγώνας	Νικόλαος Παπασπύρου	Κωνσταντίνος Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2017

Κωνσταντίνος Ρακτιβάν

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

Περίληψη

Ο concolic έλεγχος είναι μία πολλά υποσχόμενη τεχνική για τον έλεγχο προγραμμάτων. Λαμβάνει υπόψη τόσο συμβολικές (symbolic) όσο και σταθερές (concrete) τιμές των μεταβλητών, μεγιστοποιώντας το πλήθος των μονοπατιών εκτέλεσης (execution paths) που καταφέρνει εξετάσει. Χρησιμοποιούμε έναν SMT επιλυτή για να υπολογίσουμε τιμές των παραμέτρων εισόδου που οδηγούν την εκτέλεση του ελεγχόμενου κώδικα σε επιλεγμένο μονοπάτι. Αντικείμενο της παρούσας εργασίας είναι η δημιουργία ενός πλαισίου που επιτρέπει την αξιοποίηση πολλαπλών επιλυτών για τον έλεγχο ενός προγράμματος, επικοινωνώντας με αυτούς μέσω της γλώσσας SMTLIB. Επίσης, αναφέρουμε τα προβλήματα που προκύπτουν κατά την προσθήκη των επιλυτών, τον τρόπο αντιμετώπισής τους και τελικά αναδεικνύουμε τα πλεονεκτήματα της συγκεκριμένης προσέγγισης. Η εργασία αναπτύχθηκε στο εργαλείο CutEr, το οποίο εφαρμόζει concolic testing σε προγράμματα γραμμένα σε γλώσσα Erlang. Η υλοποίησή της παρέχει τη δυνατότητα στο εργαλείο να ελέγχει επιτυχημένα ένα ευρύτερο σύνολο προγραμμάτων, το οποίο δε μπορούσε να διαχειριστεί προηγουμένως.

Λέξεις κλειδιά

έλεγχος, δυναμικός συμβολικός έλεγχος, SMT επιλυτές, SMTLIB, CutEr

Abstract

Concolic testing is a most promising program testing technique. It takes into consideration both symbolic and concrete variable values, maximizing the path coverage. We use an SMT solver to calculate a suitable assignment of input values that direct the code execution to a select path. The objective of the present thesis is to design and implement a framework that allows the exploitation of multiple solvers, in order to test a program by communicating with them via the SMTLIB language. Moreover, while incorporating various solvers several problems occur that have to be confronted and, finally, the advantages of this specific approach are highlighted. The project was developed on CutEr, a tool which applies concolic testing to Erlang programs. From now on, the tool can successfully test a wider set of programs that could not be handled previously.

Keywords

testing, concolic testing, SMT solvers, SMTLIB, CutEr

Πίνακας περιεχομένων

Πίνακας εικόνων

1 Εισαγωγή

Αρχικά παρουσιάζουμε με δυο λόγια το αντικείμενο της εργασίας.

1.1 Concolic testing

Κατά την εκτέλεση του κώδικα ενός προγράμματος, ανεξαρτήτως γλώσσας προγραμματισμού, συχνά προκύπτουν σφάλματα εκτέλεσης (runtime errors). Η εύρεση και επιδιόρθωσή τους είναι αναγκαία για την ομαλή λειτουργία του προγράμματος για κάθε τιμή των παραμέτρων εισόδου. Ένας κλάδος της σχεδίασης λογισμικού ασχολείται με αυτό ακριβώς το πρόβλημα, του ελέγχου των προγραμμάτων. Έχουν προταθεί και υλοποιηθεί πολλές μέθοδοι, οι οποίες διαφέρουν σημαντικά ως προς τον τρόπο με τον οποίο εξετάζουν ένα πρόγραμμα. Στην παρούσα χρονική στιγμή, η πιο αποτελεσματική τεχνική φαίνεται πως είναι ο δυναμικός συμβολικός έλεγχος (dynamic symbolic execution), γνωστός και ως concolic testing (όρος παραγόμενος από τη μείξη των αγγλικών λέξεων concrete και symbolic). Η διαφορά του από το συμβολικό έλεγχο (symbolic execution) είναι ότι κατά την συμβολική ανάλυση του κώδικα κρατά αποθηκευμένες και τις τιμές που έλαβαν οι παράμετροι εισόδου στην προηγούμενη εκτέλεση. Κατά την εφαρμογή του concolic testing λαμβάνεται υπόψη το συντακτικό δέντρο του ελεγχόμενου κώδικα και διασχίζονται συστηματικά όλα τα μονοπάτια εκτέλεσής του, μέχρι ένα συγκεκριμένο βάθος αναζήτησης, οπότε αποκαλύπτονται τυχόν σφάλματα απολύτως αυτοματοποιημένα.

Για να οδηγηθεί η εκτέλεση του ελεγχόμενου κώδικα σε επιλεγμένο μονοπάτι, είναι απαραίτητη η εύρεση κατάλληλης αποτίμησης των παραμέτρων εισόδου. Επομένως, το εργαλείο ελέγχου οφείλει να συλλέξει όλους τους περιορισμούς (constraints) που προκύπτουν από τις εντολές διακλάδωσης (branches) του μονοπατιού εκτέλεσης και εμπλέκουν τις μεταβλητές εισόδου. Στη συνέχεια, στέλνει το σύνολο περιορισμών σε ένα μαθηματικό πρόγραμμα, το οποίο απαντά κατά πόσο είναι επιλύσιμο ή όχι και, σε περίπτωση που όντως είναι επιλύσιμο, επιστρέφει και ένα μοντέλο που να ικανοποιεί το σύνολο περιορισμών. Τα μαθηματικά προγράμματα που αντιμετωπίζουν τα σχηματιζόμενα σύνολα περιορισμών ονομάζονται SMT επιλυτές, από την κατηγορία των προβλημάτων που μπορούν να επιλύσουν. Κυκλοφορούν πολλοί SMT επιλυτές, καθένας από τους οποίους εξειδικεύεται σε ορισμένα υποσύνολα της λογικής των προβλημάτων SMT (Satisfiability Modulo Theories) [1].

1.2 Πολλαπλοί επιλυτές

Μία έξυπνη ιδέα είναι η εκμετάλλευση περισσοτέρων του ενός SMT επιλυτών με σκοπό τον επιτυχημένο έλεγχο μεγαλύτερης ποικιλίας προγραμμάτων από το ίδιο εργαλείο ελέγχου. Καθώς κάθε επιλυτής παρουσιάζει πλεονεκτήματα σε μέρος των προβλημάτων που μας απασχολούν, η συμπερίληψη πολλών επιλυτών στο πρόγραμμα ελέγχου καθιστά εφικτή την αντιμετώπιση ευρύτερου συνόλου προγραμμάτων. Αντικείμενο της παρούσας εργασίας είναι η τροποποίηση ενός εργαλείου ελέγχου ώστε να ενσωματώνει πολλαπλούς επιλυτές, οι οποίοι καλούνται είτε με τη σειρά είτε ταυτόχρονα, με βάση κάποιο σκεπτικό. Κατά το στάδιο εύρεσης μιας αποτίμησης των παραμέτρων εισόδου

λαμβάνεται υπόψη η απάντηση του πρώτου επιλυτή που αποκρίνεται επιτυχημένα, τακτική που εξυπηρετεί στην αποτελεσματική και ταχεία επίλυση διαφορετικών συνόλων περιορισμών.

Η επικοινωνία με τους SMT επιλυτές γενικά πραγματοποιείται μέσω εξειδικευμένων διεπαφών. Προσφάτως όμως σχεδιάστηκε η γλώσσα SMTLIB (SMT library) [2], η οποία επιτρέπει την κωδικοποίηση των SMT προβλημάτων και των απαντήσεών τους σε συμβολοσειρές, ανεξάρτητα από τον επιλυτή και το πρόγραμμα ελέγχου. Από τα προβλήματα αυτά δεν εξαιρούνται ούτε εκείνα που εμπλέκουν αναδρομικές συνθήκες. Πλέον μεγάλη μερίδα των επιλυτών υποστηρίζει τη γλώσσα αυτή, γεγονός που διευκολύνει σημαντικά την προσέγγισή μας, αφού η εμπλοκή αρκετών επιλυτών μπορεί να γίνει με παραπλήσια σύνολα εντολών.

1.3 Πορεία εργασίας

Στην πράξη, το προηγούμενο σκεπτικό υλοποιείται στο CutEr [3], ένα εργαλείο δυναμικού συμβολικού ελέγχου προγραμμάτων γραμμένων σε γλώσσα Erlang. Το εργαλείο αυτό αρχικά χρησιμοποιούσε μόνο τον επιλυτή Z3 μέσω του Python API του.

Σε πρώτη φάση το τροποποιούμε προσδίδοντάς του την ικανότητα επικοινωνίας με τον ίδιο επιλυτή, αλλά κωδικοποιώντας τους τύπους δεδομένων και τους περιορισμούς από τη γλώσσα Erlang του ελεγχόμενου προγράμματος σε γλώσσα SMTLIB. Εύκολα αντικαθιστούμε τον επιλυτή Z3 με άλλους, για παράδειγμα τον επιλυτή CVC4 [4], προσαρμόζοντας τις μεθόδους ώστε να ανταποκρίνεται με τον καλύτερο δυνατό τρόπο σε κάθε ελεγχόμενο πρόγραμμα ο εκάστοτε επιλυτής. Διαπιστώνουμε και πειραματικά ότι υπάρχουν προγράμματα που αντιμετωπίζονται από μεμονωμένους επιλυτές.

Έπειτα συνδέουμε πολλούς επιλυτές και υλοποιούμε το απαραίτητο πλαίσιο για να επιτρέπουμε στο πρόγραμμα να διαλέγει, σύμφωνα με τη στρατηγική ενός επιλεγμένου συντονιστή, αν θα χρησιμοποιήσει έναν ή περισσότερους επιλυτές και με ποιον τρόπο, δηλαδή αν θα ακολουθήσει ορισμένη σειρά ή θα τους καλέσει ταυτόχρονα, περιμένοντας αποτέλεσμα. Όπως είναι αναμενόμενο, παρατηρούμε πως πράγματι η αξιοποίηση πολλαπλών επιλυτών επιτρέπει την κάλυψη μεγαλύτερου συνόλου προγραμμάτων, χάρη στη χρήση προηγμένων εντολών της γλώσσας SMTLIB και την εκμετάλλευση των προτερημάτων που εμφανίζει κάθε επιλυτής σε διακεκριμένες κατηγορίες συνόλων περιορισμών.

1.4 Περιεχόμενα κεφαλαίων

Προς διευκόλυνση του αναγνώστη, καταγράφουμε τι αντικείμενο διαπραγματεύεται κάθε κεφάλαιο της εργασίας.

- Στο κεφάλαιο 1 παρουσιάζουμε εισαγωγικά το αντικείμενο της εργασίας.
- Στο κεφάλαιο 2 εξετάζουμε περιληπτικά ορισμένες θεμελιώδεις έννοιες.
- Στο κεφάλαιο 3 αναφερόμαστε σε υπάρχοντα εργαλεία σχετικά με το θέμα, ενώ δίνουμε έμφαση στην εφαρμογή CutEr, επί της οποίας εκπονήθηκε η εργασία.

- Στο κεφάλαιο 4 εξηγούμε με ποιον τρόπο καταστήσαμε εφικτή την επικοινωνία του CutEr με έναν SMT επιλυτή, κάνοντας χρήση της γλώσσας SMTLIB.
- Στο κεφάλαιο 5 αναλύουμε τη μέθοδο ενσωμάτωσης και επιλογής πολλών επιλυτών κατά τη διαδικασία ελέγχου.
- Στο κεφάλαιο 6 και το κεφάλαιο 7 καταγράφουμε συνοπτικά ποια νέα χαρακτηριστικά προσδώσαμε στο CutEr και ένα σύνολο βελτιώσεων που μπορούν να υλοποιηθούν.
- Στο κεφάλαιο 8, τέλος, παραθέτουμε λεπτομερώς ορισμένα τμήματα κώδικα, μαζί με ερμηνευτικά σχόλια και μαθηματικές αποδείξεις.

2 Υπόβαθρο

Αναφερόμαστε επιγραμματικά στον έλεγχο προγραμμάτων και στα μαθηματικά προβλήματα που προκύπτουν, έννοιες που κατέχουν εξέχουσα θέση στο αντικείμενο της παρούσας εργασίας.

2.1 Έλεγχος προγραμμάτων

Κατά την εκτέλεση του κώδικα ενός προγράμματος, ανεξαρτήτως γλώσσας προγραμματισμού, συχνά ανακύπτουν σφάλματα εκτέλεσης, η ύπαρξη των οποίων συνήθως οδηγεί το πρόγραμμα σε λανθασμένη έξοδο ή σε ανεπιτυχή πρόωρο τερματισμό. Έχουν αναπτυχθεί διάφορες μέθοδοι εξεύρεσης των σφαλμάτων αυτών, τις οποίες μπορούμε να αξιολογήσουμε ανάλογα με το πόσο αυτοματοποιημένες είναι και με το βαθμό κάλυψης του εξεταζόμενου κώδικα (code coverage). Ο βαθμός κάλυψης του ελέγχου ενός κώδικα υπολογίζεται με διάφορες μετρικές, μεταξύ των οποίων περιλαμβάνονται τα παρακάτω κριτήρια:

- line coverage: Εκτελέστηκαν όλες οι γραμμές του κώδικα κατά τον έλεγχο;
- edge coverage: Ακολουθήθηκαν και τα δύο μονοπάτια σε όλες τις εντολές διακλάδωσης; Κάθε εντολή διακλάδωσης (branch statement) οδηγεί σε δύο σώματα εντολών, ανάλογα με το αν η συνθήκη αποτιμάται ως σωστή (true branch) ή λάθος (false branch).
- path coverage: Ακολουθήθηκαν όλα τα δυνατά μονοπάτια εκτέλεσης ενός τμήματος κώδικα; Επειδή υπάρχει περίπτωση τα προσιτά μονοπάτια να είναι εκθετικά πολλά ή και άπειρα, μπορούμε να περιορίσουμε την ερώτηση με μία παράμετρο βάθους.

2.1.1 Έλεγχος μονάδας

Ένας απλός τρόπος να ελέγξουμε αν ένα πρόγραμμα λειτουργεί σωστά είναι να δημιουργήσουμε ένα σύνολο αποδεκτών εισόδων του προγράμματος και να αποτιμήσουμε, χωρίς τη βοήθεια του εξεταζόμενου προγράμματος, τις ορθές εξόδους που αντιστοιχούν σε κάθε είσοδο. Έπειτα εκτελούμε το πρόγραμμα για τις δεδομένες εισόδους και συγκρίνουμε τα αποτελέσματα που λαμβάνουμε με τις προϋπολογισμένες εξόδους. Η διαδικασία αυτή ονομάζεται έλεγχος μονάδας (unit testing) και εντοπίζει λογικά σφάλματα (logic errors) και σφάλματα εκτέλεσης (runtime errors). Ως μέθοδος είναι απολύτως χειροκίνητη και σε καμία περίπτωση δεν εγγυάται την εξεύρεση όλων των σφαλμάτων ούτε την κάλυψη του κώδικα. Ασφαλώς, η πιθανότητα εντοπισμού ενός σφάλματος αυξάνεται με το πλήθος των δοκιμαζόμενων εισόδων.

2.1.2 Τυχαίος έλεγχος

Ο δυναμικός έλεγχος (dynamic testing) του κώδικα περιλαμβάνει την εκτέλεσή του για πολλά παραδείγματα εισόδων, κατά τρόπο ώστε να ακολουθηθεί ει δυνατόν κάθε εφικτό μονοπάτι εκτέλεσης. Ο έλεγχος βασίζεται σε τεχνικές παραγωγής τυχαίων τιμών για τις παραμέτρους εισόδων με χρήση κατάλληλων γεννητριών, είναι δηλαδή εν μέρει αυτοματοποιημένη μέθοδος (semi-random). Οι τεχνικές αυτές συνήθως εξετάζουν προδιαγραφές για την έξοδο ανάλογα με την είσοδο και τη λογική που υλοποιεί το

πρόγραμμα (property-based testing). Ομοίως με τον έλεγχο μονάδας, αποκαλύπτει λογικά σφάλματα και σφάλματα εκτέλεσης και δεν εγγυάται την κάλυψη του κώδικα, επομένως δεν είναι βέβαιος ο εντοπισμός όλων των σφαλμάτων.

2.1.3 Συμβολική εκτέλεση

Όταν εφαρμόζουμε στατική ανάλυση ενός κώδικα, ακολουθούμε όλα τα μονοπάτια εκτέλεσης (execution paths), συλλέγοντας τις συνθήκες που εμπλέκουν τις μεταβλητές εισόδου σε κάθε εντολή διακλάδωσης (branch). Για να διαπιστώσουμε αν ένα μονοπάτι εκτέλεσης οδηγεί σε σφάλμα, υπολογίζουμε μία ανάθεση τιμών στις μεταβλητές εισόδου που να οδηγεί την εκτέλεση του κώδικα στο επιλεγμένο μονοπάτι και έπειτα εκτελούμε τον κώδικα με την συγκεκριμένη είσοδο. Η μετάβαση από ένα μονοπάτι εκτέλεσης σε κάποιο άλλο γίνεται με επιλεκτική αναστροφή μιας συνθήκης, κατά τρόπο ώστε τελικά να διασχιστεί όλος ο ελεγχόμενος κώδικας. Η διαδικασία αυτή ονομάζεται συμβολική εκτέλεση (symbolic execution), καθώς κατά την ανάλυση του κώδικα χρησιμοποιούμε μόνο συμβολικές (symbolic) και όχι σταθερές (concrete) τιμές για τις παραμέτρους.

Η συμβολική ανάλυση του κώδικα είναι απολύτως αυτόματη και θα αρκούσε εάν ήταν δυνατόν να υπολογίσουμε για κάθε δυνατό μονοπάτι εκτέλεσης μία είσοδο που να οδηγεί σε αυτό. Ωστόσο υπάρχουν περιπτώσεις που είτε τα μονοπάτια εκτέλεσης είναι εκθετικά πολλά ή και άπειρα, είτε είναι εξαιρετικά δύσκολο ή εντελώς αδύνατο να επιλυθεί μαθηματικά το σύνολο περιορισμών που έχει σχηματιστεί για τις μεταβλητές εισόδου με βάση τις αποφάσεις που λάβαμε σε κάθε διακλάδωση. Οι περιορισμοί μπορεί να δυσεπίλυτοι είτε λόγω πολυπλοκότητας των συνθηκών είτε εξαιτίας της εξάρτησής τους από μη προβλέψιμους παράγοντες (side effects).

2.1.4 Συνδυαστική προσέγγιση

Επιχειρούμε με κάποιο συστηματικό τρόπο να απλοποιήσουμε το πρόβλημα που δημιουργείται κατά το στατικό έλεγχο, συνδυάζοντάς τον με δυναμικό έλεγχο. Εκτελούμε τον κώδικα για μία τυχαία αρχική είσοδο και ακολουθούμε το αντίστοιχο μονοπάτι εκτέλεσης, συλλέγοντας περιορισμούς για τις μεταβλητές εισόδου από τις εντολές διακλάδωσης. Για να καλύψουμε όσα περισσότερα μονοπάτια εκτέλεσης μπορούμε, αναστρέφουμε με καθορισμένη σειρά τις αποφάσεις που πήραμε στις διακλαδώσεις, δημιουργώντας ένα νέο σύνολο περιορισμών. Η επίλυση του νέου συνόλου μάς οδηγεί σε ένα καινούριο μονοπάτι εκτέλεσης. Σε περίπτωση που το σύνολο περιορισμών είναι εξαιρετικά πολύπλοκο για να επιλυθεί από το μαθηματικό πρόγραμμα, αναθέτουμε σε μία ή περισσότερες από τις εμπλεκόμενες μεταβλητές την τιμή που είχε κατά την αμέσως προηγούμενη εκτέλεση του ελεγχόμενου κώδικα. Κατ' αυτόν τον τρόπο προσπερνούμε το πρόβλημα της μη επιλυσιμότητας των συνόλων περιορισμών, με τίμημα το ενδεχόμενο παράλειψης ορισμένων μονοπατιών εκτέλεσης. Η ανάμειξη συμβολικής εκτέλεσης με χρήση σταθερών τιμών για τις μεταβλητές εισόδου ονομάζεται concolic testing (μείξη των όρων concrete και symbolic) ή dynamic symbolic execution.

2.2 Πρόβλημα απόφασης

Όπως είδαμε παραπάνω, κατά τη συμβολική εκτέλεση ενός προγράμματος ακολουθούμε κάθε φορά ένα μονοπάτι. Σε κάθε εντολή διακλάδωσης συναντούμε συνθήκες που αφορούν άμεσα ή έμμεσα τις μεταβλητές εισόδου, υπό την έννοια ότι οι συνθήκες αποτελούν λογικές συναρτήσεις ενδεχομένως μη λογικών μεταβλητών, είναι δηλαδή κατηγορήματα. Η λήψη μιας απόφασης στην εντολή διακλάδωσης, προκειμένου να παραμείνουμε στο μονοπάτι, ισοδυναμεί με αποτίμηση του αντίστοιχου κατηγορήματος ως αληθούς ή ψευδούς. Η ένωση αυτών συνθηκών συνθέτει το σύνολο περιορισμών του μονοπατιού εκτέλεσης και είναι ένα πρόβλημα SMT (satisfiability modulo theories) [1].

Σε περίπτωση που το πρόβλημα SMT είναι ικανοποιήσιμο, δηλαδή υπάρχει ανάθεση τιμών στις μεταβλητές που να ικανοποιούν το σύνολο περιορισμών, τότε το επιλεγμένο μονοπάτι εκτέλεσης είναι προσπελάσιμο. Επομένως, μπορούμε να εκτελέσουμε τον ελεγχόμενο κώδικα με είσοδο τη συγκεκριμένη αποτίμηση και να οδηγηθούμε στο μονοπάτι εκτέλεσης, αποκαλύπτοντας τυχόν σφάλματα στη διαδρομή. Αν όμως το πρόβλημα δεν είναι ικανοποιήσιμο, είμαστε βέβαιοι ότι δεν υπάρχει καμία είσοδος που να οδηγεί την εκτέλεση του κώδικα στο επιλεγμένο μονοπάτι, οπότε οποιοδήποτε σφάλμα στο τέλος του μονοπατιού δεν θα αποκαλυφθεί ποτέ.

Σημειώνουμε πως υπάρχουν μη αποφασίσιμα προβλήματα, δηλαδή προβλήματα για τα οποία, λόγω της εμπλεκόμενης θεωρίας, είναι αδύνατο να αποφανθούμε ως προς την επιλυσιμότητά τους, αν δεν γνωρίζουμε ήδη μία λύση. Ως παράδειγμα αναφέρουμε τη θεωρία της μη γραμμικής αριθμητικής ακεραίων, στην οποία ανήκουν κατηγορήματα όπως το $x \cdot y = c$, όπου x και y ακέραιες μεταβλητές και c ακέραια σταθερά. Παρά ταύτα, αν κατά κάποιο τρόπο βρεθεί λύση στο πρόβλημα, ίσως με χρήση ευριστικής μεθόδου, ασφαλώς το πρόβλημα είναι ικανοποιήσιμο και μπορούμε να αξιοποιήσουμε τη λύση για να εκτελέσουμε τον κώδικα.

2.2.1 Επιλυτές περιορισμών

Προσφάτως έχουν αναπτυχθεί μαθηματικά προγράμματα, σχεδιασμένα ώστε να επιλύουν προβλήματα SMT, αρκετά εκ των οποίων είναι διαθέσιμα στο κοινό, ορισμένα δε και ως ανοικτό λογισμικό. Καθώς τα προβλήματα εμπίπτουν σε πολλές θεωρίες των μαθηματικών (όπως γραμμικής ή μη γραμμικής αριθμητικής ακεραίων ή πραγματικών αριθμών, θεωρίες με λίστες, με διανύσματα, με ακολουθίες δυφίων (bitvectors) σταθερού μήκους, με ποσοδείκτες ή άγνωστα σύμβολα), καθένας από τους επιλυτές εξειδικεύεται σε ένα υποσύνολο των προβλημάτων.

Οι επιλυτές συνήθως συνοδεύονται από πλήθος ρυθμίσεων που σχετίζονται με τον τρόπο επικοινωνίας τους με κάποιο εξωτερικό πρόγραμμα, με τη χρήση των πόρων του συστήματος και με τη βελτιστοποίηση της απόδοσής τους σε συγκεκριμένη μερίδα προβλημάτων SMT.

2.2.2 Γλώσσα SMTLIB

Αρχικά κάθε SMT επιλυτής παρείχε εξειδικευμένες διεπαφές (application programming interface, API) για επικοινωνία με εξωτερικά προγράμματα γραμμένα σε διάφορες

γλώσσες προγραμματισμού. Για παράδειγμα έχουμε, μεταξύ άλλων, το C++ API του CVC4 [4], το C++ API του Z3 [5] και το Python API του Z3. Κάθε διεπαφή χρησιμοποιούσε διαφορετικά σύμβολα για να κωδικοποιήσει στο εξωτερικό πρόγραμμα το ίδιο ακριβώς πρόβλημα SMT κατά τρόπο που να αντιλαμβάνεται ο εκάστοτε επιλυτής. Ομοίως, όριζε τις μεθόδους με τις οποίες θα αποκωδικοποιούσε το εξωτερικό πρόγραμμα την αντίστοιχη λύση από την απόκριση του επιλυτή.

Ξεκίνησε, λοιπόν, μία πρωτοβουλία κωδικοποίησης όλων των προβλημάτων SMT σε μία γλώσσα, την SMTLIB (SMT library). Η γλώσσα αυτή έχει καθορισμένο λεξιλόγιο και συντακτικό, τα οποία προσδίδουν την ικανότητα περιγραφής των προβλημάτων SMT και των λύσεών τους. Πλέον αρκετοί επιλυτές, θεωρώντας πλεονέκτημα τη δυνατότητα επικοινωνίας μέσω της κοινά αποδεκτής γλώσσας, την υποστηρίζουν εξ ολοκλήρου ή ως ένα βαθμό. Επομένως, αρκεί ένα πρόγραμμα να γράφει τα κατηγορήματα SMT σαν συμβολοσειρά SMTLIB στην είσοδο (standard input) ενός οποιουδήποτε συμβατού επιλυτή και να περιμένει την ανάγνωση της απάντησης ως συμβολοσειράς από την έξοδο (standard output) του επιλυτή.

Με την πάροδο των ετών, η αύξηση της υπολογιστικής ισχύος καθιστά δυνατή την εύρεση της απάντησης σε ολοένα και περισσότερα προβλήματα. Κατά συνέπεια, γίνεται εφικτή η αντιμετώπιση πολυπλοκότερων συνόλων περιορισμών. Η γλώσσα SMTLIB, ανταποκρινόμενη στις νέες συνθήκες, αναπτύσσεται εισάγοντας καινούρια στοιχεία, τα οποία βοηθούν και στην απλούστερη δήλωση των εκφράσεων. Οι κύριες διαφορές που έπαιξαν σημαντικό ρόλο στην εκπόνηση της παρούσας εργασίας είναι η συμπερίληψη τεσσάρων νέων εντολών. Από την έκδοση 2.5 της γλώσσας [6] είναι διαθέσιμες οι εντολές define-fun-rec και define-funs-rec, οι οποίες χρησιμεύουν στον ορισμό αναδρομικής συνάρτησης και συνόλου αμοιβαία αναδρομικών συναρτήσεων αντίστοιχα. Κατ’ επέκταση, στην τελευταία έκδοση 2.6 [7] εισήχθησαν οι εντολές declare-datatype και declare-datatypes, με τις οποίες είναι δυνατή η δήλωση αλγεβρικών τύπων δεδομένων, ενδεχομένως αμοιβαία αναδρομικών.

3 Σχετικά εργαλεία

Έχουν υλοποιηθεί διάφορα προγράμματα για εφαρμογή concolic testing, αλλά εξ όσων γνωρίζουμε, όλα ελέγχουν κώδικες σε προστακτικές ή αντικειμενοστρεφείς γλώσσες προγραμματισμού (όπως η C και η Java), μετατρέποντάς τους σε γλώσσα χαμηλότερου επιπέδου (bytecode ή και assembly), ενώ κάνουν χρήση ενός μόνο επιλυτή. Επομένως οι συνθήκες που περιγράφουν τα διάφορα μονοπάτια εκτέλεσης αφορούν απλούς τύπους δεδομένων όπως ακεραίους και bitvectors. Καταγράφουμε τα προγράμματα στην ακόλουθη λίστα:

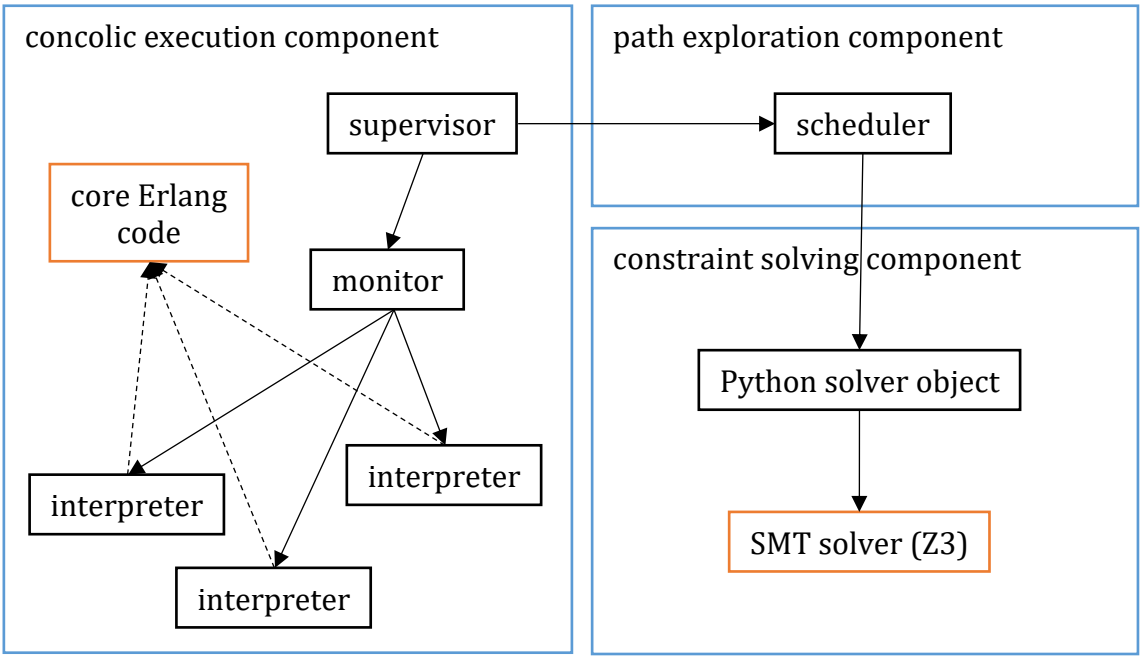
- DART [8]: Χρησιμοποιεί τον επιλυτή lp_solve [9], ο οποίος υποστηρίζει τη θεωρία ακεραίων αριθμών. Ελέγχει κώδικες γραμμένους σε γλώσσα C, αφού τους μεταφράσει σε CIL (C intermediate language) [10].
- CREST [11]: Χρησιμοποιεί τον επιλυτή Yices [12], ο οποίος υποστηρίζει τη θεωρία ακεραίων αριθμών. Ελέγχει προγράμματα γραμμένα σε γλώσσα C, μεταφράζοντάς τα σε CIL.
- jCUTE [13]: Χρησιμοποιεί επίσης τον επιλυτή lp_solve. Ελέγχει προγράμματα γραμμένα σε γλώσσα Java, μέσω του εργαλείου SOOP [14], το οποίο επεξεργάζεται κώδικα Java χαμηλού επιπέδου (Java bytecode).

Μία υλοποίηση του KLEE [15] εφαρμόζει συμβολικό έλεγχο αξιοποιώντας περισσότερους του ενός επιλυτές. Χρησιμοποιεί παράλληλα τους STP [16], Z3 [5] και Boolector [17] μέσα από ένα ενοποιημένο C++ API, διαλέγοντας τη λύση του ταχύτερου επιλυτή. Έχει σχεδιαστεί για έλεγχο προγραμμάτων σε γλώσσα C, αφού πρώτα τα μετατρέψει με το εργαλείο LLVM [18] σε γλώσσα assembly.

3.1 Πρόγραμμα CutEr

Η παρούσα εργασία εκπονήθηκε πάνω στο CutEr (Concolic Unit Testing Tool for Erlang) [3], ένα εργαλείο που αναπτύσσεται ακόμα και αρχικά χρησιμοποιούσε τον επιλυτή Z3 για να πραγματοποιήσει concolic testing. Ελέγχει όχι μόνο ακολουθιακά προγράμματα, αλλά και παράλληλα, τα οποία εκτελούνται σε πολλές διεργασίες, ενδεχομένως σε απομακρυσμένα μηχανήματα του δικτύου. Η σημαντική διαφορά του προγράμματος αυτού με όσα αναφέραμε προηγουμένως είναι ότι ελέγχει κώδικα Erlang, μιας συναρτησιακής γλώσσας υψηλού επιπέδου, έχοντας να διαχειριστεί ταίριασμα προτύπων (pattern matching), εξειδικευμένους αναδρομικούς τύπους (όπως λίστες, πλειάδες και λοιπές σύνθετες δομές) και συναρτήσεις ανωτέρου βαθμού (higher-order functions). Μάλιστα, δεν μετατρέπει τον κώδικα σε γλώσσα χαμηλού επιπέδου, παρά μόνο αξιοποιεί μία ενδιάμεση αναπαράσταση, τον πυρήνα (core) της Erlang. Ο κώδικας του εργαλείου είναι κατά κύριο λόγο γραμμένος σε Erlang και δευτερευόντως σε Python, αποτελείται δε ουσιαστικά από τρία μέρη (εικόνα 1).

3.1.1 Τμήματα εφαρμογής



εικόνα 1: Αρχιτεκτονική του CutEr

Το πρώτο μέρος ασχολείται με τον καθαυτό δυναμικό συμβολικό έλεγχο (concolic execution component). Αποτελείται από διεργασίες (interpreters) που εκτελούν τον ελεγχόμενο κώδικα πυρήνα Erlang (core Erlang code) για συγκεκριμένες τιμές των μεταβλητών εισόδου. Οι διεργασίες καταγράφουν τους συμβολικούς περιορισμούς που αντιστοιχούν στο τρέχον μονοπάτι εκτέλεσης και αφορούν τις παραμέτρους εισόδου. Ταυτόχρονα, άλλη διεργασία (monitor) ελέγχει τα αποτελέσματα και εξετάζει αν συνέβη κάποιο σφάλμα εκτέλεσης (runtime error) ή εξαίρεση που δεν αντιμετωπίστηκε (unhandled exception), εκτυπώνοντας την είσοδο για την οποία προέκυψε.

Το δεύτερο μέρος είναι υπεύθυνο για την εξερεύνηση όλων των προσπελάσιμων μονοπατιών εκτέλεσης μέχρι ένα βάθος αναζήτησης (path exploration component). Κάθε μονοπάτι εκτέλεσης είναι μία ακολουθία περιορισμών που αντιστοιχούν στις συνθήκες των εντολών διακλάδωσης της διαδρομής. Οι περιορισμοί συνοδεύονται από δύο ετικέτες προς τμήματα κώδικα: εκείνη που ανήκει στο μονοπάτι εκτέλεσης, με βάση την απόφαση που λήφθηκε στην εντολή διακλάδωσης και εκείνη που δεν ακολουθήθηκε. Εντοπίζουμε στη διεργασία του δρομολογητή (scheduler) τα μονοπάτια μεταβαίνοντας σε αυτό που ορίζεται ως επόμενο σύμφωνα με την ακόλουθη ευριστική μέθοδο: Κατευθυνόμαστε προς τις ετικέτες που δεν έχουμε επισκεφθεί, δίνοντας μεταξύ αυτών προτεραιότητα σε όσες βρίσκονται σε μικρότερο βάθος από τη ρίζα του δέντρου αναζήτησης. Εφαρμόζουμε, δηλαδή, διάσχιση κατά βάθος (breadth-first search, BFS).

Το τρίτο μέρος είναι γραμμένο σε Python και πραγματοποιεί επίλυση των συνόλων περιορισμών (constraint solving component) καλώντας έναν SMT επιλυτή (SMT solver), τον Z3, η επικοινωνία με τον οποίο επιτυγχάνεται μέσω του Python API που διαθέτει, του Z3Py. Ο εν λόγω επιλυτής υποστηρίζει σύνθετους τύπους δεδομένων, επιτρέποντας

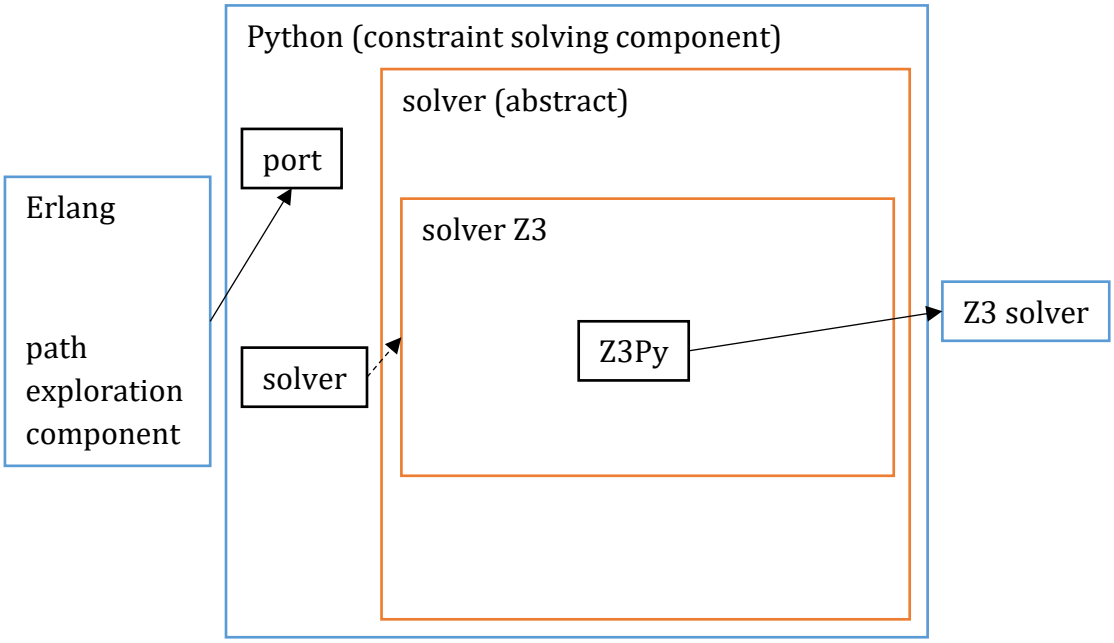
την άμεση αναπαράσταση των όρων της Erlang στο εσωτερικό του με απλές δηλώσεις και εντολές που προσφέρει η διεπαφή. Για κάθε είδος περιορισμού που συναντάται στον πυρήνα της Erlang υλοποιείται η αντίστοιχη μέθοδος στην κλάση Python του επιλυτή (Python solver). Εν κατακλείδι, αναφέρουμε πως η διαδικασία επίλυσης, σε περιπτώσεις εξαιρετικά πολύπλοκων συνόλων περιορισμού, αντιμετωπίζει τυχόν αποτυχία απόκρισης του επιλυτή με απλοποίηση του χώρου αναζήτησης λύσεων, σταθεροποιώντας μια ελεύθερη μεταβλητή στην τιμή που έλαβε κατά την προηγούμενη εκτέλεση του ελεγχόμενου κώδικα.

3.1.2 Οργάνωση Python

Το CutEr επιλέγει στη διεργασία του δρομολογητή (scheduler) του τμήματος εξερεύνησης μονοπατιών ένα μονοπάτι εκτέλεσης με βάση ένα αρχείο ίχνους (trace file) και ενεργοποιεί το τμήμα επίλυσης περιορισμών, όπου αρχικοποιείται η θύρα Erlang (port) και το αντικείμενο Python του επιλυτή (solver).

Η θύρα είναι υπεύθυνη για την επικοινωνία του CutEr με τον επιλυτή: δέχεται εντολές από την Erlang και της στέλνει δεδομένα από την Python. Επιπλέον, περιλαμβάνει συναρτήσεις που υλοποιούν τις ουσιαστικές λειτουργίες για το CutEr, όπως η αποστολή των περιορισμών στην Python, η εντολή της επίλυσης των περιορισμών και η λήψη του μοντέλου σε περίπτωση που το σύνολο των περιορισμών είναι ικανοποιήσιμο.

Υπάρχει μία βασική κλάση που κωδικοποιεί τους περιορισμούς στη γλώσσα του επιλυτή, του στέλνει τις εντολές και διαβάζει τα αποτελέσματα από αυτόν. Είναι αφηρημένη κλάση (abstract) και την επεκτείνει η υποκλάση Solver_Z3. Το αντικείμενο του επιλυτή ανήκει στην κλάση Solver_Z3 και επικοινωνεί με τον επιλυτή Z3 μέσω του Python API, του Z3Py (εικόνα 2).



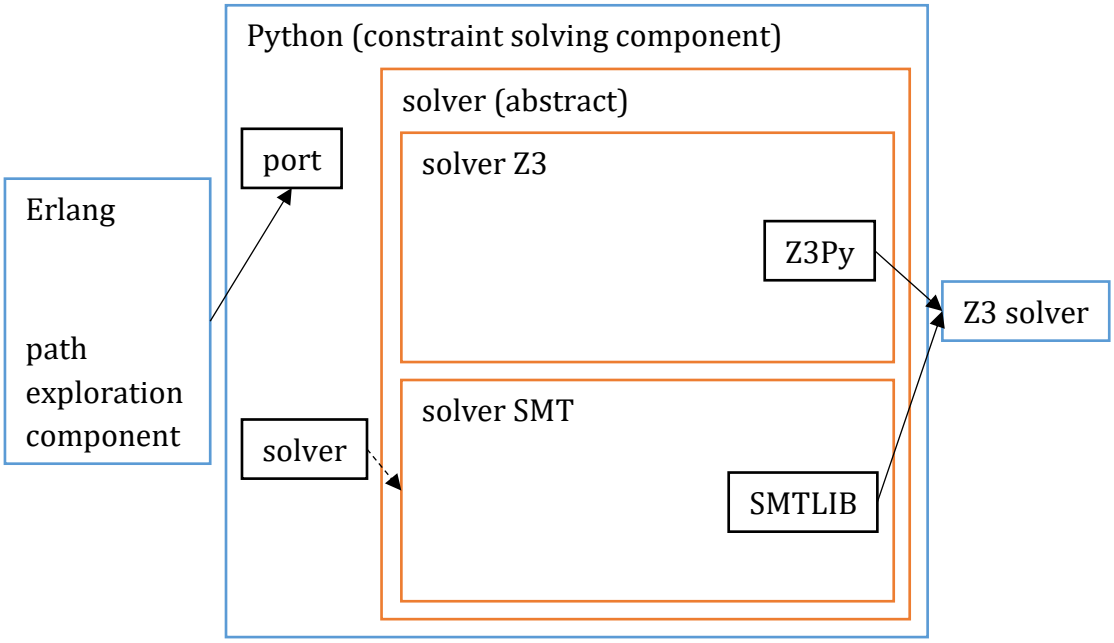
εικόνα 2: Επικοινωνία με τον επιλυτή Z3 μέσω του Z3Py

4 Υποστήριξη SMT επιλυτών στο CutEr

Στην αρχή το CutEr καλούσε τον επιλυτή Z3 μέσω του Python API του, δηλαδή ήταν σχεδιασμένο να μεταχειρίζεται αποκλειστικά τον επιλυτή Z3. Ένα πρώτο βήμα προς την ενσωμάτωση πολλών επιλυτών είναι η επικοινωνία του CutEr με τον ίδιο επιλυτή, αλλά μέσω της κοινά αποδεκτής γλώσσας SMTLIB. Στον παρόν κεφάλαιο περιγράφουμε αυτήν ακριβώς τη διαδικασία.

4.1 Οργάνωση Python

Το διάβασμα και την ανάλυση του ελεγχόμενου κώδικα Erlang αναλαμβάνει ο πυρήνας του CutEr, ο οποίος στη συνέχεια τρέχει εντολές Python, προκειμένου να μιλήσει με τον μαθηματικό επιλυτή. Οι εντολές οργανώνονται κυρίως σε μεθόδους κλάσεων ώστε να εκμεταλλευθούμε την κληρονομικότητα για το μοίρασμα και την επιλεκτική τροποποίηση τμημάτων κώδικα μεταξύ διαφορετικών κλάσεων (εικόνα 3).



εικόνα 3: Επικοινωνία με τον επιλυτή Z3 μέσω SMTLIB

Σε αντιδιαστολή με το προϋπάρχον σχήμα του CutEr (εικόνα 2), το τμήμα επίλυσης περιορισμών επιλέγει με ποιον τρόπο το αντικείμενο του επιλυτή θα επικοινωνεί με τον Z3. Κατασκευάζουμε τη νέα κλάση Solver_SMT που επεκτείνει και αυτή την αφηρημένη κλάση του επιλυτή, όπως και η Solver_Z3, αλλά υλοποιεί τις μεθόδους ώστε να κωδικοποιεί τους περιορισμούς και να διαβάζει αποτελέσματα χρησιμοποιώντας τη γλώσσα SMTLIB και όχι μέσω του Python API του επιλυτή.

Στο αντικείμενο του επιλυτή ορίζεται μία μεταβλητή για το χειρισμό της διεργασίας του επιλυτή. Η μεταβλητή επιτρέπει τη δημιουργία της διεργασίας του επιλυτή (init) και αναλαμβάνει την επικοινωνία μαζί του γράφοντας (write) στο standard input και διαβάζοντας (read) στο standard output της διεργασίας. Περιέχει μεθόδους για βασικές διαδικασίες της επίλυσης, όπως ο έλεγχος για το κατά πόσο είναι ικανοποιήσιμο ένα

σύνολο περιορισμών (check_sat), η λήψη της τιμής μιας έκφρασης (get_value) σε ένα μοντέλο που βρέθηκε να ικανοποιεί το σύνολο περιορισμών και ο τερματισμός της διεργασίας του επιλυτή (exit).

4.2 Παράδειγμα εκτέλεσης

Για να καταλάβουμε περίπου πώς πρέπει να ανταποκρίνεται το CutEr σε ένα απλό παράδειγμα κώδικα, επιλέγουμε τον SMT επιλυτή Z3 και θεωρούμε το αρχείο test1.erl (εικόνα 4):

```
-module(test1).
-export([non_neg/1]).

-spec non_neg(integer()) -> ok.
non_neg(N) when N < 0 -> error(bug);
non_neg(_) -> ok.
```

εικόνα 4: Απλό παράδειγμα Erlang

Ο παραπάνω κώδικας εμφανίζει σφάλμα όταν η συνάρτηση non_neg κληθεί με όρισμα έναν αρνητικό ακέραιο.

Ελέγχουμε τον κώδικα με το CutEr, εκτελώντας:

```
./cuter test1 non_neg '[0]'
```

Εκτελείται ο κώδικας με είσοδο το 0 και δημιουργείται το αντίστοιχο αρχείο ίχνους. Η συνθήκη σύγκρισης της παραμέτρου εισόδου με το 0 είναι ψευδής.

Το CutEr επιχειρεί να αντιστρέψει τη συνθήκη ότι η παράμετρος εισόδου είναι ακέραιος. Οι βασικές εντολές που στέλνονται στον επιλυτή είναι:

```
(declare-const x Term)
(assert (is-int x))
(assert (not (is-int x)))
(check-sat)
```

Δηλαδή δηλώνεται η παράμετρος x, επιβάλλεται ο περιορισμός τύπου (το x να είναι ακέραιος) και ακολουθεί η αντεστραμμένη συνθήκη (το x να μην είναι ακέραιος). Ο επιλυτής, όπως είναι προφανές, αποκρίνεται με "unsat" (ότι το σύνολο περιορισμών είναι μη ικανοποιήσιμο, καθώς δε γίνεται η είσοδος να είναι ακέραιος και να μην είναι, ταυτόχρονα). Το αποτέλεσμα προωθείται στο CutEr.

Έπειτα δοκιμάζει να αντιστρέψει τη δεύτερη συνθήκη, ότι η σύγκριση της παραμέτρου εισόδου με το 0 είναι αληθής.

```
(declare-const x Term)
(assert (is-int x))
(assert (< (iv x) 0))
(check-sat)
```

Πλέον, το σύνολο περιορισμών είναι ικανοποιήσιμο και ο επιλυτής παράγει ένα μοντέλο που να το ικανοποιεί. Ζητούμε την τιμή της παραμέτρου εισόδου σύμφωνα με το μοντέλο αυτό.

```
(get-value (x))
```

Η τιμή που επιστρέφεται συμβαίνει να είναι ο αρνητικός ακέραιος -1.

```
((x (int (- 1))))
```

Το CutEr εκτελεί τον υπό έλεγχο κώδικα με είσοδο το -1 και εντοπίζει το σφάλμα, δημιουργώντας το αντίστοιχο αρχείου ίχνους.

Η εξερεύνηση του κώδικα σταματά καθώς, στο απλό μας παράδειγμα, εξαντλήθηκαν όλα τα μονοπάτια εκτέλεσης.

Στο τέλος, τυπώνεται η είσοδος που οδήγησε σε σφάλμα. Η είσοδος αποκαλύπτει το είδος του σφάλματος, ότι δηλαδή εμφανίζεται για κάθε αρνητικό ακέραιο:

```
=== Inputs That Lead to Runtime Errors ===
#1 test1:non_neg(-1)
```

Στην πραγματικότητα, οι εντολές SMTLIB που γράφονται στην είσοδο του επιλυτή είναι πολυπλοκότερες, ώστε να υποστηρίζονται και πιο σύνθετα προγράμματα για έλεγχο. Ωστόσο, τις παραθέτουμε απλοποιημένες για ευκολία στην κατανόηση.

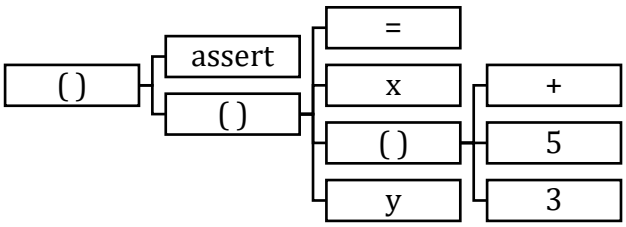
4.3 Σχήμα μετατροπής

Προκειμένου να επιλύσουμε ένα σύνολο περιορισμών, το οποίο έχει προκύψει από ένα μονοπάτι εκτέλεσης του κώδικα υπό εξέταση (στην περίπτωσή μας γραμμένο σε Erlang), πρέπει να τους μετατρέψουμε στην γλώσσα που μιλά ο επιλυτής. Εφόσον έχουμε την προοπτική να χρησιμοποιήσουμε πλήθος επιλυτών, θα επιλέξουμε όχι κάποιο συγκεκριμένο API ενός επιλυτή, αλλά την ευρέως αποδεκτή SMTLIB.

Η γλώσσα αυτή μεταχειρίζεται γράμματα, ψηφία και σύμβολα που διαχωρίζονται από λευκούς χαρακτήρες και ομαδοποιούνται με παρενθέσεις, ώστε να δομηθούν οι διάφορες εντολές με τις εμφωλευμένες εκφράσεις, όπως:

```
(assert (= x (+ 5 3) y))
```

Η εν λόγω μορφή ακολουθείται από τους επιλυτές είτε κατά την ανάγνωση εντολών είτε κατά την εκτύπωση αποτελεσμάτων. Παρατηρούμε ότι η μορφή αυτή είναι δενδρική:



Επομένως, για τη δημιουργία και αναγνώριση των συμβολοσειρών, διευκολύνει η αποθήκευσή τους σε αντίστοιχη αναδρομική δομή.

Η γλώσσα που επιλέξαμε να μεσολαβεί ανάμεσα στο CutEr και στον επιλυτή είναι η Python. Η απλούστερη υλοποίηση αυτής της δομής είναι η απεικόνιση των φύλλων με συμβολοσειρές (str) και των εσωτερικών κόμβων με λίστες (list). Το παραπάνω παράδειγμα γράφεται:

```
["assert", ["=", "x", ["+", "5", "3"], "y"]]
```

Η μετατροπή από τη δενδρική αυτή δομή σε SMTLIB ουσιαστικά είναι σειριοποίηση (μετατροπή σε συμβολοσειρά) και υλοποιείται με τη συνάρτηση serialize. Η αντίστροφη διαδικασία υλοποιείται στην unserialize. Ο κώδικάς τους παρουσιάζεται στο παράρτημα.

4.3.1 Αποκωδικοποίηση εκφράσεων

Η κλάση που μεσολαβεί χρειάζεται να αποκωδικοποιήσει τις εκφράσεις της Erlang που λαμβάνει από το πρόγραμμα ελέγχου. Αυτό επιτυγχάνεται στη μέθοδο decode. Τα επιμέρους στοιχεία των εκφράσεων τα λαμβάνει σαν τύπους της Python και τα μετατρέπει σε δενδρική δομή με την οικογένεια των μεθόδων build_*. Παραδείγματος χάρη, αν έχουμε να διαβάσουμε έναν ακέραιο:

```
if cc.is_int(data):
     return ["int", self.build_int(cc.get_int(data))]
```

όπου

```
def build_int(self, value):
     if value < 0:
          return ["-", str(-value)]
     else:
          return str(value)
```

4.3.2 Κωδικοποίηση εκφράσεων

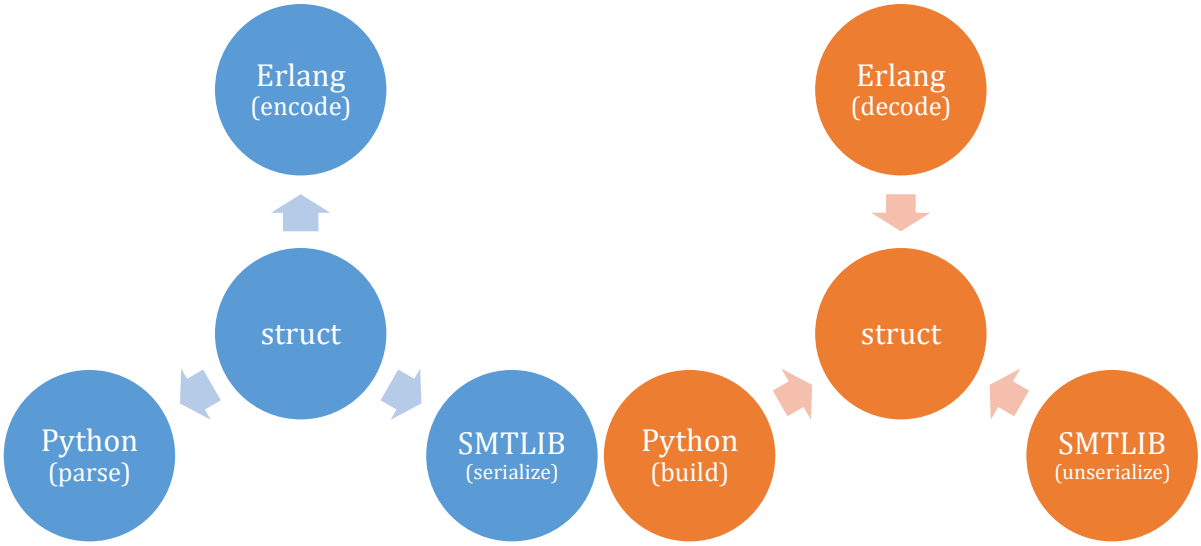
Η ακριβώς αντίστροφη διαδικασία, για την αναγνώριση της δενδρικής δομής και κωδικοποίησή της σε εκφράσεις της Erlang, υλοποιείται στη μέθοδο encode, η οποία καλεί τις διάφορες συναρτήσεις parse_* για τη δημιουργία των τύπων της Python. Στο αντίστοιχο παράδειγμα του ακεραίου:

```
if data[0] == "int":
     return cc.mk_int(self.parse_int(data[1]))
```

όπου

```
def parse_int(self, expr):
     if isinstance(expr, list):
          if expr[0] == "-" and len(expr) == 2:
               return -self.parse_int(expr[1])
     else:
          return int(expr)
```

Οι αναφερθείσες μετατροπές συνοψίζονται στο ακόλουθο σχήμα:



4.4 Τύποι δεδομένων

Στον επιλυτή πρέπει να δηλωθούν οι όροι που υποστηρίζονται από την Erlang. Μία μεταβλητή μπορεί να είναι οποιοσδήποτε όρος (term), εκτός αν υπάρχει σχετικός περιορισμός. Οπότε σαν περιπτώσεις όρων δηλώνουμε τους τύπους:

- ακέραιος (integer, σύντομο int)
- πραγματικός (real)
- άτομο (atom)
- λίστα (list)
- πλειάδα (tuple)
- διάνυσμα δυφίων (bitstring, σύντομο str)
- συνάρτηση (function, σύντομο fun)

Η δυνατότητα μία μεταβλητή να μπορεί να λάβει οποιοδήποτε τύπο επιτυγχάνεται με την SMTLIB εντολή declare-datatypes, η οποία υποστηρίζεται από την έκδοση 2.6 της γλώσσας [7]. Συνολικά η εντολή που στέλνουμε συνοψίζεται στο παράρτημα.

Οι αριθμητικοί τύποι integer και real δηλώνονται απλά.

```
(int (iv Int))
(real (rv Real))
```

Για παράδειγμα, ένας όρος t με την ακέραια τιμή -5 κωδικοποιείται ως

```
(int (- 5))
```

και η αριθμητική του τιμή λαμβάνεται με την έκφραση (iv t), ενώ ο πραγματικός αριθμός 3.14 κωδικοποιείται ως

```
(real 3.14)
```

και η έκφραση γράφεται (rv t).

4.4.1 Σύνθετοι τύποι

Ο τύπος atom δηλώνεται με τη χρήση γραμμικής αναδρομικής δομής ακεραίων (λίστα με τους κωδικούς των χαρακτήρων από τους οποίους αποτελείται το άτομο), οι τύποι tuple και list ως λίστες όρων και ο τύπος bitstring ως λίστα λογικών μεταβλητών.

```
(list (lv TList))
(tuple (tv TList))
(atom (av IList))
(str (sv SList))
```

όπου

```
(TList (tn) (tc (th Term) (tt TList)))
(IList (in) (ic (ih Int) (it IList)))
(SList (sn) (sc (sh Bool) (st SList)))
```

Το άτομο nil, η λίστα [9, 4] και το bitstring "01001" κωδικοποιούνται αντίστοιχα ως:

```
(atom (ic 110 (ic 105 (ic 108 in))))
(list (tc (int 9) (tc (int 4) tn)))
(str (sc false (sc true (sc false (sc false (sc true sn))))))
```

4.4.2 Τύπος συνάρτησης

Ο τύπος function δηλώνεται ως ακέραιος (μοναδικό αναγνωριστικό ανά συνάρτηση), σε αντιστοιχία με την αναπαράστασή του στο εσωτερικό ενός υπολογιστή. Αυτού του είδους η αναπαράσταση επιτρέπει τη δημιουργία (ενδεχομένως αμοιβαία) αναδρομικών συναρτήσεων στο επιστρεφόμενο μοντέλο. Διαφορετικά και μόνο η αναπαράστασή της θα οδηγούσε σε ατέρμονα συμβολοσειρά.

```
(fun (fv Int))
```

Επιπροσθέτως, στον επιλυτή ορίζονται δύο SMTLIB συναρτήσεις που απεικονίζουν το αναγνωριστικό στο πλήθος των ορισμάτων που δέχεται η συνάρτηση και στο σώμα της αντίστοιχα.

```
(declare-fun fa (Int) Int)
(declare-fun fm (Int) FList)
```

Ιδανικά θα έπρεπε να μπορούμε να υποστηρίξουμε κάθε συνάρτηση, ενδεχομένως με τη χρήση συναρτήσεων SMTLIB, ωστόσο μέχρι στιγμής οι επιλυτές αποκρίνονται με συναρτήσεις που ορίζονται σε πεπερασμένο σύνολο σημείων, ενώ για όλες τις υπόλοιπες εισόδους επιστρέφουν μία σταθερή τιμή. Κατόπιν εκτεταμένου πειραματισμού καταλήξαμε στη δήλωση του τύπου function ως αναδρομικής δομής ζευγών λίστας όρων με όρο.

```
(FList (fn) (fc (fx TList) (fy Term) (ft FList)))
```

Δεν προτιμήσαμε τη χρήση συνάρτησης από λίστα όρων σε όρο για τον εξής λόγο. Όταν ζητούσαμε τη λήψη της τιμής μιας παραμέτρου, η οποία είχε τύπο function, από το μοντέλο, ο επιλυτής αποκρινόταν με ένα διάνυσμα. Η αναπαράσταση του διανύσματος γινόταν με τη χρήση μιας βοηθητικής συνάρτησης, της οποίας η μορφή ήταν εξαιρετικά δύσκολο να αναγνωριστεί, καθώς περιείχε εκφράσεις ite (if-then-else) που ενέπλεκαν την παράμετρο εισόδου της βοηθητικής συνάρτησης ή το αποτέλεσμα μιας άλλης βοηθητικής συνάρτησης για την παράμετρο εισόδου.

Ως παράδειγμα αναφέρουμε τη συνάρτηση

$$f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}, \qquad f(2,0) = 0.5, \qquad f(-1,1) = -3.8$$

η οποία συμβαίνει να κωδικοποιείται με αναγνωριστικό 3, δηλαδή (fun 3). Τότε η έκφραση (fa 3) αποτιμάται ως 2 (το πλήθος των ορισμάτων της συνάρτησης) και η έκφραση (fm 3) ως:

```
(
      fc
      (tc (int 2) (tc (int 0) tn))
      (real 0.5)
      (
            fc
            (tc (int (- 1)) (tc (int 1) tn))
            (real (- 3.8))
            fn
      )
)
```

4.4.3 Εξειδικευμένοι τύποι

Οι εξειδικευμένοι (συνήθως αναδρομικοί) όροι που υποστηρίζονται από την Erlang δηλώνονται στον επιλυτή ως συνδυασμοί των παραπάνω τύπων. Ο ορισμός τους γίνεται

με διαδικασία που θα αναλυθεί στην ενότητα εφαρμογής του περιορισμού του τύπου κάθε μεταβλητής, όπως είναι δηλωμένος στην Erlang.

4.4.4 Λογικές μεταβλητές

Ο ενσωματωμένος τύπος δεδομένων λογικής μεταβλητής (boolean, σύντομο bool) που υπάρχει στη γλώσσα SMTLIB δεν αξιοποιείται καθώς στην Erlang οι λογικές σταθερές true και false ορίζονται ως atoms. Επομένως και στον επιλυτή θα χρησιμοποιούνται οι αντίστοιχες σταθερές ακολουθίες ακεραίων:

```
[116, 114, 117, 101]
```

```
[102, 97, 108, 115, 101]
```

Εάν οι σταθερές αυτές δηλώνονταν ως λογικές, ενώ θα είχαν πολύ συντομότερη κωδικοποίηση, θα έπρεπε να υπερφορτωθούν οι τελεστές σύγκρισης (καθώς το άτομο true ταυτίζεται με το λογικό true) και οι συναρτήσεις που σχετίζονται με τα άτομα, όπως η συνάρτηση που δίνει την κεφαλή (head) ενός ατόμου, καθώς η κεφαλή του λογικού false είναι ο ακέραιος 102, δηλαδή ο ASCII κωδικός του χαρακτήρα f.

4.5 Διεργασία επιλυτή

Ανάλογα με το ποιον επιλυτή διαλέξαμε, περνούμε και τις διάφορες παραμέτρους ώστε να αποκρίνεται στις ανάγκες μας. Οι παράμετροι αυτές είναι πρωτίστως η μορφή εισόδου και εξόδου (η SMTLIB) και δευτερευόντως ο μέγιστος χρόνος εκτέλεσης της διεργασίας (timeout) του επιλυτή. Κατόπιν αρκετών δοκιμών με το σύνολο των αρχείων ελέγχου της εργασίας, παρατηρήσαμε ότι στην συντριπτική πλειοψηφία των περιπτώσεων οι επιλυτές είτε αποκρίνονται (με επιτυχία ή αποτυχία ή αδυναμία εύρεσης λύσης) σε χρόνο λιγότερο του ενός δευτερολέπτου, είτε δεν αποκρίνονται ποτέ. Επιλέξαμε, λοιπόν, τα δύο δευτερόλεπτα ως το μέγιστο χρόνο, αφήνοντας λίγο ακόμα περιθώριο. Επιπλέον, ορίζουμε σημαίες για ενεργοποίηση ή απενεργοποίηση λειτουργιών.

Η διεργασία του επιλυτή καλείται με τη βοήθεια της μονάδας subprocess της Python [19], ώστε να επιτυγχάνεται εύκολα η επικοινωνία του επιλυτή με απεικόνιση των βασικών ρευμάτων εισόδου (stdin) και εξόδου (stdout) της διεργασίας σε σωληνώσεις (pipes).

```
self.process = subprocess.Popen(
     arguments,
     stdin=subprocess.PIPE,
     stdout=subprocess.PIPE,
     stderr=subprocess.PIPE,
     universal_newlines=True
)
```

Η κλήση δεν πραγματοποιείται άμεσα, αντιθέτως, πρώτα αναγνωρίζονται όλοι οι περιορισμοί από το αρχείο ίχνους, συσσωρεύονται σε μία λίστα εντολών και στη συνέχεια σειριοποιούνται και αποστέλλονται.

Οι αρχικές εντολές που πρόκειται να εγγραφούν στην είσοδο του επιλυτή είναι:

- ο ορισμός της λογικής του επιλυτή, μέσω της εντολής set-logic, ανάλογα με το εύρος της θεωρίας που απαιτείται για την κατανόηση των περιορισμών και την εύρεση λύσης
- η επιλογή (set-option :produce-models true) για την παραγωγή του μοντέλου, δηλαδή της λύσης, σε περίπτωση επιτυχίας
- ο ορισμός των τύπων δεδομένων, συνοδευόμενος με τις δηλώσεις των βοηθητικών συναρτήσεων απεικόνισης των όρων function στο πλήθος των ορισμάτων και στο σώμα της συνάρτησης.

4.6 Αποστολή παραμέτρων

Οι πρώτες εντολές που πρόκειται να περάσουμε στον επιλυτή είναι οι παράμετροι του κώδικα υπό εξέταση, για τις οποίες και επιθυμούμε να βρούμε μία ανάθεση τιμών που να ικανοποιεί τους περιορισμούς του αρχείου ίχνους. Καθεμιά παράμετρο τη δηλώνουμε στον επιλυτή με το όνομά της ανάμεσα στον χαρακτήρα "|", δηλαδή αν η παράμετρος είναι καταγεγραμμένη στο αρχείο ίχνους ως variable, στον επιλυτή θα στείλουμε την εντολή (declare-const |variable| term). Επιπλέον, αποθηκεύουμε την παράμετρο στη λίστα με τις (κύριες) παραμέτρους για να ξέρουμε, πρώτον, ότι έχει δηλωθεί στον επιλυτή (επομένως σε μεταγενέστερη εμφάνισή της σε κάποιο περιορισμό να μην τη δηλώσουμε εκ νέου) και, δεύτερον, για να έχουμε τη δυνατότητα αίτησης της τιμής της σε περίπτωση που ο επιλυτής αποκριθεί πως το σύνολο περιορισμών είναι ικανοποιήσιμο.

Στο σημείο αυτό διακρίνουμε τις κύριες παραμέτρους από τις βοηθητικές: Οι κύριες παράμετροι αντιστοιχούν κατά σειρά στις παραμέτρους εισόδου του προγράμματος υπό έλεγχο, ενώ οι βοηθητικές παράμετροι είναι εσωτερικές μεταβλητές που υπάρχουν στην αναπαράσταση του κώδικα υπό εξέταση σε γλώσσα πυρήνα Erlang. Η αποθήκευση μιας παραμέτρου στη λίστα με τις κύριες ή τις βοηθητικές παραμέτρους γίνεται ανάλογα με μία σημαία η οποία αλλάζει τιμή αμέσως μετά την ανάγνωση όλων των κυρίων παραμέτρων.

Κατά την αποστολή παραμέτρων, λαμβάνουμε από το CutEr την τιμή που είχε κάθε κύρια παράμετρος στην προηγούμενη εκτέλεση του εξεταζόμενου κώδικα, όπως έχει καταγραφεί στο αρχείο ίχνους. Τις αντιστοιχίσεις παραμέτρων και τιμών τις αποθηκεύουμε σε μία λίστα, προκειμένου να τις χρησιμοποιήσουμε σε ενδεχόμενη αποτυχία απόκρισης του επιλυτή.

4.7 Περιορισμοί τύπου

Μία λειτουργία της Erlang είναι η επιβολή περιορισμού στον τύπο των παραμέτρων εισόδου και αποτελέσματος μίας συνάρτησης. Τέτοιου είδους περιορισμοί οφείλουν να

περαστούν και στον επιλυτή, προκειμένου να ψάξει για αποδεκτές λύσεις. Κατά την ανάπτυξη του εργαλείου ελέγχου λαμβάνουμε υπόψη μόνο τον περιορισμό των παραμέτρων εισόδου, χωρίς αυτό να σημαίνει πως είναι δύσκολη η εφαρμογή περιορισμού τύπου στην επιστρεφόμενη τιμή μιας συνάρτησης. Οι περιορισμοί τύπου μπορούν να επιβληθούν αναδρομικά και στα στοιχεία ενός σύνθετου τύπου (όπως, για παράδειγμα, η λίστα και η συνάρτηση). Υποστηρίζονται οι ακόλουθοι περιορισμοί:

- χωρίς περιορισμό (any)· απλά προσθέτουμε το κατηγόρημα true στη λίστα των περιορισμών που αποστέλλονται στον επιλυτή
- ακέραια σταθερά (integer literal)
- σταθερό άτομο (atom literal)
- ακέραιος (integer)
- πραγματικός (float)
- άτομο (atom)
- λίστα, ενδεχομένως μη κενή (list και nonempty list)· για όλα τα στοιχεία της λίστας ισχύει ο ίδιος περιορισμός τύπου
- πλειάδα, ενδεχομένως με περιορισμό του τύπου κάθε στοιχείου της (tuple και tupledet)· κάθε στοιχείο της πλειάδας μπορεί να έχει διαφορετικό περιορισμό τύπου
- ένωση (union)· συνδυασμός δύο ή περισσοτέρων τύπων
- εύρος (range)· ακέραιος, περιορισμένος προαιρετικά από ένα κάτω φράγμα ή και ένα άνω φράγμα
- διάνυσμα δυφίων (bitstring)· ένα τέτοιο διάνυσμα αποτελείται από $n \cdot k + m$ ακριβώς λογικές μεταβλητές, όπου n και m σταθεροί μη αρνητικοί ακέραιοι και k μη αρνητικός ακέραιος
- συνάρτηση, ενδεχομένως με περιορισμό των παραμέτρων εισόδου (generic function και complete function)· από εδώ προκύπτει περιορισμός για το πλήθος των παραμέτρων εισόδου, ενώ επιβάλλεται και κοινός περιορισμός σε κάθε ζευγάρι λίστας όρων με όρο
- τύποι ορισμένοι από το χρήστη (user defined types)

Οι απλοί περιορισμοί τύπου (όπως integer, float, tuple) επιβάλλονται με μία απλή έκφραση, όπως (is-real x). Στο εύρος χρησιμοποιείται σύζευξη, όπως (and (is-int x) (>= x 273) (<= x 373)). Στην ένωση χρησιμοποιείται διάζευξη, όπως (or (is-list x) (is-tuple x)).

4.7.1 Σύνθετοι τύποι

Για τους υπόλοιπους, σύνθετους περιορισμούς τύπου χρησιμοποιούμε σύζευξη και μία ή περισσότερες SMTLIB βοηθητικές συναρτήσεις. Πριν επιτευχθεί η επικοινωνία του CutEr με τον επιλυτή μέσω SMTLIB, η επιβολή του περιορισμού τύπου κάθε στοιχείου μιας αναδρομικής δομής (όπως οι λίστες) γινόταν κάθε φορά που εμφανιζόταν κάποιος επιπρόσθετος περιορισμός σε καθένα στοιχείο. Δηλαδή, σε μία λίστα ακεραίων, ο περιορισμός ότι ένα στοιχείο είναι ακέραιος λαμβανόταν υπόψη όταν αναφερόμαστε ειδικά σε εκείνο το στοιχείο, επομένως υπήρχε περίπτωση για ορισμένα στοιχεία να μην

συνυπολογιστεί η ύπαρξη του περιορισμού και να λάβουμε ένα ασυνεπές μοντέλο. Πλέον, η αξιοποίηση των αναδρομικών συναρτήσεων εξακριβώνει την ορθότητα του επιστρεφόμενου μοντέλου σε σχέση με τους περιορισμούς τύπου, παρόλο που επιβαρύνεται σημαντικά η πολυπλοκότητα του συνόλου περιορισμών.

Για παράδειγμα λαμβάνουμε έναν όρο t με περιορισμό μη κενής λίστας ακεραίων. Ορίζουμε τη βοηθητική αναδρομική συνάρτηση f1 ως εξής:

```
(define-fun-rec f1 ((l TList)) Bool (
     or
     (is-tn l)
     (and (is-tc l) (is-int (th l)) (f1 (tt l)))
))
```

Αυτή η συνάρτηση επιστρέφει true για κάθε Tlist που αποτελείται αποκλειστικά από ακεραίους. Τότε για τον όρο t επιβάλλουμε:

```
(assert (and (is-list t) (is-tc (lv t)) (f1 (lv t))))
```

Οι αναδρομικές συναρτήσεις είναι απαραίτητες για την επιβολή ενός οποιουδήποτε περιορισμού σε κάποιον όρο που από τη φύση του είναι αναδρομικός. Οι εντολές που ορίζουν αναδρομικές συναρτήσεις στην SMTLIB (define-fun-rec, define-funs-rec) υποστηρίζονται από την έκδοση 2.5 της γλώσσας [6]. Τα ονόματά τους τα παράγουμε αυτόματα, χρησιμοποιώντας έναν αύξοντα αριθμό για καθεμία.

Παρατηρήσαμε πως σε περίπτωση που έχουμε, για παράδειγμα, δύο περιορισμούς λίστας ακεραίων, η Python έστελνε στον επιλυτή δύο ίδιες βοηθητικές συναρτήσεις, αλλά προφανώς με διαφορετικό όνομα. Σε πιο περίπλοκους εξεταζόμενους κώδικες αυτό το φαινόμενο οδηγούσε τον επιλυτή σε αδυναμία απόκρισης, λόγω της σημαντικής αύξησης της πολυπλοκότητας του μοντέλου και αντιμετωπίστηκε με το να ορίζουμε τις συναρτήσεις με το ίδιο σώμα μία φορά. Χρησιμοποιούμε, λοιπόν, ένα λεξικό (dictionary), με κλειδί το χαρακτηριστικό της συνάρτησης, εν προκειμένω (is-int (th l)), και αντιστοιχισμένη τιμή το όνομα της συνάρτησης, εν προκειμένω f1. Κατά συνέπεια, πριν ορίσουμε νέα αναδρομική συνάρτηση, εξετάζουμε αν υπάρχει στο λεξικό συνάρτηση με το χαρακτηριστικό κλειδί. Καταρτίζουμε διαφορετικά λεξικά για τα διαφορετικά είδη συναρτήσεων που επιβάλλουν περιορισμό τύπου, όπως λεξικό για συναρτήσεις περιορισμού τύπου λίστας, λεξικό για συναρτήσεις περιορισμού τύπου συνάρτησης, ώστε να απαιτείται η ελάχιστη δυνατή πληροφορία στο κλειδί.

Η διαδικασία εφαρμογής περιορισμού τύπου atom ή bitstring γίνεται με τη βοήθεια δύο αναδρομικών συναρτήσεων που αναλύονται στο παράρτημα. Πριν τη χρήση των αναδρομικών συναρτήσεων δεν γινόταν έλεγχος για το εύρος τιμών των στοιχείων ενός atom, ενώ για τα bitstrings χρειαζόταν να αποθηκεύουμε επιπροσθέτως και το πλήθος των στοιχείων τους. Συναρτήσεις όπως οι δύο προηγούμενες δεν χρειάζεται να δηλώνονται στον επιλυτή κάθε φορά, παρά μόνο όταν υπάρχει ανάγκη εφαρμογής των αντίστοιχων περιορισμών. Για το λόγο αυτό, τις δηλώνουμε στον επιλυτή μόνο εφόσον

παραστεί ανάγκη και αποθηκεύουμε σε μία μεταβλητή λίστα την πληροφορία ότι δηλώθηκαν, ώστε να μην δηλωθούν εκ νέου σε μεταγενέστερο περιορισμό. Κατ' αυτό τον τρόπο στέλνουμε στον επιλυτή μόνο τις απολύτως απαραίτητες εντολές, απλοποιώντας το χώρο καταστάσεων που πρέπει να ψάξει και ελαχιστοποιώντας την πιθανότητα να λάβουμε απόκριση αδυναμίας απόφασης εάν το σύνολο περιορισμών είναι ικανοποιήσιμο ή μη ικανοποιήσιμο.

4.7.2 Αναδρομικοί τύποι

Μία σημαντική λειτουργία που προστέθηκε στο εργαλείο ελέγχου μέσα από την ανάπτυξη της παρούσας εργασίας είναι η υποστήριξη αναδρομικών τύπων ορισμένων από το χρήστη (οι μη αναδρομικοί μπορούν να αναλυθούν με τη βοήθεια των προηγούμενων τύπων). Ακολουθεί ένα παράδειγμα αναδρομικού τύπου ορισμένου από το χρήστη:

```
-type ctree() :: nil | {integer(), ctree(), ctree()}.
```

Δηλαδή ο όρος t μπορεί να είναι είτε το άτομο nil είτε μία πλειάδα τριών στοιχείων, το πρώτο της οποίας είναι ένας ακέραιος και τα υπόλοιπα είναι (αναδρομικά) ctree. Για την εφαρμογή του περιορισμού αυτού ορίζουμε τη συνάρτηση:

```
(define-fun-rec ctree ((t Term)) Bool (
     or
     (= t (atom (ic 110 (ic 105 (ic 108 in)))))
     (
          and
          (is-tuple t)
          (is-tc (tv t))
          (is-int (th (tv t)))
          (is-tc (tt (tv t)))
          (ctree (th (tt (tv t))))
          (is-tc (tt (tt (tv t))))
          (ctree (th (tt (tt (tv t)))))
          (is-tn (tt (tt (tt (tv t)))))
     )
))
```

Καθώς υπάρχει η δυνατότητα δύο τύποι να είναι αμοιβαία αναδρομικοί, οι αναδρομικές συναρτήσεις περιορισμού τύπου ορισμένου από το χρήστη δηλώνονται όλες στον επιλυτή με μία εντολή, τη define-funs-rec, ακριβώς μετά τη δήλωση των κυρίων παραμέτρων και πριν την επιβολή των περιορισμών τύπου. Μετά τον καταρτισμό της εντολής αυτής, οι υπόλοιπες αναδρομικές συναρτήσεις δεν γίνεται να περιλαμβάνουν αμοιβαία αναδρομή, οπότε δηλώνονται με ξεχωριστές εντολές define-fun-rec.

4.8 Λοιποί περιορισμοί

Εφόσον για κάθε παράμετρο εισόδου έχουμε επιβάλει ένα περιορισμό τύπου, μπορούμε να προχωρήσουμε στην εφαρμογή των περιορισμών του εξεταζόμενου κώδικα που

αντιστοιχούν στο τρέχον μονοπάτι εκτέλεσης. Για κάθε είδος περιορισμού καλείται η αντίστοιχη μέθοδος στην Python, η οποία, αφού δηλώσει τις μη δηλωμένες βοηθητικές μεταβλητές που εμπλέκονται στον περιορισμό, προσθέτει στη λίστα των εντολών τον ανάλογο περιορισμό. Για παράδειγμα, αν στο μονοπάτι εκτέλεσης υπάρχει ο περιορισμός της μεταβλητής t να είναι ο ακέραιος 12, καλείται η μέθοδος match_equal, η οποία προσθέτει στις εντολές προς αποστολή στον επιλυτή την εντολή:

```
(assert (= t (int 12)))
```

Υπάρχει πλήθος μεθόδων που υλοποιεί τους διάφορους περιορισμούς που υποστηρίζονται από τη γλώσσα του εξεταζόμενου κώδικα. Οι περισσότερες μέθοδοι εμφανίζονται και στην αντεστραμμένη μορφή τους (reversed), καθώς τα νέα μονοπάτια εκτέλεσης προκύπτουν με αντιστροφή ενός συνόλου περιορισμών. Δεν χρησιμοποιούμε απευθείας άρνηση για απλοποίηση εκφράσεων με διπλή άρνηση, όπως (not (not (= t (int 12)))). Εμείς έχουμε μεθόδους όπως:

- ισότητα ή ανισότητα όρων
- ένας όρος να μην είναι λίστα ή να είναι λίστα, κενή ή μη κενή
- εξίσωση όλων των στοιχείων μιας λίστας με συγκεκριμένους όρους
- μία λίστα να έχει συγκεκριμένη κεφαλή ή / και ουρά
- ένας όρος να μην είναι πλειάδα ή να είναι πλειάδα, ενδεχομένως με καθορισμένο πλήθος στοιχείων
- εξίσωση όλων των στοιχείων μιας πλειάδας με συγκεκριμένους όρους
- μετατροπή μιας λίστας σε πλειάδα και αντιστρόφως
- κατασκευή bitstring με μετατροπή αριθμού ή με συγχώνευση πολλών δυαδικών ψηφίων
- ένας όρος να μην είναι bitstring, ή να είναι bitstring, ενδεχομένως κενό ή μη κενό
- ένα bitstring να έχει συγκεκριμένο πρόθεμα και ενδεχομένως συγκεκριμένη ουρά
- ορισμός πλήθους ορισμάτων για μία συνάρτηση
- η τιμή μιας συνάρτησης για κάποια είσοδο να ισούται με συγκεκριμένο όρο
- ένας όρος να είναι το άτομο true ή το άτομο false
- ένα atom να είναι κενό (να αποτελείται από μηδενικό πλήθος χαρακτήρων) ή να έχει συγκεκριμένη κεφαλή (πρώτο χαρακτήρα) ή ουρά
- ένας όρος να είναι integer ή real ή αριθμός (number), δηλαδή οποιοδήποτε από τα δύο
- ένας όρος είναι το αποτέλεσμα μιας αριθμητικής πράξης δύο αριθμητικών όρων (πρόσθεση, αφαίρεση, πολλαπλασιασμός, διαίρεση, ακέραια διαίρεση, υπόλοιπο ακέραιας διαίρεσης, δύναμη)
- ένας όρος είναι μικρότερος ή αντίθετος (αριθμητικά) ενός άλλου όρου
- ένας όρος είναι το αποτέλεσμα μιας bitwise (δυφίο προς δυφίο) μαθηματικής πράξης δύο ακεραίων
- ένας όρος είναι η μετατροπή ενός ακεραίου σε πραγματικό ή ενός πραγματικού σε ακέραιο, με αποκοπή (truncation) του κλασματικού μέρους

Προς το παρόν, το εργαλείο ελέγχου κατά την κατασκευή ή σύγκριση bistrings θεωρεί σταθερό το πλήθος των bits στο οποίο κωδικοποιείται ένας αριθμός (σταθερός ή άγνωστος), επομένως οι αντίστοιχες μέθοδοι δεν χρησιμοποιούν κάποια αναδρομική συνάρτηση, αλλά προσθέτουν σε μία σύζευξη τους περιορισμούς για κάθε bit.

Ως προς τον περιορισμό τιμής μιας συνάρτησης για συγκεκριμένα ορίσματα, αυτή επιβάλλεται μέσω μιας SMTLIB αναδρομικής συνάρτησης, καθώς, όπως έχουμε αναφέρει προηγουμένως, η αναπαράστασή της γίνεται σαν λίστα ζευγαριών ορισμάτων και αποτελεσμάτων. Ο ορισμός της περιλαμβάνεται στο παράρτημα.

Οι αριθμητικές πράξεις πρόσθεση, αφαίρεση και πολλαπλασιασμός, όταν εφαρμόζονται σαν περιορισμοί, εμφανίζουν πολλές ομοιότητες: Το αποτέλεσμα της πράξης (σαν τύπος) είναι ακέραιος αν και μόνο αν τα δύο τελούμενα είναι ακέραιοι, διαφορετικά είναι πραγματικός. Επίσης, η συνθήκη κωδικοποιείται με τον ίδιο τρόπο, αν εξαιρέσουμε το σύμβολο του τελεστή της κάθε πράξης. Συνεπώς, χρησιμοποιούμε την ίδια μέθοδο για την εξαγωγή του περιορισμού, στην οποία παρέχουμε επιπροσθέτως τη συμβολοσειρά (μήκους ενός χαρακτήρα) που αντιστοιχεί στην εκάστοτε πράξη.

Από την άλλη πλευρά, δεν υπάρχει SMTLIB συνάρτηση υπολογισμού της δύναμης, επομένως χρειάστηκε να υλοποιήσουμε μία συνάρτηση υπολογισμού της, με αναδρομή στον (ακέραιο) εκθέτη. Παρομοίως, ο ορισμός της αναλύεται στο παράρτημα. Γενικά όμως οι επιλυτές που κυκλοφορούν δεν ανταποκρίνονται ικανοποιητικά σε σύνολα μη γραμμικών περιορισμών (πολλαπλασιασμοί, διαιρέσεις, δυνάμεις), αφού το ίδιο το πρόβλημα της μη γραμμικής αριθμητικής πραγματικών αριθμών είναι εξαιρετικά ακριβό υπολογιστικά, ενώ το πρόβλημα της μη γραμμικής αριθμητικής ακεραίων αριθμών, από θεωρητικής άποψης, είναι μη αποφασίσιμο [20].

Μία ακόμη δυσκολία που αντιμετωπίσαμε ήταν η υλοποίηση των περιορισμών που σχετίζονται με bitwise πράξεις μεταξύ δύο ακεραίων. Οι πράξεις αυτές είναι το "και" (bitwise and), το "ή" (bitwise or) και το "αποκλειστικό ή" (bitwise xor) – η πράξη "όχι" (bitwise not) μετατρέπεται εσωτερικά από το πρόγραμμα ελέγχου σύμφωνα με τον ορισμό του αντιθέτου ενός αριθμού ως το συμπλήρωμα του δύο. Το πρόβλημα έγκειται στο ότι η γλώσσα SMTLIB υποστηρίζει αυτές τις πράξεις, αλλά τα τελούμενα πρέπει να είναι διανύσματα bits (bitvectors) σταθερού μήκους. Η πρώτη προσέγγιση ήταν να ορίσουμε ένα μέγιστο μήκος διανύσματος, να μετατρέψουμε τους ακεραίους σε διανύσματα και να εφαρμόσουμε την κάθε πράξη. Κατ' αυτό τον τρόπο δεν υποστηρίζαμε αριθμούς μεγαλύτερης τάξης μεγέθους της μέγιστης επιτρεπτής, ενώ συχνά δαπανούσαμε σημαντικά περισσότερους υπολογιστικούς πόρους απ' όσους απαιτούνταν στην πραγματικότητα. Επομένως, υλοποιήσαμε τις πράξεις αναδρομικά, ως πράξεις μεταξύ ακεραίων. Οι λεπτομέρειες του ορισμού των εν λόγω συναρτήσεων παρουσιάζονται στο παράρτημα.

4.9 Επίλυση περιορισμών

Από τη στιγμή που έχουν προστεθεί στη λίστα εντολών όλες οι εντολές τις οποίες απαιτεί ο επιλυτής για την εύρεση ενός μοντέλου που ικανοποιεί τους περιορισμούς (ρυθμίσεις,

δηλώσεις μεταβλητών και βοηθητικών συναρτήσεων, περιορισμοί τύπου και μονοπατιού εκτέλεσης), μένει να αποσταλούν σειριοποιημένες στον επιλυτή. Το πρώτο βήμα που κάνουμε είναι να δημιουργήσουμε τη διεργασία του επιλυτή, καλώντας τον με τις κατάλληλες παραμέτρους γραμμής. Έπειτα, γράφουμε με τη σειρά τις εντολές στην είσοδο του επιλυτή και στο τέλος γράφουμε την εντολή (check-sat), περιμένοντας την απόκριση του επιλυτή. Ο επιλυτής αποκρίνεται στην έξοδό του με μία γραμμή που περιέχει μία από τις ακόλουθες λέξεις:

- sat (ικανοποιήσιμο). Το σύνολο περιορισμών που εστάλη στον επιλυτή είναι ικανοποιήσιμο. Εφόσον ο επιλυτής έχει ενεργοποιημένη την επιλογή produce-models, μπορούμε στη συνέχεια να του ζητήσουμε και ένα μοντέλο που να ικανοποιεί το σύνολο περιορισμών.
- unsat (μη ικανοποιήσιμο). Το σύνολο περιορισμών που εστάλη στον επιλυτή δεν είναι ικανοποιήσιμο, δηλαδή δεν υπάρχει μοντέλο στο καθορισμένο πεδίο λογικής που να το ικανοποιεί. Αυτό το αποτέλεσμα σημαίνει ότι το μονοπάτι εκτέλεσης που ελέγχουμε δεν είναι δυνατόν να ακολουθηθεί για καμία είσοδο στο εξεταζόμενο πρόγραμμα, συνεπώς ενδεχόμενα σφάλματα που εντοπίζονται στο εν λόγω μονοπάτι είναι μη προσεγγίσιμα.
- unknown ή timeout (άγνωστο). Ο επιλυτής δεν μπορεί να αποφανθεί για το κατά πόσο το σύνολο περιορισμών είναι ικανοποιήσιμο ή όχι, καθώς είτε αδυνατεί να αναζητήσει μοντέλα για το είδος και την πολυπλοκότητα των περιορισμών είτε δεν πρόλαβε στο ορισμένο χρονικό διάστημα. Εφόσον πράγματι θέλουμε να πάρουμε ένα αποτέλεσμα, οφείλουμε να απλοποιήσουμε το σύνολο περιορισμών και να επαναλάβουμε τη διαδικασία.

Μετά την εμπλοκή του επιλυτή, ζητούμε το μοντέλο που ικανοποιεί τους περιορισμούς στην περίπτωση που είναι ικανοποιήσιμο και τον τερματίζουμε, γράφοντας στην είσοδο του επιλυτή την εντολή (exit), ενώ αναφέρουμε στο πρόγραμμα ελέγχου για το κατά πόσο είναι ικανοποιήσιμο ή όχι το σύνολο των περιορισμών, σύμφωνα με τον επιλυτή.

4.10 Λήψη μοντέλου

Στην περίπτωση που ο επιλυτής απάντησε με sat, έχει υπολογίσει ένα μοντέλο που ικανοποιεί τους περιορισμούς. Για κάθε κύρια παράμετρο, ζητούμε από τον επιλυτή την τιμή που της αναθέτει στο μοντέλο αυτό. Για παράδειγμα, αν η κύριες παράμετροι είναι οι x και y, γράφουμε στην είσοδο του επιλυτή με τη σειρά τις παρακάτω εντολές:

```
(get-value (x))
(get-value (y))
```

Στις εντολές αυτές ο επιλυτής απαντά στην έξοδό του με μία SMTLIB έκφραση, σε μία ή περισσότερες γραμμές. Δεν μπορούμε να διαβάσουμε μέχρι να εξαντληθούν οι γραμμές στην έξοδο του επιλυτή, καθώς τότε δεν θα είχαμε τη δυνατότητα περαιτέρω διαδραστικότητας με τον επιλυτή. Επομένως, η ανάγνωση από την έξοδο γίνεται γραμμή προς γραμμή, μέχρι η συμβολοσειρά που έχει αναγνωσθεί να αντιστοιχεί σε έγκυρη έκφραση. Στην υλοποίησή μας η εγκυρότητα της έκφρασης ελέγχεται εύκολα με το κατά

πόσο είναι ισοσταθμισμένες οι παρενθέσεις, δηλαδή αν οι παρενθέσεις που ανοίγουν, "(", είναι ίσες στο πλήθος με τις παρενθέσεις που κλείνουν, ")".

Ας υποθέσουμε ότι στην εντολή (get-value (x)) ο επιλυτής απάντησε με ((x (int 12))). Μετατρέπουμε την SMTLIB έκφραση αυτή στην αντίστοιχη [["x", ["int", "12"]]]. Συνεπώς, στο πρόγραμμα ελέγχου αναφέρουμε ότι στο μοντέλο που ικανοποιεί το σύνολο περιορισμών η κύρια παράμετρος x παίρνει την ακέραια τιμή 12.

Υπάρχουν περιπτώσεις όπου ο επιλυτής επιστρέφει την τιμή μιας κύριας μεταβλητής με τη χρήση μιας βοηθητικής μεταβλητής και την SMTLIB έκφραση let. Αυτές τις εκφράσεις τις αναγνωρίζουμε αμέσως μετά την αποσειριοποίηση της απόκρισης του επιλυτή και αντικαθιστούμε τις βοηθητικές μεταβλητές με τις εκφράσεις τους πριν την αποκωδικοποίησή τους. Ένα σημείο που θέλει προσοχή είναι ότι ενδεχομένως προκύπτουν εμφωλευμένες εκφράσεις let. Η εν λόγω διαδικασία υλοποιείται στη συνάρτηση expand_lets, η οποία παρατίθεται στο παράρτημα.

4.11 Σταθεροποίηση παραμέτρου

Υπάρχουν περιπτώσεις στις οποίες τα σύνολα περιορισμών που προκύπτουν δεν επιλύονται. Θεωρούμε το αρχείο test2.erl (εικόνα 5):

```
-module(test2).
-export([non_lin/2]).

-spec non_lin(integer(), integer()) -> ok.
non_lin(X, Y) ->
     case X * X * Y of
          35 -> error(bug);
          _ -> ok
     end.
```

εικόνα 5: Δυσεπίλυτο σύνολο περιορισμών

Είναι προφανές ότι το σφάλμα εμφανίζεται για $x^2 \cdot y = 35$. Βεβαίως, μπορούμε εύκολα να παρατηρήσουμε ότι για $|x| = 1$ και $y = 35$ εντοπίζεται το σφάλμα. Ωστόσο, ο περιορισμός αυτός είναι μη γραμμικός και είναι πολύ πιθανό για έναν επιλυτή να μην μπορεί να αποφανθεί ως προς την ικανοποιησιμότητά του. Το ίδιο συμβαίνει και όταν έχουμε πολύπλοκα σύνολα περιορισμών, αποτελούμενα από πλήθος εξαρτημένων περιορισμών ή και αναδρομικές συναρτήσεις.

Επιλέγουμε τον CVC4 ως επιλυτή και ελέγχουμε τον κώδικα με το CutEr, δίνοντας:

```
./cuter test2 non_lin '[1, 1]'
```

Αρχικά, το μονοπάτι εκτέλεσης οδηγείται στην τελευταία (catch all) διακλάδωση (branch) του case, καθώς οι παράμετροι έχουν τιμές $x = 1$ και $y = 1$. Στη συνέχεια, αντιστρέφεται η συνθήκη του case:

```
(assert (= (* x x y) 35))
```

Ο επιλυτής αποκρίνεται με αδυναμία απόφασης ως προς την ικανοποιησιμότητα του προβλήματος (διαλέξαμε επίτηδες την έκδοση 1.5 του επιλυτή CVC4 καθώς η έκδοση 4.5 του επιλυτή Z3 το χειρίζεται επιτυχώς). Δοκιμάζουμε, λοιπόν, να σταθεροποιήσουμε την πρώτη παράμετρο στην τιμή που είχε κατά την προηγούμενη εκτέλεση:

```
(assert (= x (int 1))
```

Πλέον, το πρόβλημα έχει απλοποιηθεί σημαντικά και ο επιλυτής απαντά ότι το σύνολο περιορισμών είναι ικανοποιήσιμο. Ζητούμε με τη σειρά τις τιμές των παραμέτρων στο μοντέλο που βρέθηκε.

```
(get-value (x))
((x (int 1)))
(get-value (y))
((y (int 35)))
```

Το CutEr εκτελεί τον εξεταζόμενο κώδικα για τις παραπάνω τιμές και οδηγείται στο σφάλμα, το οποίο και τυπώνει μετά την εξερεύνηση όλων των μονοπατιών εκτέλεσης.

```
=== Inputs That Lead to Runtime Errors ===
#1 test2:non_lin(1, 35)
```

Σημειώνουμε πως, αν είχαμε καλέσει το CutEr με διαφορετικές αρχικές παραμέτρους, χρησιμοποιώντας την ίδια έκδοση του CVC4, το σφάλμα δεν θα εντοπιζόταν.

```
./cuter test2 non_lin '[0, 1]'
No Runtime Errors Occured
```

Γενικά, εάν ο επιλυτής δεν μπορεί να αποφανθεί για το κατά πόσο το σύνολο περιορισμών είναι ικανοποιήσιμο ή όχι, εφόσον θέλουμε να προχωρήσει ο έλεγχος του προγράμματος, είμαστε αναγκασμένοι να καταφύγουμε σε συμβιβασμό. Πιο συγκεκριμένα, με σκοπό την απλούστευση του συνόλου περιορισμών και την ελαχιστοποίηση του χώρου αναζήτησης λύσεων, δοκιμάζουμε να σταθεροποιήσουμε την τιμή μιας κύριας μεταβλητής και ρωτούμε εκ νέου τον επιλυτή για την ικανοποιησιμότητα του τροποποιημένου συνόλου περιορισμών.

- Σε περίπτωση που ο επιλυτής καταφέρει να βρει λύση (δηλαδή το νέο σύνολο περιορισμών είναι ικανοποιήσιμο), τότε γνωρίζουμε ότι και το αρχικό σύνολο περιορισμών είναι ικανοποιήσιμο από το ίδιο μοντέλο και προχωρούμε τη διαδικασία ελέγχου κανονικά.

- Σε περίπτωση που ο επιλυτής αποφανθεί ότι το τροποποιημένο σύνολο εντολών είναι μη ικανοποιήσιμο, τότε δεν μπορούμε να είμαστε σίγουροι για το αρχικό σύνολο εντολών. Ωστόσο, προκειμένου να συνεχιστεί η διαδικασία ελέγχου, δεχόμαστε ότι και το αρχικό σύνολο είναι μη ικανοποιήσιμο. Επισημαίνουμε ότι εδώ υπάρχει το ενδεχόμενο να θεωρηθεί ως απροσπέλαστο ένα μονοπάτι εκτέλεσης το οποίο στην πραγματικότητα δεν είναι απροσπέλαστο, συνεπώς τυχόν σφάλμα στο ίδιο μονοπάτι δεν θα αποκαλυφθεί ποτέ.
- Σε περίπτωση που ο επιλυτής εξακολουθεί να επιστρέφει αδυναμία επίλυσης, επαναλαμβάνουμε τη διαδικασία με σταθεροποίηση άλλης κύριας παραμέτρου ή και συνδυασμού κυρίων παραμέτρων.

4.12 Συμπεριφορά επιλυτών

Ο τρόπος με τον οποίο λαμβάνονται υπόψη οι περιορισμοί διαφέρει από επιλυτή σε επιλυτή και μεταβάλλεται από έκδοση σε έκδοση, ενώ είναι από εξαιρετικά πολύπλοκος ως παντελώς άγνωστος. Το γεγονός αυτό καθιστά δύσκολη τη συγγραφή των περιορισμών ώστε να αναγνωρίζονται αποδοτικά από τον εκάστοτε επιλυτή, ιδίως όταν πρόκειται για περίπλοκους περιορισμούς που εμπλέκουν αναδρομικές συναρτήσεις. Σε απλούς περιορισμούς δεν παρατηρήσαμε αξιοσημείωτες διαφοροποιήσεις. Επομένως απαιτείται αφιέρωση χρόνου για τη διερεύνηση της βέλτιστης μορφής κάθε αναδρομικής συνάρτησης ανά επιλυτή.

4.12.1 Χρήση συναρτήσεων

Κατά την ανάπτυξη της εφαρμογής καταλήξαμε σε διάφορα γενικά συμπεράσματα ως προς αυτόν τον τομέα. Υπήρξαν περιπτώσεις που είχαμε δύο θεωρητικά ισοδύναμα σύνολα περιορισμών, όπως τα παρακάτω, τα οποία είναι εμφανές ότι αμφότερα ζητούν ο όρος t να είναι μία πλειάδα ακριβώς δύο στοιχείων, με το πρώτο να είναι ακέραιος όρος και το δεύτερο πραγματικός όρος. Το πρώτο αποτελείται από μία σύζευξη απλών περιορισμών τύπου:

```
(assert (
     and
     (is-tuple t)
     (is-tc (tv t))
     (is-int (th (tv t)))
     (is-tc (tt (tv t)))
     (is-real (th (tt (tv t))))
     (is-tn (tt (tt (tv t))))
))
```

Από την άλλη, το δεύτερο, κάνει χρήση συνάρτησης:

```
(define-fun f ((l TList)) Bool (
     (is-tc l)
     (is-int (th l))
     (is-tc (tt l))
     (is-real (th (tt l)))
     (is-tn (tt (tt l)))
))
(assert (
     and
     (is-tuple t)
     (f (tv t))
))
```

Είναι εμφανές ότι το δεύτερο σύνολο είναι φαινομενικά πολυπλοκότερο. Παρά ταύτα, οδηγούσε σε αποτέλεσμα, σε αντίθεση με το πρώτο που οδηγούσε σε αδυναμία επίλυσης. Μία υπόθεση που ερμηνεύει αυτή τη συμπεριφορά είναι ότι αν το σύνολο περιορισμών εφαρμόζεται σε περισσότερους όρους (και όχι μόνο στον t), γράφαμε το σώμα της συνάρτησης f μόνο μία φορά στην είσοδο του επιλυτή και ενδεχομένως έτσι απλοποιούσαμε τον χώρο αναζήτησης λύσεων.

Μία ακόμα απροσδόκητη παρατήρηση είναι ότι ορισμένες φορές ο επιλυτής αποτύγχανε να αποκριθεί ως προς την ικανοποιησιμότητα ενός συνόλου περιορισμών όταν χρησιμοποιούσαμε την εντολή define-fun για τον ορισμό μιας συνάρτησης η οποία δεν περιείχε αναδρομή, ενώ αποκρινόταν με επιτυχία χρησιμοποιώντας την εντολή define-fun-rec, παρόλο που υποτίθεται ότι η ύπαρξη της δεύτερης συνίσταται στον ορισμό των αναδρομικών συναρτήσεων.

4.12.2 Μορφή αναδρομής

Η πιο μυστηριώδης περιοχή συμπεριφοράς των επιλυτών για τη λειτουργία που επιτελούν στο CutEr είναι οι αναδρομικές συναρτήσεις. Η μελέτη καθίσταται δυσκολότερη όταν συνυπάρχουν πολλές συναρτήσεις, εμπλέκονται πολλές φορές και ενδεχομένως η μία καλεί την άλλη. Μας είναι άγνωστη η σειρά και ο τρόπος με τον οποίο θα τις αναπτύξει ο επιλυτής πάνω στην προσπάθειά του να αποφανθεί για την ικανοποιησιμότητα ενός συνόλου περιορισμών. Μπορούμε να φανταστούμε πολλούς διαφορετικούς τρόπους να δομήσουμε ένα περιορισμό με τη χρήση μιας αναδρομικής συνάρτησης, όχι μόνο στο πεδίο της SMTLIB, αλλά γενικότερα στη σχεδίαση λογισμικού. Ως παράδειγμα θα παρουσιάσουμε τη συνάρτηση υπολογισμού της δύναμης του 2.

Χρησιμοποιώντας τον κλασικό αναδρομικό ορισμό (εικόνα 6):

$$2^n = \begin{cases} 1, & n = 0 \\ 2 \cdot 2^{n-1}, & n > 0 \end{cases}$$

```
(define-fun-rec pow2 ((n Int)) Int (
     ite
     (<= n 0)
     1
     (* 2 (pow2 (- n 1)))
))

(declare-const n Int)
(declare-const p Int)

; Z3 when n is fixed and p is free
(assert (= n 745))

; Z3 when n is free and p is fixed
;(assert (= n 515))
(assert (= p
107262463439540776796592199985646769019834926564739147021788491549774112240588375814414994385335227421520254865491888406830031062495572559571469192048672768))

(assert (= p (pow2 n)))

(check-sat)
(get-value (n p))
```

εικόνα 6: Υπολογισμός δύναμης με κλασική αναδρομή

Με αναδρομή ουράς (εικόνα 7):

$$p(n,a) = \begin{cases} 1, & n = 0 \\ p(n-1, 2\cdot a), & n > 0 \end{cases} \text{ και } 2^n = p(n,1)$$

```
(define-fun-rec pow2 ((n Int) (a Int)) Int (
     ite
     (<= n 0)
     a
     (pow2 (- n 1) (* 2 a))
))

(declare-const n Int)
(declare-const p Int)

; Z3 when n is fixed and p is free
(assert (= n 12000))

; Z3 when n is free and p is fixed
;(assert (= n 1700))
(assert (= p
5636280893478578262065541516747052591403096441110831494040
4024234186092896890683553559713795918977451734030100782637
4752985664640194985253964496984843470018650172259313260910
2394906926373262117075563330306831364144677433787178189964
1796135880585055559392506416165288963296180015806240876047
9070707000264636376781037602458046194739037527615885835752
9962546338406854636102805723631292609311399696875316557143
9453911023115321656885066922616340471973247067529508189278
1547238599269322875230618761899538302846073674137 6))

(assert (= p (pow2 n 1)))

(check-sat)
(get-value (n p))
```

εικόνα 7: Υπολογισμός δύναμης με αναδρομή ουράς

Με χρήση κατηγορήματος, όπως σε γλώσσες λογικού προγραμματισμού (εικόνα 8):

$$p = 2^n \Leftrightarrow (n = 0 \Lambda p = 1) \vee \left(n > 0 \Lambda p > 1 \Lambda \bmod(p,2) = 0 \Lambda \frac{p}{2} = 2^{n-1}\right)$$

```
(define-fun-rec pow2 ((n Int) (p Int)) Bool (
     or
     (and (= n 0) (= p 1))
     (
          and
          (> n 0)
          (> p 1)
          (= 0 (mod p 2))
          (pow2 (- n 1) (div p 2))
     )
))

(declare-const n Int)
(declare-const p Int)

; Z3 when n is fixed and p is free
(assert (= n 690))

; Z3 when n is free and p is fixed
;(assert (= n 2600))
(assert (= p
476418619515067823588714793638656202201857139291083867959899955667653856265612022291529050965698956954980403301461703015038614868574193962397485499515664041394986281501470694218497988307090612350614568889981811581986638138356992766991125087684991039263797693265785104917948470363624188015370806861100688624248678603221354110605494242806696770917600281683633644108236805349374738136918815731368582599191891954042154031640578510234055103862187476613117399488481460317825611407737894256555037882572203580919569719825554820635703945122525655142867259335210893272327056051927519150227161849803955583483422130016277366149645187225958439976324236745223185422914992006460893085523949518747052097319358193378697931652326752042142239336421790958496682974373076635520836784028919211864805605376)
)

(assert (pow2 n p))

(check-sat)
(get-value (n p))
```

εικόνα 8: Υπολογισμός δύναμης με χρήση κατηγορήματος

Εκτελέσαμε διάφορα πειράματα, μετατρέποντας σε σχόλια τις γραμμές που αναθέτουν τιμές στις παραμέτρους και αφήνοντας άλλοτε γνωστό τον εκθέτη και άλλοτε τη δύναμη. Θέσαμε κοινό όριο χρόνου 1 δευτερόλεπτο και καταγράψαμε τις μέγιστες τιμές (προσεγγιστικά) των παραμέτρων για τις οποίες ο επιλυτής Z3 έκδοσης 4.5 αποκρίνεται επιτυχώς, δηλαδή επιστρέφει sat και τις τιμές των παραμέτρων στο μοντέλο (πίνακας 1).

Παρατηρούμε πως γενικά αξίζει να διερευνήσουμε σε κάθε είδος περιορισμού τη βέλτιστη μορφή της αναδρομικής συνάρτησης ανά επιλυτή, καθώς όπως φαίνεται παραπάνω η απλούστερη προσέγγιση δεν είναι πάντοτε η καλύτερη δυνατή. Αντιθέτως, μπορούμε να βελτιώσουμε εξαιρετικά την απόδοση χρησιμοποιώντας εναλλακτικές τεχνικές. Τέλος, σημειώνουμε πως οι ίδιες δοκιμές με την έκδοση 1.5-prerelease του επιλυτή CVC4 έβγαλαν τελείως διαφορετικά (και πάντοτε μικρότερα) ανώτατα όρια.

γνωστή παράμετρος	εκθέτης (n)	δύναμη (p)
κλασικός ορισμός	$n = 745$	$p = 2^{515}$
αναδρομή ουράς	$n = 12000$	$p = 2^{1700}$
χρήση κατηγορήματος	$n = 690$	$p = 2^{2600}$

πίνακας 1: Μέγιστες τιμές παραμέτρων κατά τον υπολογισμό δύναμης με τον επιλυτή Z3

5 Υποστήριξη πολλαπλών επιλυτών στο CutEr

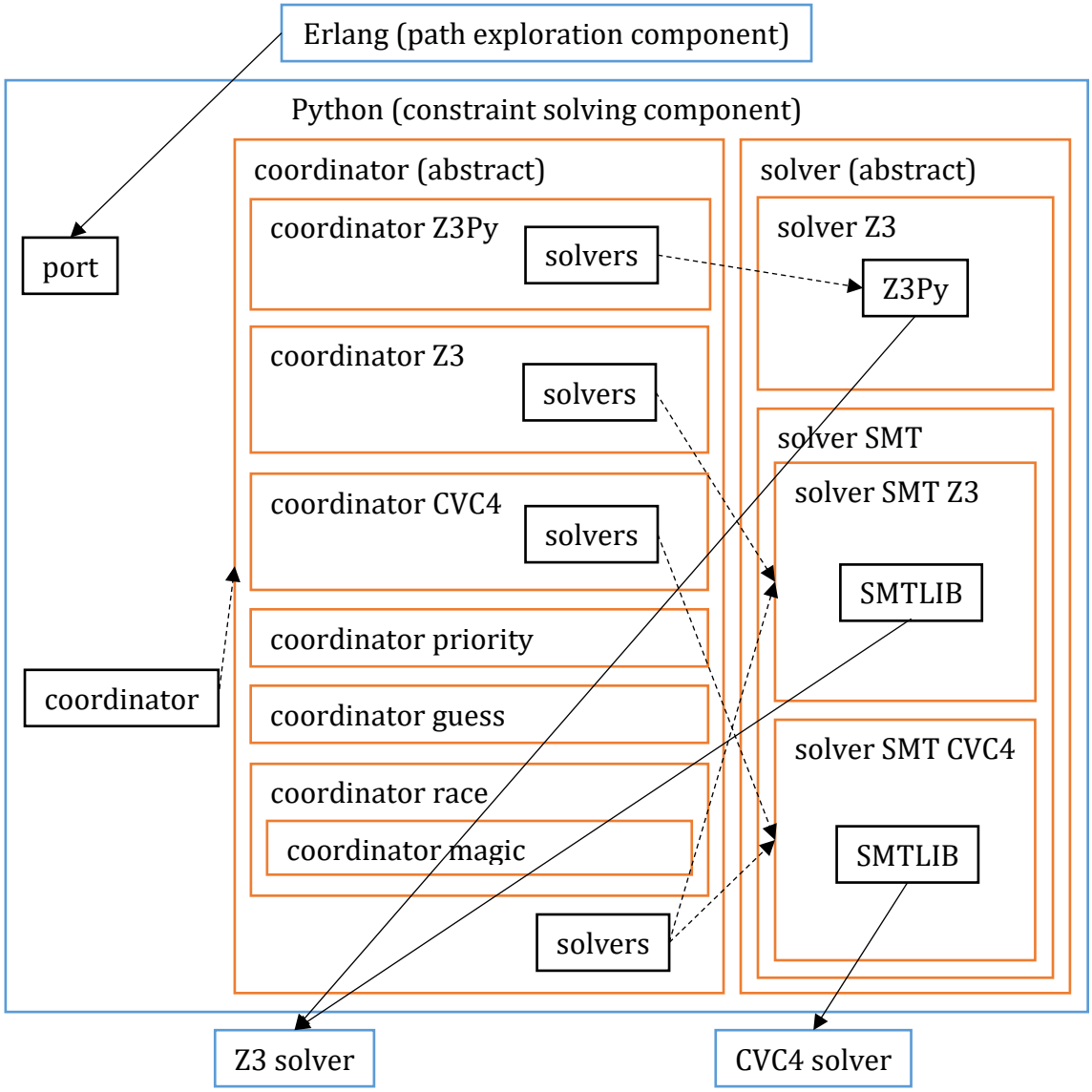
Από τη στιγμή που το CutEr επικοινωνεί με τον επιλυτή Z3 μέσω της γλώσσας SMTLIB, είναι σχετικά απλή διαδικασία η αντικατάσταση του επιλυτή με κάποιον άλλον, όπως ο CVC4. Τουλάχιστον τα απλούστερα παραδείγματα θα ελέγχονται επιτυχώς, εφόσον καλέσουμε το νέο επιλυτή ρυθμίζοντας τις στοιχειώδεις επιλογές τους. Ζητούμενο πλέον είναι η συνύπαρξη πολλών διαφορετικών επιλυτών κατά τρόπο ώστε να μπορούμε εύκολα να ορίσουμε κάποια στρατηγική που θα διαλέγει αν θα χρησιμοποιείται ένας επιλυτής ή περισσότεροι, καλώντας τους σειριακά ή παράλληλα.

5.1 Οργάνωση Python

Η ύπαρξη πολλών επιλυτών και η δυνατότητα επιλογής τους οδηγεί σε μικρές προσθήκες σταδίων στο τμήμα της Python: υλοποιούνται υποκλάσεις για κάθε επιλυτή και μεσολαβεί η στρατηγική επιλογής επιλυτών (εικόνα 9).

Μία πρώτη διαφορά σε σχέση με την περίπτωση υποστήριξης ενός μόνο επιλυτή (εικόνα 3) είναι ότι στην περίπτωση των πολλαπλών επιλυτών δεν επιλέγουμε άμεσα τον επιλυτή που θα εμπλέκεται, αλλά ορίζουμε την στρατηγική με την οποία θα καλούμε τους διάφορους επιλυτές. Δηλαδή διαλέγουμε ένα συντονιστή που θα αποφασίζει ποιοι επιλυτές θα χρησιμοποιούνται κάθε φορά.

Επίσης, πλέον η Erlang δεν επικοινωνεί απ' ευθείας με κάποια κλάση επιλυτή. Αντιθέτως, ανταλλάζει πληροφορίες με ένα συντονιστή. Ο συντονιστής προωθεί τις εντολές της Erlang στους επιλυτές που ο ίδιος επιλέγει, ενώ διαβιβάζει το πρώτο αποτέλεσμα που λαμβάνει στην Erlang.



εικόνα 9: Επικοινωνία με πολλαπλούς επιλυτές

5.1.1 Κλάση συντονιστή

Το πρώτο πράγμα που εξετάζει ένας συντονιστής είναι να ελέγξει ποιοι επιλυτές είναι εγκατεστημένοι στο σύστημα. Θεωρούμε πως ένας επιλυτής (ο Z3 εν προκειμένω) είναι σίγουρα διαθέσιμος. Σε περίπτωση που ένας επιλυτής απουσιάζει, δεν τον λαμβάνει υπόψη του κατά τη διαδικασία της επιλογής επιλυτών. Για κάθε έναν από τους υπόλοιπους επιλυτές, στους οποίους έχει πρόσβαση, δημιουργεί ένα στιγμιότυπο της κλάσης του επιλυτή και το τοποθετεί σε μία λίστα.

Κατά τη λήψη των εντολών από την Erlang, καταρτίζει μία λίστα με τις κύριες παραμέτρους και τις τιμές που είχαν στο αρχείο ίχνους. Τις αντιστοιχίσεις αυτές τις χρησιμοποιεί σε περίπτωση που κανένας επιλυτής δεν μπόρεσε να αποφανθεί για το κατά πόσο το σύνολο περιορισμών είναι ικανοποιήσιμο ή μη ικανοποιήσιμο, οπότε και εξισώνει με τη σειρά τις κύριες παραμέτρους με τις τιμές τους, ώστε να περιορίσει το χώρο αναζήτησης λύσεων.

Μόλις αναγνωσθούν όλοι οι περιορισμοί, ο συντονιστής δίνει εντολή (σύμφωνα με τη στρατηγική επιλογής που υλοποιεί) στις κλάσεις των επιλυτών να απαντήσουν ως προς την ικανοποιησιμότητα του συνόλου περιορισμών. Από τον πρώτο επιλυτή που θα πάρει ως απάντηση ότι το σύνολο περιορισμών είναι ικανοποιήσιμο ή μη ικανοποιήσιμο χαρακτηρίζει και το σύνολο περιορισμών αναλόγως, ενώ στην περίπτωση ικανοποιήσιμου ζητάει επιπλέον και τις τιμές των παραμέτρων στο μοντέλο που βρέθηκε. Αυτό το αποτέλεσμα το αναφέρει και στο CutEr. Αν ένας επιλυτής δεν μπορεί να αποφασίσει ως προς την ικανοποιησιμότητα, αγνοεί το αποτέλεσμά του. Αν όμως όλοι οι επιλυτές δεν μπορούν να αποφανθούν, ξεκινά τη διαδικασία εξίσωσης παραμέτρων με τις τιμές που έχουν στο αρχείο ίχνους.

5.1.2 Υποκλάσεις επιλυτή

Ο συντονιστής μπορεί να καλέσει οποιοδήποτε επιλυτή, ανεξάρτητα από το αν είναι SMT. Κλάσεις που δεν εμπλέκουν SMT επιλυτές (όπως η προϋπάρχουσα κλάση που καλούσε τον Z3 μέσω του Python API της) είναι υποκλάσεις της γενικευμένης κλάσης επιλυτή και υλοποιούν όλες τις μεθόδους της. Από την άλλη πλευρά, η υποκλάση SMT επιλυτή, η οποία κληρονομεί την γενικευμένη κλάση, έχει ήδη υλοποιημένες όλες τις μεθόδους της αρχικής από το στάδιο της υποστήριξης ενός μόνο επιλυτή. Το μόνο που χρειάζεται για την εμπλοκή των διαφόρων επιλυτών είναι να τροποποιήσουν ελαφρά ορισμένες μεθόδους της κλάσης SMT επιλυτή, με σκοπό την βελτιστοποίηση της απόδοσης σύμφωνα με τις ιδιαιτερότητες του επιλυτή, καθώς και η υλοποίηση δύο μεθόδων που επιτρέπουν να ελέγξουμε κατά πόσο είναι εγκατεστημένος ένας επιλυτής στο σύστημα και να δημιουργήσουν μια διεργασία του επιλυτή αυτού με τις κατάλληλες παραμέτρους.

5.2 Παράδειγμα εκτέλεσης

Ένα πολύ χαρακτηριστικό παράδειγμα που επιδεικνύει τη λειτουργία του CutEr με πολλαπλούς επιλυτές είναι το test3.erl, το οποίο έχει ληφθεί από τα αρχεία ελέγχου κατά την ανάπτυξη του CutEr (εικόνα 10).

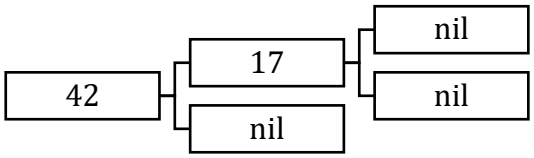
```
-module(test3).
-export([fctree/1]).

-type ctree() :: nil | {integer(), ctree(), ctree()}.

-spec fctree(ctree()) -> ok.
fctree(X) ->
     case X of
          {42, {17, nil, nil}, nil} -> error(bug);
          _ -> ok
     end.
```

εικόνα 10: Παράδειγμα με δυαδικό δέντρο

Δηλώνεται η δομή ctree, η οποία είναι ένα δυαδικό δέντρο με ακέραιες τιμές στους κόμβους και το άτομο nil στα φύλλα. Είναι φανερό πως το πρόγραμμα έχει σχεδιαστεί να εμφανίζει σφάλμα για την είσοδο:



Επιλέγουμε τον συντονιστή προτεραιότητας. Ο συντονιστής αυτός δοκιμάζει όλους τους εγκατεστημένους επιλυτές με τη σειρά, ξεκινώντας από τον επιλυτή Z3 (έκδοση 4.5.0) και, σε περίπτωση αποτυχίας, συνεχίζοντας με τον επιλυτή CVC4 (έκδοση 1.5.0). Ελέγχουμε τον κώδικα δίνοντας σαν αρχική είσοδο το απλούστερο δυαδικό δέντρο:

```
./cuter test3 fctree '[nil]'
```

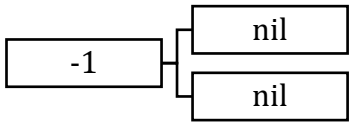
Λόγω της ύπαρξης του εξειδικευμένου τύπου ctree, στον επιλυτή αποστέλλεται ο SMTLIB αναδρομικός ορισμός του:

```
(define-fun-rec ctree ((t Term)) Bool (
     or
     (= t (atom (ic 110 (ic 105 (ic 108 in)))))
     (
          and
          (is-tuple t)
          (is-tc (tv t))
          (is-int (th (tv t)))
          (is-tc (tt (tv t)))
          (ctree (th (tt (tv t))))
          (is-tc (tt (tt (tv t))))
          (ctree (th (tt (tt (tv t)))))
          (is-tn (tt (tt (tt (tv t)))))
     )
))
```

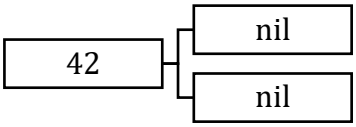
Στην πραγματικότητα, επειδή το CutEr είναι σχεδιασμένο να χειρίζεται πολύ γενικότερες και πολυπλοκότερες περιπτώσεις, ο παραπάνω ορισμός γράφεται σε μία εντολή define-funs-rec με τη βοήθεια δύο συναρτήσεων. Με τέτοιου είδους εκφράσεις φαίνεται πως ο Z3 δυσκολεύεται περισσότερο σε σχέση με τον CVC4. Προσθέτουμε και τους ακόλουθους περιορισμούς (που ζητούν το t να έχει ύψος τουλάχιστον 2) με σκοπό να οδηγηθούμε στο μονοπάτι εκτέλεσης με το σφάλμα:

```
(declare-const t Term)
(assert (ctree t))
(assert (
     and
     (is-tuple t)
     (is-tc (tv t))
     (is-tc (tt (tv t)))
     (is-tc (tt (tt (tv t))))
     (is-tn (tt (tt (tt (tv t)))))
))
```

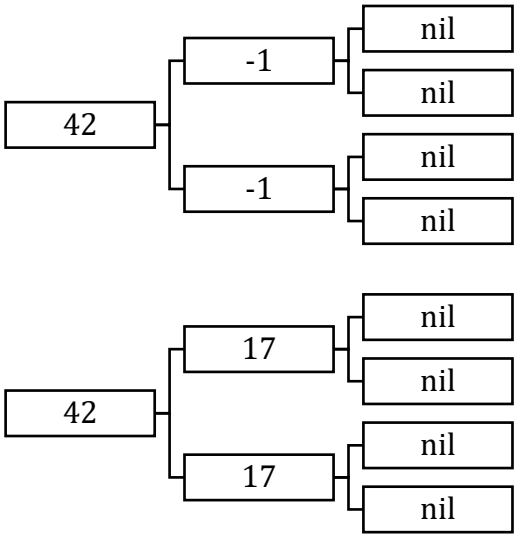
Δοκιμάζοντας, λοιπόν, τον επιλυτή Z3, οδηγούμαστε σε αποτυχία απόφασης λόγω υπέρβασης του χρονικού ορίου. Έπειτα, γράφοντας το ίδιο σύνολο εντολών στην είσοδο του επιλυτή CVC4 παίρνουμε απάντηση ότι είναι ικανοποιήσιμο και την ακόλουθη τιμή της κυρίας παραμέτρου στο μοντέλο:



Το CutEr εκτελεί τον κώδικα με τη νέα τιμή και επαναλαμβάνεται η διαδικασία: Δημιουργείται καινούριο αρχείο ίχνους, αντιστρέφεται ένας περιορισμός, καλείται ο Z3 που αποτυγχάνει, καλείται ο CVC4 που επιτυγχάνει και επιστρέφει την τιμή:

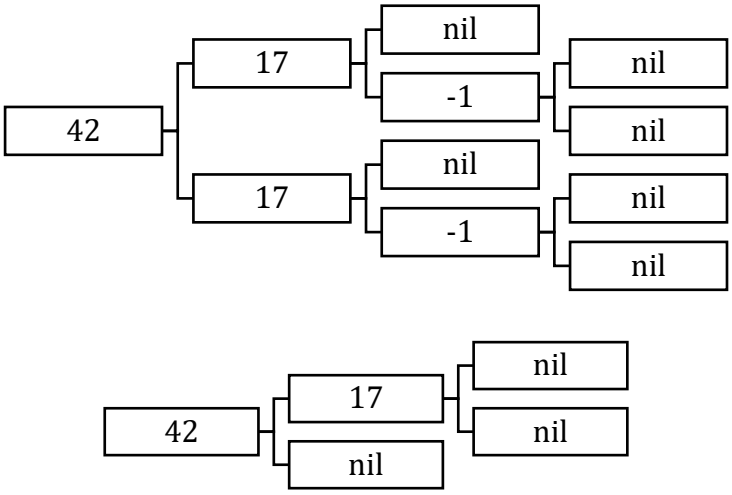


Συνεχίζοντας, με τις ίδιες παρατηρήσεις πάντα:



Σε αυτό το σημείο σχηματίζεται ένα σύνολο περιορισμών για το οποίο αποτυγχάνουν και οι δύο επιλυτές. Συνεπώς καταφεύγουμε στο στάδιο της σταθεροποίησης των

παραμέτρων. Στο συγκεκριμένο πρόβλημα έχουμε μόνο μία παράμετρο, συνεπώς η εξίσωση του όρου με την προηγούμενη τιμή της σε συνδυασμό με το γεγονός ότι στο νέο σύνολο τιμών έχει αντιστραφεί ένας περιορισμός είναι μη ικανοποιήσιμο σύνολο, πράγμα που αναφέρει ο Z3. Όπως έχουμε σημειώσει παραπάνω, αυτό ενδεχομένως οδηγεί σε προσπέραση του σφάλματος. Εν πάση περιπτώσει, συνεχίζουμε όπως προηγουμένως, με τον επιλυτή Z3 να αποτυγχάνει και τον επιλυτή CVC4 να επιστρέφει:



Δίνοντας την τελευταία τιμή σαν είσοδο, το πρόγραμμα εμφανίζει το σφάλμα, επομένως δεν το χάσαμε στο σημείο που και οι δύο επιλυτές απέτυχαν να αποφανθούν ως προς την ικανοποιησιμότητα του συνόλου περιορισμών. Το CutEr πλέον έχει εξερευνήσει όλα τα μονοπάτια και τερματίζει, τυπώνοντας τα ευρήματα:

```
=== Inputs That Lead to Runtime Errors ===
#1 test3: fctree({42,{17,nil,nil},nil})
```

Σε αυτό το σημείο μπορούμε να κάνουμε δύο σύντομες παρατηρήσεις. Πρώτον, είδαμε ότι ο συντονιστής κάθε φορά έπρεπε να περιμένει τον επιλυτή Z3 να τερματίσει (με υπέρβαση του χρονικού ορίου), προκειμένου να καλέσει τον επιλυτή CVC4. Όμως ο Z3 κάθε φορά αποτύγχανε, οπότε θα ήταν προτιμότερος ένας συντονιστής που μαντεύει ότι ο CVC4 θα αποκριθεί με μεγαλύτερη επιτυχία ή ένας που θα τους καλεί ταυτόχρονα. Τέτοιες στρατηγικές υλοποιούνται σε συντονιστές που θα παρουσιαστούν στη συνέχεια. Δεύτερον, το CutEr οδηγήθηκε στο σφάλμα αφού προηγουμένως εξέτασε μονοπάτια που δεν ήταν ανάγκη να εξερευνήσει. Το πρόβλημα αυτό δίνει αφορμή για μία ενδεχόμενη σημαντική βελτίωση του εργαλείου.

5.3 Υποστήριξη εντολών

Στην παρούσα εργασία ασχολούμαστε με επιλυτές που διαθέτουν διεπαφή για τη γλώσσα SMTLIB [1]. Αρκετοί εξ αυτών, ενώ τυπικά επικοινωνούν μέσω SMTLIB, στην πράξη δεν μας είναι χρήσιμοι, καθώς δεν υποστηρίζουν ορισμένες απαραίτητες εντολές που βρίσκονται στις νεότερες εκδόσεις της γλώσσας. Πιο συγκεκριμένα, οι εντολές define-fun-rec και define-funs-rec, οι οποίες επιτρέπουν τη δήλωση αναδρομικών συναρτήσεων, εμφανίζονται για πρώτη φορά στην έκδοση 2.5 (28 Ιουνίου 2015) [6] και,

όπως έχουμε διαπιστώσει, χρησιμεύουν σε πλήθος περιπτώσεων, όπως η επιβολή περιορισμών τύπου λίστας, η αποτίμηση συναρτήσεων, οι bitwise πράξεις μεταξύ ακεραίων. Πολύ περισσότερο όμως, αναγκαίες για τη λειτουργία του CutEr είναι οι εντολές declare-datatype και declare-datatypes, οι οποίες επιτρέπουν τη δήλωση τύπων δεδομένων και, κατά συνέπεια, αφήνουν το περιθώριο σε μία μεταβλητή να μπορεί να πάρει τιμή από δύο και πάνω τύπους δεδομένων. Οι εντολές αυτές εισάγονται στην έκδοση 2.6 της γλώσσας (18 Ιουλίου 2017) [7]. Όπως γίνεται αντιληπτό, με αυτό το δεδομένο περιορίζεται δραστικά η έρευνά μας ως προς την αναζήτηση, σύνδεση και σύγκριση επιλυτών πάνω στην εφαρμογή μας.

Αρκετοί SMT επιλυτές είτε έχουν εγκαταλειφθεί είτε δεν έχουν σαφείς οδηγίες εγκατάστασης ή χρήσης. Ο MathSAT5 (5.3.14) [21] εκτυπώνει σφάλμα ότι δεν αναγνωρίζει τις εντολές declare-datatypes και define-fun-rec. Ο SMT-RAT (2.1.0) [22], ο veriT (stable2016) [23] και ο Yices (2.5.2) [12] δεν υποστηρίζουν την εντολή declare-datatypes. Προς το παρόν, μονάχα οι επιλυτές CVC4 (1.5) [4] και Z3 (4.5.0) [5] πληρούν τα εν λόγω κριτήρια, ωστόσο εύκολα μπορούμε στο μέλλον να συμπεριλάβουμε και οποιονδήποτε άλλο επιλυτή ικανοποιήσει τις απαιτήσεις μας.

5.4 Ρυθμίσεις επιλυτών

Οι επιλυτές σχεδιάζονται ώστε να αντιμετωπίζουν ποικίλα είδη προβλημάτων. Προβλήματα που σχετίζονται με διαφορετικούς τομείς της μαθηματικής λογικής (όπως θεωρία ακεραίων, διανυσμάτων, συναρτήσεων), που αφορούν αποδείξεις θεωρημάτων (όπως υπολογισμός μοντέλου που ικανοποιεί ένα σύνολο περιορισμών ή εύρεση ελαχίστου μη ικανοποιήσιμου υποσυνόλου περιορισμών), που εμπλέκουν αναδρομή ή ποσοδείκτες. Τις περισσότερες φορές τα σύνολα περιορισμών που παράγονται είναι εξειδικευμένα, οπότε δεν είναι αναγκαία η υποστήριξη όλων αυτών των λειτουργιών και, εφόσον είναι ενεργοποιημένες, δυσχεραίνεται το έργο των επιλυτών στις πιο σύνθετες περιπτώσεις. Επιλέγουμε λοιπόν να ελαχιστοποιήσουμε το χώρο αναζήτησης λύσεων ρυθμίζοντας κατάλληλα τους επιλυτές μέσω παραμέτρων της γραμμής εντολών ή μέσω οδηγιών στην είσοδο. Συχνά οι παράμετροι είναι πολλές και οι συνδυασμοί που πρέπει να δοκιμάσουμε για να πετύχουμε το βέλτιστο αποτέλεσμα ακόμη περισσότεροι.

Έχουμε ήδη αναφέρει τις ρυθμίσεις που αφορούν τη γλώσσα επικοινωνίας, το μέγιστο χρόνο εκτέλεσης και την παραγωγή των μοντέλων σε περίπτωση ικανοποιήσιμου συνόλου εντολών. Ο χρόνος εκτέλεσης στον επιλυτή CVC4 καθορίζεται σε χιλιοστά του δευτερολέπτου, ενώ στον επιλυτή Z3 σε ακέραια δευτερόλεπτα. Ο CVC4 χρειάζεται επιπλέον και τη σημαία --fmf-fun για την παραγωγή μοντέλων, εφόσον ορίζονται αναδρομικές συναρτήσεις. Η λίστα των παραμέτρων γραμμής εντολών για τους επιλυτές Z3 και CVC4 που ανταποκρίνονται στις ανάγκες του CutEr είναι αντίστοιχα:

```
["z3", "-smt2", "-T:{}".format(timeout), "-in"]
```

```
["cvc4", "--lang=smt2.5", "--tlimit={}".format(timeout *
1000), "--fmf-fun"]
```

Εναλλακτικά, θα μπορούσαμε να παραλείψουμε τη σημαία --fmf-fun από τη λίστα ορισμάτων γραμμής εντολών και να προσθέσουμε στην είσοδο του επιλυτή την εντολή (set-option :fmf-fun true).

Ο επιλυτής Z3 δεν έχει αναλυτικές οδηγίες για τον περιορισμό της εμπλεκόμενης λογικής και φαίνεται πως δεν αναγνωρίζει όλα τα προβλεπόμενα σύνολα λογικής της SMTLIB [2], οπότε χρησιμοποιούμε την προεπιλογή, η οποία υποστηρίζει όλες τις λογικές. Αντιθέτως, στον επιλυτή CVC4 υπάρχει η δυνατότητα λεπτομερούς επιλογής του συνόλου λογικής. Τα παρακάτω αρχικά με τη σειρά παρέχουν υποστήριξη στα εξής πεδία:

- QF (quantifier free): στους περιορισμούς δεν επιτρέπεται χρήση ποσοδεικτών, κατά συνέπεια ούτε αναδρομικών συναρτήσεων που ερμηνεύονται με τη βοήθειά τους
- A (arrays): λογική διανυσμάτων
- UF (uninterpreted functions): σε ορισμένες συναρτήσεις δεν γνωρίζουμε εκ των προτέρων το σώμα, δηλαδή την πλήρη αντιστοίχιση παραμέτρων εισόδου με τιμές εξόδου
- DT (datatypes): μία παράμετρος μπορεί να λάβει τιμές από διάφορους, ενδεχομένως εξειδικευμένους, τύπους δεδομένων
- L (linear): αποκλειστικά γραμμική λογική
- N (non-linear): υπάρχουν μη γραμμικοί περιορισμοί
- IA (integer arithmetic): αριθμητική ακεραίων
- RA (real arithmetic): αριθμητική πραγματικών
- IRA (integer and real arithmetic): αριθμητική ακεραίων και πραγματικών

Κάποιες από τις λογικές είναι εξ ορισμού αμοιβαία αποκλειόμενες, δηλαδή δεν μπορούν να περιέχονται ταυτόχρονα στο επιλεγμένο σύνολο λογικής. Εμείς, στην περίπτωση του επιλυτή CVC4, αρκούμαστε στη λογική που ορίζεται με την εντολή (set-logic UFDTLIRA), καθώς δεν υποστηρίζεται η παραγωγή μοντέλων παρουσία μη γραμμικών συνόλων περιορισμού.

5.5 Διαφοροποιήσεις υλοποιήσεων

Εκτός από τον ορισμό εξειδικευμένων ρυθμίσεων κατά την κλήση των επιλυτών, σημειώνονται και διαφορές κατά τη χρήση τους, ειδικά σε περιπτώσεις περίπλοκων δηλώσεων αναδρομικών συναρτήσεων. Ο επιλυτής CVC4, μετά τις τελευταίες ενημερώσεις λογισμικού, είναι σε θέση να υπολογίσει όρους τύπου function, όπως τους έχουμε ορίσει στους τύπους δεδομένων, χωρίς κάποιο βοηθητικό πρόσθετο περιορισμό στην επιβολή της συνθήκης της τιμής μιας συνάρτησης. Αντιθέτως, στον επιλυτή Z3 χρειάστηκε να προσθέσουμε μία παράμετρο βάθους αναζήτησης που αυξάνεται με το πλήθος των περιορισμών τιμής συνάρτησης (με την ύπαρξη της οποίας παραδόξως ο CVC4 αδυνατούσε να υπολογίσει μοντέλα).

Πιο συγκεκριμένα, είδαμε πως μία συνάρτηση κωδικοποιείται σαν λίστα ζευγών λίστας όρων με όρο. Για παράδειγμα, αν για μία συνάρτηση $f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ ισχύουν $f(2,0) = 0.5$ (συνθήκη πρώτη) και $f(-1,1) = -3.8$ (συνθήκη δεύτερη), πέραν του περιορισμού

```
(assert (= (fa (fv f)) 3))
```

θα απαιτήσουμε με τη σειρά

```
(assert (
     flist-equals
     (fm (fv f))
     (tc (int 2) (tc (int 0) tn))
     (real 0.5)
))
```

και

```
(assert (
     flist-equals
     (fm (fv f)) (tc (int (- 1))
     (tc (int 1) tn))
     (real (- 3.8))
))
```

ώστε κάπου στη λίστα ζευγών να εμφανιστούν οι δύο παραπάνω συνθήκες. Παρά ταύτα, στον επιλυτή Z3 τροποποιούμε τους περιορισμούς ώστε η πρώτη συνθήκη να εξασφαλίζεται από το πρώτο ζεύγος και η δεύτερη συνθήκη από το πρώτο ή το δεύτερο ζεύγος της λίστας. Το τρέχον βάθος αναζήτησης αποθηκεύεται σε μία μεταβλητή (property) της κλάσης του SMT επιλυτή.

5.6 Σύγκριση αποκρισιμότητας

Οι δύο ανταγωνιζόμενοι επιλυτές που υποστηρίζουν με επιτυχία το CutEr καλύπτουν επιτυχώς τις απλές περιπτώσεις συνόλων περιορισμών. Τα σημεία που εντοπίζονται οι περισσότερες διαφορές είναι οι περιορισμοί που επιβάλλονται με τη βοήθεια SMTLIB αναδρομικών συναρτήσεων.

Όπως είδαμε προηγουμένως, ο Z3 αντιμετωπίζει τους περιορισμούς τιμής τύπου συνάρτησης μόνο εφόσον καθορίσουμε ένα βάθος αναζήτησης στη δομή, οπότε και σημειώνει θετικά αποτελέσματα σε όλα τα παραδείγματα ελέγχου. Αντιθέτως, ο CVC4 επιτυγχάνει όταν ψάχνει χωρίς συγκεκριμένο βάθος αναζήτησης, αλλά σε περιπτώσεις που αναμιγνύονται περιορισμοί αναδρομικών τύπων (όπως λίστες) αδυνατεί να αποφανθεί ως προς την ικανοποιησιμότητα.

Σε μη γραμμικά σύνολα περιορισμών, όσο εύκολα ή περίπλοκα κι αν είναι, ο CVC4 αρνείται να δημιουργήσει μοντέλα. Αντιθέτως, ο Z3 καταβάλλει προσπάθεια να τα επιλύσει και σε ορισμένες περιπτώσεις επιστρέφει κάποιο μοντέλο. Συνεπώς, το CutEr

κατορθώνει με τη βοήθεια του Z3 να εντοπίσει κάποιο σφάλμα σε κώδικα, ακόμα κι αν περιέχονται συνθήκες όπως ο υπολογισμός ρίζας αριθμού.

Παρομοίως, σε ορισμένους περιορισμούς τύπου bitstring ο CVC4 αδυνατεί να αποφανθεί. Το ίδιο ισχύει και σε bitwise αριθμητικές πράξεις μεταξύ ακεραίων και σε άλλα πολύπλοκα σύνολα περιορισμών.

Ένας τομέας στον οποίο υπερέχει αισθητά ο CVC4 του Z3 είναι οι περιορισμοί εξειδικευμένων αναδρομικών τύπων οριζόμενων από το χρήστη. Ο Z3 αντιμετωπίζει επιτυχώς απλές αναδρομικές δομές όπως μία συνδεδεμένη λίστα, ωστόσο αποτυγχάνει σε πολυπλοκότερες δομές όπως τα δυαδικά δένδρα. Από την άλλη, ο CVC4 αποφαίνεται ως προς την ικανοποιησιμότητα του συνόλου περιορισμών σε αμφότερες τις περιπτώσεις.

Το CutEr οδηγείται σε ένα μονοπάτι εκτέλεσης με βάση το μοντέλο που επιστρέφει ο επιλυτής. Τις περισσότερες φορές το μοντέλο αυτό δεν είναι μοναδικό και ο επιλυτής επιστρέφει ένα με βάση κάποια στρατηγική που υλοποιεί εσωτερικά. Υπάρχει, λοιπόν, η περίπτωση να προκύψουν διαφορετικά σύνολα περιορισμών για τον ίδιο εξεταζόμενο κώδικα. Συμβαίνει μάλιστα σε συγκεκριμένο παράδειγμα ο CVC4 να οδηγηθεί σε σύνολο περιορισμών που δεν μπορεί να επιλύσει κανείς από τους δύο επιλυτές, απλά με τον επιλυτή Z3 λόγω διαφορετικών προηγούμενων μοντέλων δεν εμφανίζεται.

Παρενθετικά αναφέρουμε ότι πριν την αξιοποίηση των SMTLIB αναδρομικών συναρτήσεων το CutEr δεν υποστήριζε εξειδικευμένους αναδρομικούς τύπους οριζόμενους από το χρήστη. Επομένως, ο Z3 που εμπλέκεται μέσω του Python API αντιμετωπίζει όλες τις υπόλοιπες περιπτώσεις, παρόλο που υπάρχει το ενδεχόμενο να επιστρέφει ασυνεπή μοντέλα ως προς τους περιορισμούς τύπου. Επιπλέον, απουσία των αναδρομικών συναρτήσεων, οι bitwise πράξεις ακεραίων επιτρέπουν αριθμούς μέχρι μία καθορισμένη τάξη μεγέθους.

5.7 Μείξη επιλυτών

Έχοντας στη διάθεσή μας τουλάχιστον δύο επιλυτές, καθώς και μία εικόνα του βαθμού στον οποίο καθένας τους αντιμετωπίζει τα διάφορα σύνολα περιορισμών, μπορούμε να σχεδιάσουμε πλήθος στρατηγικών επιλογής των επιλυτών. Παρακάτω παρουσιάζουμε τις στρατηγικές που υλοποιήσαμε σαν υποκλάσεις του βασικού συντονιστή.

5.7.1 Συντονιστής απλός

Στην κλάση Solver_Coordinator_Z3 αγνοούμε την ύπαρξη των υπολοίπων επιλυτών και αξιοποιούμε μόνο τον SMT επιλυτή Z3. Ομοίως, στις κλάσεις Solver_Coordinator_CVC4 και Solver_Coordinator_Z3Py λαμβάνουμε υπόψη μόνο τον SMT επιλυτή CVC4 και τον Z3Py αντίστοιχα. Το CutEr συμπεριφέρεται όπως ακριβώς και στην περίπτωση που υπάρχει μόνο ένας επιλυτής.

Ο συντονιστής αυτός εξυπηρετεί στην ανάπτυξη της κλάσης του εκάστοτε επιλυτή ώστε να αποδίδει καλύτερα απουσία των υπολοίπων. Σκοπός όμως της εργασίας είναι η μείξη

πολλών επιλυτών, καθώς καθένας τους εμφανίζει πλεονεκτήματα και μειονεκτήματα, συνεπώς έχει νόημα να παρατηρήσουμε με έμφαση τους επόμενους συντονιστές.

5.7.2 Συντονιστής προτεραιότητας

Στην κλάση Solver_Coordinator_Priority αξιοποιούμε όλους τους SMT επιλυτές που υπάρχουν διαθέσιμοι στο σύστημα, δημιουργώντας ένα αντικείμενο της κλάσης του επιλυτή για καθέναν και τοποθετώντας τα αντικείμενα με συγκεκριμένη σειρά σε μία λίστα. Μόλις ο συντονιστής παραλάβει ένα σύνολο περιορισμών, καλεί τον πρώτο επιλυτή της λίστας. Εάν αυτός αποφανθεί επιτυχώς ως προς την επιλυσιμότητα ή μη του εν λόγω συνόλου περιορισμών, ολοκληρώνεται το έργο του. Διαφορετικά καλεί τον επόμενο επιλυτή της λίστας με τις ίδιες συνθήκες. Αν όλοι οι επιλυτές αποτύχουν να αποφανθούν, σταθεροποιείται η τιμή μιας μεταβλητής με βάση το αρχείο ίχνους και επαναλαμβάνεται η διαδικασία με την εν σειρά κλήση των επιλυτών.

Σημειώνουμε ότι, εφόσον ο συντονιστής αυτός αξιοποιεί όλους τους υπάρχοντες επιλυτές, είναι κατά κάποιο τρόπο ιδανικός ως προς την αντιμετώπιση των διαφορετικών συνόλων περιορισμού. Δηλαδή, αν ένα σύνολο περιορισμού επιλύεται από τουλάχιστον έναν από τους διαθέσιμους επιλυτές, θα αντιμετωπίζεται επιτυχώς και από τον συντονιστή προτεραιότητας. Το ίδιο ισχύει και για όλους τους ακολουθούντες επιλυτές. Βεβαίως, μπορούμε να βελτιστοποιήσουμε την απόδοση ως προς το συνολικό χρόνο εκτέλεσης και το πλήθος των διεργασιών που δημιουργούνται.

5.7.3 Συντονιστής υπόθεσης

Στην κλάση Solver_Coordinator_Guess προσθέτουμε ένα στοιχείο ευφυΐας. Η μόνη διαφοροποίησή του από το συντονιστή προτεραιότητας είναι ότι κατά την παραλαβή του συνόλου περιορισμών, τους ελέγχει ως προς το είδος τους και κρατάει μία συνοπτική πληροφορία για το σύνολό τους. Γνωρίζοντας λοιπόν ότι ένας επιλυτής έχει μεγαλύτερες πιθανότητες επιτυχίας στην αντιμετώπιση ενός είδους περιορισμού, σε σύγκριση με κάποιον άλλον, αναδιατάσσει τη λίστα των αντικειμένων της κλάσης των επιλυτών, φέρνοντας πρώτο τον επιλυτή που υποθέτει ότι θα αποδώσει καλύτερα.

Στην πραγματικότητα αυτός είναι και ο λόγος για τον οποίο στο συντονιστή προτεραιότητας τοποθετούμε πάντοτε πρώτο τον επιλυτή Z3: είναι ο επιλυτής που αποκρίνεται επιτυχώς στις περισσότερες περιπτώσεις. Ωστόσο, παρατηρήσαμε πως όταν υπάρχουν αναδρομικοί τύποι δεδομένων ορισμένοι από το χρήστη, ο επιλυτής CVC4 εμφανίζει ένα προβάδισμα. Κατά συνέπεια, στην υλοποίησή μας ο συντονιστής υπόθεσης διορθώνει τον συντονιστή προτεραιότητας ως εξής: Εξετάζει αν στο σύνολο περιορισμών υπάρχει ο ορισμός κάποιου εξειδικευμένου αναδρομικού τύπου δεδομένων. Εάν όντως υπάρχει, μαντεύει ότι ο επιλυτής CVC4 είναι πιθανότερο να χειριστεί επιτυχώς το σύνολο περιορισμών και τον τοποθετεί πρώτο στη λίστα.

Πράγματι, αν στο παράδειγμα εκτέλεσης test3.erl χρησιμοποιούσαμε τον συντονιστή υπόθεσης, θα δημιουργούσαμε πρακτικά τις μισές διεργασίες επιλυτών, αφού ο CVC4 θα αποκρινόταν επιτυχώς και θα αποφεύγαμε τη δημιουργία της διεργασίας του επιλυτή Z3 και την αναμονή του να τερματίσει λόγω παρέλευσης του μεγίστου χρονικού ορίου

εκτέλεσης. Συμπερασματικά, θα εξοικονομούσαμε πόρους (λιγότερες διεργασίες) και χρόνο (αφού ο CVC4 αποκρίνεται άμεσα, ενώ ο Z3 εξαντλεί τα χρονικά όρια κι ύστερα δηλώνει την αποτυχία του).

5.7.4 Συντονιστής ανταγωνισμού

Όπως αναφέραμε χαρακτηριστικά, ο συντονιστής υπόθεσης μαντεύει ότι ένας επιλυτής θα είναι αποτελεσματικότερος στην αντιμετώπιση ενός συνόλου εντολών, με βάση την εμπειρία που συλλέξαμε κατά την σύγκριση των διαφόρων επιλυτών. Υπάρχουν όμως περιπτώσεις για τις οποίες δεν μπορούμε να υποθέσουμε με κάποιο βαθμό βεβαιότητας ποιος είναι ο βέλτιστος επιλυτής, όπως για παράδειγμα σύνολα εντολών που περιέχουν ταυτόχρονα εντολές στις οποίες αποδίδει καλύτερα ένας επιλυτής και εντολές στις οποίες αποδίδει καλύτερα άλλος επιλυτής. Ή υπάρχουν σύνολα εντολών τα οποία δεν τα έχουμε εξετάσει καθόλου.

Στην κλάση Solver_Coordinator_Race υλοποιούμε μία εντελώς διαφορετική προσέγγιση. Εκκινούμε ταυτόχρονα όλους τους επιλυτές σε διαφορετικές διεργασίες. Ο συντονιστής στέλνει σήματα σε κάθε διεργασία μέσω ξεχωριστής σωλήνωσης (pipe) και λαμβάνει πληροφορίες από τους επιλυτές μέσω κοινής ουράς (queue). Τα σήματα που στέλνει είναι η εντολή επίλυσης και η εντολή σταθεροποίησης της τιμής μιας μεταβλητής. Οι πληροφορίες που λαμβάνει είναι το αποτέλεσμα της επίλυσης, συνοδευόμενο από ένα μοντέλο που ικανοποιεί τους περιορισμούς, εφόσον το σύνολό τους βρέθηκε ικανοποιήσιμο. Το έργο του συντονιστή ολοκληρώνεται μόλις μία διεργασία στείλει ότι το σύνολο περιορισμών είναι ικανοποιήσιμο ή μη ικανοποιήσιμο και στην πρώτη περίπτωση επιστρέφει και το παρεχόμενο μοντέλο στο CutEr. Εάν όλες οι διεργασίες δεν καταφέρουν να αποφανθούν ως προς την ικανοποιησιμότητα του συνόλου περιορισμών, στέλνεται το σήμα σταθεροποίησης της τιμής μιας μεταβλητής σε όλες τις διεργασίες επιλυτών και επαναλαμβάνεται η διαδικασία.

Παρατηρούμε ότι κατ' αυτόν τον τρόπο λαμβάνουμε το αποτέλεσμα από την διεργασία που αποκρίθηκε γρηγορότερα και αγνοούμε όλες τις υπόλοιπες. Εκ πρώτης όψεως η συγκεκριμένη στρατηγική δίνει με το γρηγορότερο δυνατό τρόπο το αποτέλεσμα. Ωστόσο υπενθυμίζουμε ότι προστίθεται ο χρόνος της δημιουργίας μιας διεργασίας και της αλληλεπίδρασης με αυτήν, ενώ λόγω του πλήθους των νέων διεργασιών είναι πιθανό εξαιτίας της χρονοδρομολόγησης του πυρήνα η διεργασία που θα έδινε γρηγορότερα το αποτέλεσμα να γίνει ενεργή σε μεταγενέστερο στάδιο. Ήδη το CutEr εξερευνά πολλά μονοπάτια εκτέλεσης σε διαφορετικά νήματα. Σαν ένα ακόμα μειονέκτημα του συντονιστή αναφέρουμε τις αυξημένες απαιτήσεις μνήμης, λόγω της δημιουργίας ξεχωριστής διεργασίας και της αντιγραφής των αντικειμένων των κλάσεων των επιλυτών.

5.7.5 Συντονιστής επιλογής

Μία επιπλέον βελτίωση που μπορούμε να υλοποιήσουμε στους συντονιστές είναι η απόρριψη των επιλυτών για τους οποίους είμαστε βέβαιοι ότι δεν αποκρίνονται επιτυχώς σε ορισμένες εντολές. Για παράδειγμα, ο επιλυτής CVC4 αδυνατεί εξ ολοκλήρου

να χειριστεί περιορισμούς bitwise αριθμητικών πράξεων μεταξύ ακεραίων μέσω των αναδρομικών συναρτήσεων που έχουμε γράψει. Κατά την ανάγνωση, λοιπόν, των περιορισμών και πριν την προώθησή τους στους επιλυτές, ελέγχουμε στην κλάση Cuter_Coordinator_Magic αν ο περιορισμός αφορά τέτοιου είδους αριθμητική πράξη, οπότε και αφαιρούμε τον επιλυτή CVC4 από τη λίστα. Μάλιστα, ο εν λόγω συντονιστής επεκτείνει τον συντονιστή ανταγωνισμού, με αποτέλεσμα να προκύπτει ένας συντονιστής ανταγωνισμού που δημιουργεί μόνο τις διεργασίες που έχουν κάποια πιθανότητα να αποφανθούν, εξοικονομώντας πόρους του συστήματος.

6 Συμπεράσματα

Το CutEr ήταν εξ αρχής ένα εργαλείο που εκτελεί συμβολικό έλεγχο, διατηρώντας παράλληλα και αποτίμηση των μεταβλητών εισόδου. Με την ολοκλήρωση της παρούσας εργασίας, το καταστήσαμε ικανό να εκμεταλλεύεται τα προτερήματα πολλών επιλυτών για την επίλυση των διαφόρων συνόλων περιορισμού που προκύπτουν. Η βελτίωση της εφαρμογής δεν είναι μόνο θεωρητική, καθιστώντας την πρωτοποριακή ως προς τον τρόπο με τον οποίο λειτουργεί ως πρόγραμμα ελέγχου, αλλά τη διαπιστώνουμε στην πράξη, όπως εξηγούμε παρακάτω.

6.1 Κύρια αποτελέσματα

Σε πρώτο στάδιο το εργαλείο CutEr τροποποιήθηκε με την υποστήριξη SMT επιλυτών. Η επικοινωνία με τον αυθαίρετο επιλυτή πραγματοποιείται με την κοινώς αποδεκτή γλώσσα SMTLIB. Μεταγενέστερα προστέθηκαν διάφορες στρατηγικές επιλογής επιλυτή, ανάλογα με το εκάστοτε σύνολο περιορισμών μέσω ενός συντονιστή. Οι αλλαγές αυτές προσέδωσαν στο εργαλείο τη ικανότητα εύρεσης σφαλμάτων σε πολύ μεγαλύτερο εύρος εξεταζόμενων προγραμμάτων. Η πρόταση αυτή στηρίζεται στη μεταβολή του συνόλου προγραμμάτων ελέγχου λειτουργικότητας του εργαλείου κατά τη διάρκεια της εισαγωγής των παραπάνω χαρακτηριστικών. Το σύνολο αυτό επεκτάθηκε σημαντικά χάρη στις νέες δυνατότητες και εν τέλει καλύφθηκε σε απόλυτο βαθμό.

Επιγραμματικά αναφέρουμε ορισμένες βασικές βελτιώσεις. Τα επιστρεφόμενα μοντέλα είναι συνεπή ως προς το σύνολο περιορισμών. Είναι θεωρητικά εφικτές οι bitwise αριθμητικές πράξεις μεταξύ ακεραίων οποιασδήποτε τάξης μεγέθους. Ελέγχονται με επιτυχία εξειδικευμένοι αναδρομικοί τύποι δεδομένων οριζόμενοι από το χρήστη. Οι περισσότερες εξ αυτών των βελτιώσεων οφείλονται στην εκμετάλλευση των αναδρομικών συναρτήσεων της γλώσσας SMTLIB.

Αδιαμφισβήτητα ο SMT επιλυτής Z3 είναι ικανός να διαχειριστεί τη συντριπτική πλειοψηφία των περιπτώσεων συνόλων περιορισμών. Ωστόσο, αδυνατούσε να αντιμετωπίσει ορισμένες φορές τους εξειδικευμένους αναδρομικούς τύπους, εξαιρώντας ένα εξέχουσας σημασίας κομμάτι των δομών δεδομένων. Η συμπλήρωση του εργαλείου με την προσθήκη του επιλυτή CVC4 έλυσε αυτό το πρόβλημα και πλέον, χάρη στη συνύπαρξη και αξιοποίηση των δύο επιλυτών, το εργαλείο περνά επιτυχώς όλα τα προγράμματα ελέγχου λειτουργικότητας.

Δυστυχώς μόνο οι επιλυτές CVC4 και Z3 μπόρεσαν να ικανοποιήσουν τις θεμελιώδεις ανάγκες του CutEr. Παρά ταύτα, το εργαλείο είναι έτοιμο να χρησιμοποιήσει αυθαίρετο αριθμό SMT επιλυτών. Ευελπιστούμε ότι στο μέλλον θα γίνουν κι άλλοι επιλυτές ανταγωνιστικοί, επιτρέποντας την επίλυση μεγαλύτερης ποικιλίας συνόλων περιορισμού και με υψηλότερη ταχύτητα.

6.2 Παρεμφερείς βελτιώσεις

Κατά την υλοποίηση της παρούσας εργασίας αποκαλύφθηκαν διάφορες ελλείψεις και ελαττώματα του CutEr. Επιπλέον, αρκετά στάδια της ανάπτυξης στάθηκαν αφορμή προκειμένου να βελτιωθεί ο κύριος κώδικας του εργαλείου, που δεν σχετίζεται άμεσα με την επικοινωνία με τον επιλυτή.

Βρέθηκαν ρουτίνες που δεν είχαν γραφεί και υλοποιήθηκαν στα πλαίσια της εργασίας. Δεν υπήρχε στο σύνολο προγραμμάτων ελέγχου κώδικας που να οδηγεί στην κλήση της μεθόδου unfold_tuple της κλάσης του επιλυτή. Παρομοίως, η συνάρτηση is_unsat έλειπε από το αρχείο cuter_common.py, καθώς δεν γινόταν χρήση της.

Από τους επανειλημμένους ελέγχους του προγράμματος καταλήξαμε σε ελαφρώς διαφορετική μορφή των διαγνωστικών μηνυμάτων που τυπώνονται στην έξοδο standard error, όταν το CutEr έχει δομηθεί με ενεργή τη σημαία αναλυτικής εκτύπωσης της διαδικασίας επίλυσης (verbose solving). Πιο συγκεκριμένα, αντί να τυπώνεται η δεκαεξαδική αναπαράσταση της εντολής, τυπώνεται ένα φιλικό στο χρήστη μήνυμα με την εντολή και τις παραμέτρους της.

Ένα δύσκολο σημείο που αντιμετωπίσαμε ήταν η απρόσμενη διαφορά στη συμπεριφορά της Erlang κατά την κατασκευή και το ταίριασμα bitstring [24] [25]. Συγκεκριμένα, κατά την κατασκευή ενός bitstring, αποκόπτονται τα bits υπερχείλισης, ενώ κατά τη σύγκριση συνυπολογίζονται ο κωδικοποιούμενος αριθμός και το μήκος του bitstring. Για παράδειγμα, θεωρούμε τον ακέραιο αριθμό 42, του οποίου η δυαδική αναπαράσταση είναι "101010". Αν επιχειρήσουμε να κατασκευάσουμε το <<42:5>>, κωδικοποιώντας τον ακέραιο σε 5 bits, θα αποκοπεί το πιο σημαντικό ψηφίο που υπερχειλίζει, δίνοντας αποτέλεσμα το bitstring <<10:5>>, δηλαδή το "01010". Αντιθέτως, το ταίριασμα προτύπου <<42:5>> = <<10:5>> αποτυγχάνει. Τη συγκεκριμένη ιδιοτροπία δεν τη διαχειριζόμασταν ξεκάθαρα πριν την προσαρμογή του CutEr ώστε να επικοινωνεί μέσω SMTLIB.

Αρχικά το CutEr χρησιμοποιούσε τον επιλυτή Z3 μέσω του Python API. Η κωδικοποίηση των ίδιων συνόλων περιορισμών και το πέρασμά τους στον ίδιο επιλυτή αλλά μέσα από το SMT API οδήγησε σε διαφορετικά μοντέλα. Το ίδιο συνέβαινε και μεταξύ ανανεωμένων εκδόσεων του ίδιου επιλυτή. Συνεπώς προκύπτουν διαφορετικές είσοδοι που οδηγούν στην αποκάλυψη των σφαλμάτων των προγραμμάτων ελέγχου λειτουργικότητας. Για την επαλήθευση ότι το CutEr εντοπίζει πράγματι μία είσοδο που οδηγεί σε σφάλμα, την περιγράφουμε όχι σαν μία σταθερή ντετερμινιστική τιμή, αλλά σαν ένα σύνολο συνθηκών. Για παράδειγμα, θεωρούμε την ακόλουθη συνάρτηση Erlang:

```
-spec trunc1(number()) -> ok.
trunc1(X) ->
     case trunc(X) of
          2 ->
               case X - 2 > 0.5 of
                    true -> error(bug);
                    false -> ok
               end;
          _ -> ok
     end.
```

Ο Z3 εντόπιζε κάθε φορά το σφάλμα για $x = 2.75$, οπότε θεωρούσαμε ότι το CutEr δούλευε σωστά μόνο όταν επέστρεφε τη συγκεκριμένη τιμή:

```
{
     "module": "collection",
     "function": "trunc1",
     "args": "[1]",
     "depth": "25",
     "errors": true,
     "solutions": [
          "[2.75]"
     ],
     "skip": false
}
```

Στην πραγματικότητα, όμως, το ίδιο σφάλμα εντοπίζεται για οποιαδήποτε είσοδο με $\lfloor x \rfloor = 2 \wedge x - 2 > 0.5 \Leftrightarrow 2 \leq x < 3 \wedge x > 2.5 \Leftrightarrow 2.5 < x < 3$, οπότε η ορθή γραφή είναι:

```
{
     "module": "collection",
     "function": "trunc1",
     "args": "[1]",
     "depth": "25",
     "errors": true,
     "arity": 1,
     "nondeterministic": true,
     "solutions": [
          "2.5 < $1 and $1 < 3"
     ],
     "skip": false
}
```

Η προσθήκη του επιλυτή CVC4 έφερε στην επιφάνεια επιπλέον ελλείψεις του εργαλείου. Υπάρχουν μεγέθη που εξυπακούεται πως λαμβάνουν τιμές σε συγκεκριμένο εύρος, χωρίς

να αποτελούν κύριες παραμέτρους του προβλήματος. Αναφέρουμε δύο παραδείγματα. Πρώτον, το μήκος ενός bitstring, δηλαδή το πλήθος των bits από τα οποία απαρτίζεται, είναι μη αρνητικός αριθμός. Δεύτερον, οι χαρακτήρες ενός έγκυρου atom της Erlang ακολουθούν συγκεκριμένη μορφή κωδικοποίησης, σύμφωνα με λεπτομέρειες που αναφέρονται στο παράρτημα. Ο επιλυτής Z3, όταν επρόκειτο να επιστρέψει αυθαίρετη ακέραια τιμή, πολύ σπάνια επέλεγε κάποιον αρνητικό ακέραιο. Όταν όμως ο CVC4 επέστρεψε αρνητικές τιμές, εκδηλώθηκε το σφάλμα, το οποίο και εν συνεχεία διορθώσαμε με την προσθήκη αντίστοιχων περιορισμών.

Όσο γίνονταν οι προσπάθειες ενσωμάτωσης του επιλυτή CVC4 στο CutEr, χρησιμοποιούσαμε μία ασταθή έκδοση του CVC4 (1.5 prerelease), η οποία συνεχώς ανανεωνόταν. Η προηγούμενη ευσταθής έκδοση του επιλυτή (1.4) δεν αναγνώριζε τις βασικές SMTLIB εντολές declare-datatypes, define-fun-rec και define-funs-rec. Η ασταθής έκδοση σε αρκετά απλά παραδείγματα επέστρεφε αδυναμία απόφασης ως προς την επιλυσιμότητα του τρέχοντος συνόλου περιορισμών. Ενημερώσαμε, λοιπόν, τους προγραμματιστές του CVC4 και εκείνοι τροποποίησαν τον επιλυτή, συμπεριλαμβάνοντας στα αρχεία ελέγχου δικά μας παραδείγματα SMT εισόδου. Οι αλλαγές αυτές τελικά συμπεριλήφθηκαν στην πρόσφατη ευσταθή έκδοση 1.5 του CVC4.

7 Εκκρεμούσες εργασίες

Τη δεδομένη χρονική στιγμή το CutEr λειτουργεί όπως περιγράψαμε παραπάνω, κάνοντας χρήση πολλών επιλυτών. Παρά ταύτα, υπάρχουν αρκετές βελτιώσεις που μπορούν να γίνουν στο εργαλείο, ώστε να δίνει ορθότερα αποτελέσματα και σε συντομότερο χρονικό διάστημα, ενδεχομένως χρησιμοποιώντας λιγότερους υπολογιστικούς πόρους.

7.1 Συνέπεια μοντέλου

Συναντώνται περιπτώσεις κατά τις οποίες δεν υπάρχει περιορισμός τύπου για κάποια παράμετρο του προβλήματος, συνεπώς δηλώνεται στον επιλυτή μία ελεύθερη μεταβλητή.

```
(declare-const x Term)
```

Στο επιστρεφόμενο μοντέλο, θα έχει ανατεθεί ένας αυθαίρετος τύπος και μία αυθαίρετη τιμή στην εν λόγω μεταβλητή. Αν υποθέσουμε ότι ο επιλυτής αποτιμά τη μεταβλητή ως atom, με βάση τον ορισμό του τύπου αυτού δεν αποκλείεται να επιστρέψει μία τιμή με στοιχεία μη αποδεκτούς ακεραίους, όπως:

```
(get-value (x))
(x (atom (ic (- 1) in)))
```

Αυτό θα μπορούσε να αποτραπεί για κάθε atom με τη χρήση του ποσοδείκτη forall.

```
(assert (
     forall
     ((t Term))
     (
          =>
          (is-atom t)
          (ilist-spec (av t))
     )
))
```

Παρά ταύτα, στα παραδείγματα που εξετάζουμε, οι επιλυτές αδυνατούν να ανταποκριθούν στην πολυπλοκότητα της παραπάνω εντολής. Μπορεί μαθηματικά η εφαρμογή του ποσοδείκτη είναι απλή και σαφής, ωστόσο πρακτικά συνεπάγεται πλήθος υπολογισμών στο εσωτερικό οποιουδήποτε επιλυτή αξιοποιούμε μέχρι στιγμής.

7.2 Επιλογές επιλυτών

Για την ενσωμάτωση ενός επιλυτή στο εργαλείο αρκεί μία γρήγορη ματιά στις βασικές οδηγίες χρήσης του. Παρά ταύτα, αν θέλουμε να εκμεταλλευτούμε στο έπακρο τα χαρακτηριστικά του, υπάρχουν πολλές παράμετροι που οφείλουμε να εξετάσουμε. Σε ένα βαθμό έχουν ληφθεί υπόψη μερικές εξ αυτών, σίγουρα όπως εξακολουθούν να αποτελούν αντικείμενο μελέτης.

7.2.1 Γραμμικότητα περιορισμών

Αρκετά περιορισμένη είναι η υποστήριξη προγραμμάτων που εμπλέκουν μη γραμμικούς συνδυασμούς των παραμέτρων εισόδου:

```
(set-logic LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (* x y) 17))
(check-sat)
```

Επιλυτές αποτυγχάνουν παντελώς να ανταποκριθούν σε μη γραμμικά σύνολα περιορισμών, όπως ο παραπάνω περιορισμός $x \cdot y = 17$, εφόσον επιλέγουμε λογική αποκλειστικά γραμμικής αριθμητικής, που επιλύεται ευκολότερα, λόγω της μειωμένης πολυπλοκότητάς της. Για παράδειγμα, ο CVC4 τερματίζει με κωδικό εξόδου 1 (σφάλμα) και το εξής διαγνωστικό μήνυμα:

```
(error "A non-linear fact was asserted to arithmetic in a linear logic.
The fact in question: (= (* x y) 17)
")
```

Καλό είναι να φροντίσουμε να μην οδηγούνται σε τερματισμό με μήνυμα σφάλματος, αντιθέτως, πρέπει να παρέχουν την ευκαιρία επίλυσης πιο συγκεκριμένων συνόλων περιορισμού, που προκύπτουν με σταθεροποίηση μιας εκ των μεταβλητών. Αν στο προηγούμενο παράδειγμα αλλάξουμε τη λογική σε μη γραμμική αριθμητική (set-logic NIA), ο CVC4 επιστρέφει απλώς "unknown", χωρίς να τερματίζει με σφάλμα. Ωστόσο, δεν επωφελούμαστε από το αποτέλεσμα, αντιθέτως, στις περιπτώσεις που είναι παρούσα μόνο γραμμική αριθμητική, ο επιλυτής αναζητά σε αρκετά ευρύτερο χώρο καταστάσεων.

7.2.2 Απαιτούμενη λογική

Οι SMTLIB επιλυτές υποστηρίζουν μία μαθηματική λογική με εξαιρετικά μεγάλο εύρος εφαρμογών. Η λογική αυτή σε αρκετούς επιλυτές, όπως ο CVC4, δηλώνεται μέσω της εντολής (set-logic ALL) και σε άλλους, όπως ο Z3, θεωρείται ως προεπιλεγμένη, αν δεν οριστεί κάποια διαφορετική. Δεν είναι όμως όλες οι πτυχές της λογικής απαραίτητες για τον υπολογισμό της επιλυσιμότητας των συνόλων περιορισμού που δημιουργεί το CutEr. Συνεπώς, μπορούμε να χρησιμοποιήσουμε κάποιο υποσύνολο της λογικής, μειώνοντας δραστικά την πολυπλοκότητα υπολογισμού μιας λύσης.

Αρκετά υποσύνολα της SMTLIB λογικής αναγράφονται στη σελίδα της βιβλιοθήκης [2]. Παρά ταύτα, δεν υποστηρίζονται όλα τα καταγεγραμμένα υποσύνολα από όλους τους επιλυτές, ενώ υπάρχουν και υποσύνολα που δεν συμπεριλαμβάνονται στη σχετική σελίδα και υποστηρίζονται από ορισμένους επιλυτές. Για το λόγο αυτό χρειάζεται ειδική διερεύνηση της ελάχιστης απαιτούμενης λογικής κάθε φορά που εισάγουμε νέο επιλυτή ή επεκτείνουμε το είδος των περιορισμών που χρησιμοποιεί το CutEr.

Χαρακτηριστικά αναφέρουμε οδηγίες για το σχηματισμό της ελάχιστης λογικής: Από τη στιγμή που χρησιμοποιούμε αναδρομικές συναρτήσεις, υποκρύπτεται η χρήση

ποσοδεικτών, συνεπώς η λογική χωρίς ποσοδείκτες (quantifier-free, QF) δεν μας αρκεί. Εξάλλου, υπάρχουν SMTLIB συναρτήσεις των οποίων δεν γνωρίζουμε το σώμα εκ των προτέρων (uninterpreted functions, UF). Τέλος, δεν μεταχειριζόμαστε διανύσματα (arrays, A), ακολουθίες bits σταθερού μήκους (fixed size bit vectors, BV) καθώς και αριθμούς κινητής υποδιαστολής (floating point, FP).

7.2.3 Συνδυασμοί επιλογών

Έχουμε ήδη αναφέρει την αναγκαιότητα της επιλογής παραγωγής μοντέλων (produce-models) σε όσους επιλυτές δεν την έχουν ενεργή εξ ορισμού, όπως και της επιλογής εύρεσης μοντέλων των αναδρομικά οριζόμενων συναρτήσεων (--fmf-fun) στον επιλυτή CVC4. Κάθε επιλυτής φέρει πλήθος επιλογών που αφορούν διαφόρους τομείς και είτε αφορούν τη γενική συμπεριφορά του επιλυτή, είτε ενεργοποιούν την υποστήριξη εξεζητημένων τμημάτων της λογικής, είτε βελτιώνουν την αποκρισιμότητα σε περιορισμένα σύνολα εντολών. Επομένως, κάθε φορά που προσθέτουμε ένα νέο επιλυτή στο CutEr οφείλουμε να πειραματιζόμαστε σχετικά, ενώ και ως προς τους ήδη εγκατεστημένους επιλυτές είναι πολλές οι επιλογές που δεν έχουν δοκιμαστεί.

7.3 Βελτιώσεις εργαλείου

Το CutEr συνολικά μπορεί να βελτιωθεί σε αρκετά σημεία του, ώστε να τρέχει σε μικρότερο χρόνο και να χρησιμοποιεί λιγότερους υπολογιστικούς πόρους. Κάποιες από τις τροποποιήσεις αυτές είναι περισσότερο εύκολο να υλοποιηθούν, ενώ άλλες ενδεχομένως αποτελούν ολόκληρα θέματα εργασιών. Εμείς τις αναφέρουμε συνοπτικά παρακάτω.

7.3.1 Διεπαφές προγραμματισμού

Η ιδέα της ύπαρξης πολλών επιλυτών που μιλούν την ίδια γλώσσα (SMTLIB) καθιστά ευκολότερη την ένταξη και χρήση τους σε ένα ενιαίο πρόγραμμα. Με χαρακτηριστικά ελάχιστες προσθήκες κώδικα μπορεί να ενσωματωθούν οι δυνατότητες ενός νέου επιλυτή. Παρά ταύτα, η επικοινωνία μέσω της κοινής γλώσσας προϋποθέτει τη σειριοποίηση και αποσειριοποίηση των δεδομένων, είτε κατά την αποστολή τους από το πρόγραμμα στον επιλυτή, είτε κατά την απάντηση του επιλυτή στο πρόγραμμα. Αυτή η διαρκής μετατροπή των δεδομένων από τύπους της Python σε συμβολοσειρές και εν συνεχεία σε τύπους του επιλυτή και αντιστρόφως απαιτεί επιπλέον χρόνο και μνήμη.

Μία πρόταση για την αντιμετώπιση του συγκεκριμένου προβλήματος είναι η αξιοποίηση των διεπαφών προγραμματισμού των διαφόρων επιλυτών που είναι γραμμένοι σε Python (Python APIs). Κατ' αυτό τον τρόπο ελαχιστοποιούμε τις μετατροπές των δεδομένων. Παράδειγμα εργαλείου που εφαρμόζει αυτή τη στρατηγική είναι μία υλοποίηση του KLEE [15], η οποία πραγματοποιεί έλεγχο προγραμμάτων εκτελώντας τα συμβολικά και επικοινωνώντας με τρεις επιλυτές (STP, Z3 και Boolector) μέσω του C++ API τους.

7.3.2 Λογικές μεταβλητές

Μία ιδιαιτερότητα της Erlang που μοιραία κληροδοτείται στο CutEr είναι η αναπαράσταση των λογικών μεταβλητών με χρήση των ατόμων "true" και "false".

Αρχικά, κατά την ανάπτυξη του κώδικα που επέτρεπε την επικοινωνία μέσω SMTLIB γλώσσας, καταβάλαμε προσπάθεια ώστε οι λογικές μεταβλητές να αναπαριστώνται στον επιλυτή με χρήση του ενσωματωμένου τύπου δεδομένων λογικής μεταβλητής, Bool. Αυτό το επιτύχαμε κωδικοποιώντας τα εν λόγω δύο άτομα ως λογικές μεταβλητές και τα υπόλοιπα ως λίστες ακεραίων, ενώ κατά την αποκωδικοποίηση μετατρέπαμε τις επιστρεφόμενες λογικές τιμές και πάλι σε άτομα. Παρόλο που σε όλες τις πρακτικές εφαρμογές δεν εμφανίστηκε κάποια δυσλειτουργία, αναγνωρίσαμε ότι για να είναι θεωρητικά ορθό το πρόγραμμα οφείλουμε να υπερφορτώσουμε και αρκετούς τελεστές, ώστε να είναι δυνατή, για παράδειγμα, η ισότητα λογικής μεταβλητής με μεταβλητή ατόμου, καθώς και η διαίρεση μιας λογικής μεταβλητής στην κεφαλή (head) και την ουρά (tail) του αντίστοιχου ατόμου.

7.3.3 Ενσωματωμένοι τύποι

Ορισμένοι επιλυτές, για παράδειγμα οι CVC4 και Z3, έχουν επεκτείνει τη γλώσσα SMTLIB ώστε να περιλαμβάνει τύπους δεδομένων όπως συμβολοσειρές (String). Οι συμβολοσειρές θα μπορούσαν να χρησιμεύσουν στην αναπαράσταση των ατόμων της Erlang στον επιλυτή. Από την άλλη ο Z3 ορίζει ως και αφηρημένους τύπους δεδομένων όπως ακολουθίες (Sequence, Seq), ενώ ήδη περιλαμβάνεται στη γλώσσα και ο αφηρημένος τύπος λίστας (List). Οι αφηρημένοι τύποι δεδομένων χρησιμεύουν στον ορισμό άλλων τύπων δεδομένων. Εκ πρώτης όψεως η ύπαρξή τους θα μπορούσε να διευκολύνει στον ορισμό των αναδρομικών δομών των τύπων δεδομένων (atom, list, tuple, bitstring και function), ωστόσο μέσα στον ορισμό του τύπου δεδομένων Term δεν επιτρέπεται η χρήση του όρου Term ως ορίσματος ενός αφηρημένου τύπου (όπως List), με αποτέλεσμα να είναι αδύνατον μία λίστα από Term να είναι ταυτόχρονα η ίδια υποτύπος του τύπου Term.

7.3.4 Συναρτήσεις Erlang

Για την αντιμετώπιση προγραμμάτων που κάνουν χρήση διαφόρων σύνθετων ενσωματωμένων συναρτήσεων της Erlang πρέπει να γραφούν οι αντίστοιχες μέθοδοι στην Python, ενδεχομένως εξειδικευμένες για κάθε επιλυτή, ώστε να μεταφράζονται οι επιβαλλόμενοι από τις συναρτήσεις περιορισμοί σε κατηγορήματα SMTLIB. Ας θεωρήσουμε το πρόγραμμα Erlang, που κάνει χρήση της bit_size (εικόνα 11):

```
-module(test4).
-export([fbit_size/1]).

-spec fbit_size(bitstring()) -> ok.
fbit_size(Bits) ->
     case bit_size(Bits) of
          Sz when Sz < 4 -> ok
     end.
```

εικόνα 11: Παράδειγμα με ενσωματωμένη συνάρτηση Erlang

Η συνάρτηση fbit_size παρουσιάζει σφάλμα όταν η παράμετρος εισόδου είναι ένα bitstring αποτελούμενο από τουλάχιστον 4 bits. Αυτή τη στιγμή το CutEr εντοπίζει μία είσοδο που να οδηγεί στο σφάλμα ως εξής: Αρχικά τρέχει με την ορισμένη είσοδο, το κενό bitstring (""). Έπειτα οδηγείται κατά σειρά στα "0", "00" και "110", τα οποία όλα έχουν αποδεκτό μήκος. Στο τέλος κατασκευάζει το "1110" και τερματίζει. Παρατηρούμε δηλαδή ότι το CutEr καλεί τον επιλυτή περνώντας του κατά σειρά τον περιορισμό η παράμετρος να αποτελείται από 1, 2, 3 και εν τέλει 4 bits. Όμως αυτή η μέθοδος δεν είναι αποδοτική. Θα μπορούσε απλά να ζητά κατ' ευθείαν από τον επιλυτή να του επιστρέψει ένα bitstring με το κατάλληλο μήκος, περιγράφοντας την εν λόγω απαίτηση με τη χρήση αναδρομής πάνω στο μήκος της παραμέτρου εισόδου.

Γενικότερα, με την προσθήκη των αναδρομικών συναρτήσεων παρέχεται η δυνατότητα να επιτρέψουμε σε ένα bitstring στον επιλυτή να αποτελείται από αυθαίρετο πλήθος bits, με συνέπεια τη μεταφορά της πολυπλοκότητας επίλυσης από τον πυρήνα του CutEr στον επιλυτή. Προς το παρόν όμως, ακόμα και σε σχετικά απλές περιπτώσεις, οι επιλυτές αποτυγχάνουν να χειριστούν bitstrings μη προκαθορισμένου μήκους.

Παρομοίως με τη συνάρτηση bit_size οφείλει να τροποποιηθεί η διαχείριση της συνάρτησης byte_size και να γραφούν μέθοδοι για πληθώρα άλλων σύνθετων ενσωματωμένων συναρτήσεων της Erlang.

7.3.5 Παραπλήσιες είσοδοι

Συμβαίνει συχνά ο επιλυτής να μην μπορεί να αποφανθεί για το κατά πόσο ένα σύνολο περιορισμών είναι ικανοποιήσιμο. Όπως είδαμε παραπάνω, το CutEr αντιμετωπίζει αυτές τις περιπτώσεις σταθεροποιώντας με τη σειρά τις παραμέτρους εισόδου στην τιμή που έλαβαν κατά την προηγούμενη εκτέλεση του ελεγχόμενου κώδικα. Συνεπώς, το νέο σύνολο περιορισμών περιέχει ακριβώς τους ίδιους περιορισμούς, επαυξημένους με ένα περιορισμό ισότητας. Πλέον ο επιλυτής καλείται ουσιαστικά να λύσει ένα αρκετά παρόμοιο πρόβλημα με το αρχικό.

Ας υποθέσουμε πως αν αντί να τερματίζουμε άμεσα τη διεργασία του επιλυτή μόλις μας επιστρέψει αποτυχία απόφασης, λύνουμε το νέο πρόβλημα στέλνοντας στην ίδια διεργασία την επιπρόσθετη εντολή. Μάλιστα, αυτή η εντολή θα μπορούσε να περαστεί ανάμεσα σε σώμα εντολών (push) – (pop), ώστε στη συνέχεια να υπάρχει η δυνατότητα να εξισώσουμε κι άλλη παράμετρο με μία τιμή, χωρίς να ισχύει ο περιορισμός τιμής της πρώτης παραμέτρου. Η προσέγγιση αυτή είναι αποδοτική από άποψη χρόνου και χρήσης υπολογιστικών πόρων, καθώς δε δημιουργούμε εκ νέου τη διεργασία του επιλυτή, ούτε στέλνουμε για συνεχόμενη φορά τους κοινούς περιορισμούς. Επίσης, κατ' αυτό τον τρόπο εκμεταλλευόμαστε ενδεχόμενη δυνατότητα του χρησιμοποιούμενου επιλυτή να αποκρίνεται γρηγορότερα σε παραπλήσια σύνολα περιορισμών, αφού έχει ήδη αναλύσει σημαντικό κομμάτι του χώρου καταστάσεων.

Σε πιο προχωρημένο στάδιο, παρατηρούμε ότι εξ ορισμού ο τρόπος με τον οποίο το CutEr παράγει σύνολα περιορισμών, καθώς επιχειρεί να επισκεφθεί γειτονικά μονοπάτια εκτέλεσης, έχει ως αποτέλεσμα σύνολα που να διαφέρουν πρακτικά σε ένα μόλις

περιορισμό. Ακολουθώντας, λοιπόν, το ίδιο σκεπτικό, θα μπορούσαμε να αντιμετωπίζουμε με μία διεργασία επιλυτή ολόκληρες ομάδες μονοπατιών εκτέλεσης. Σημειώνουμε ότι στην προκειμένη ιδέα, το εργαλείο οφείλει να ζητά την επίλυση κάθε μονοπατιού εκτέλεσης από την αντίστοιχη διεργασία επιλυτή και με τη σωστή προτεραιότητα, ώστε να αποστέλλεται ο ελάχιστος συνολικός αριθμός περιορισμών.

Τις διαπιστώσεις αυτές τις αναφέρουμε απλώς ως σκέψεις, αφού η συλλογιστική που λαμβάνει υπόψη τέτοιου είδους ομοιότητες, ενώ παρουσιάζει ιδιαίτερο ενδιαφέρον, υλοποιείται εξαιρετικά πολυπλοκότερα και, κατά συνέπεια, ξεφεύγει από τα πλαίσια της παρούσας εργασίας.

7.3.6 Προδιαγραφές μεθόδων

Καθώς το CutEr συλλέγει περιορισμούς από ένα μονοπάτι εκτέλεσης του ελεγχόμενου κώδικα, έρχεται αντιμέτωπο με συνθήκες που εξετάζουν, για παράδειγμα, αν το πλήθος των στοιχείων μιας μεταβλητής ως λίστας (ή πλειάδας) ισούται με έναν αριθμό. Στην τρέχουσα υλοποίηση, ο κόμβος αυτός αναλύεται σε τρία διαφορετικά μονοπάτια. Στο ένα μονοπάτι η μεταβλητή δεν είναι λίστα και, όπως είναι λογικό, οποιαδήποτε εντολή εκτελείται με τη συνθήκη της ισότητας του μήκους με τον αριθμό, δεν θα εκτελεστεί ποτέ. Παρά ταύτα, η προσπάθεια εύρεσης του μήκους μιας λίστας σε μία μεταβλητή που δεν αποτελεί λίστα εγείρει σφάλμα, το οποίο εντοπίζεται επιτυχώς με το σχετικό αντιπαράδειγμα. Στο δεύτερο μονοπάτι η μεταβλητή είναι μεν λίστα, αλλά δεν έχει το σωστό πλήθος στοιχείων, οπότε ο επιλυτής απαντά ότι το αντίστοιχο σύνολο περιορισμών είναι μη ικανοποιήσιμο, ο κώδικας που βρίσκεται υπό τη συνθήκη ισότητας δεν εκτελείται ποτέ και οποιοδήποτε σφάλμα περιέχει ορθώς δεν εντοπίζεται από το εργαλείο. Στο τρίτο μονοπάτι η μεταβλητή είναι πράγματι λίστα με πλήθος στοιχείων όσο και ο δεδομένος αριθμός, οπότε συνεχίζεται κανονικά η εξερεύνηση μονοπατιών με δεδομένη τη συνθήκη ισότητας.

Το πρόβλημα με την ανάλυση ενός κόμβου σε τρία διακριτά μονοπάτια εντοπίζεται κατά την αντιστροφή των περιορισμών που εφαρμόζεται κατά τον έλεγχο των προγραμμάτων. Πιο συγκεκριμένα, όταν αντιστρέφεται η συνθήκη ισότητας, το CutEr οδηγείται από το τρίτο μονοπάτι στο δεύτερο και ενδεχομένως χάνει την ευκαιρία να εντοπίσει το σφάλμα που παρουσιάζεται όταν η μεταβλητή δεν είναι λίστα. Μία απλή αντιμετώπιση είναι η διαίρεση της συνθήκης κατ’ αρχάς σε δύο μονοπάτια, ανάλογα με το αν η μεταβλητή είναι λίστα ή όχι. Εν συνεχεία, το μονοπάτι στο οποίο οδηγούμαστε όταν η μεταβλητή είναι λίστα οφείλει να διαιρείται σε δύο μονοπάτια, ανάλογα με το αν το πλήθος των στοιχείων της μεταβλητής (που είναι σίγουρο πως αποτελεί λίστα) ισούται με το δεδομένο αριθμό. Έτσι το εργαλείο αποκλείεται να χάσει κάποια περίπτωση, καθώς όλες προκύπτουν από ένα μονοπάτι εκτέλεσης με μία σειρά αντιστροφής συνθηκών.

Συμπληρωματικά αναφέρουμε πως πρέπει να καταστεί σαφές κάθε πότε καλείται μία μέθοδος της κλάσης Python του επιλυτή (με έμφαση στις μεθόδους που σχετίζονται bitstrings) και με ποιες προϋποθέσεις καλείται. Για παράδειγμα, όταν καλούμε τη μέθοδο

ισότητας ενός όρου με το άθροισμα δύο άλλων, επισυνάπτουμε συνθήκες που ζητούν τους όρους να έχουν αριθμητικό τύπο (ακέραιο ή πραγματικό). Εξετάζοντας την κατάσταση με τις εντολές που εστάλησαν στον επιλυτή, παρατηρούμε ότι οι εντολές περιορισμού τύπου σε αριθμητικό έχουν γραφεί πάνω από μία φορά. Ασφαλώς, μπορεί η εν λόγω επανάληψη να μην προσθέτει φόρτο σε έναν επιλυτή που αναγνωρίζει και διαγράφει τις πολλαπλές ταυτόσημες εντολές, παρά ταύτα η διαρκής επανάληψη αποστολής όμοιων περιορισμών κατά την εξέταση ενός προγράμματος με σημαντικό πλήθος αριθμητικών πράξεων σίγουρα επιβραδύνει τη διαδικασία επίλυσης.

7.3.7 Συμβατότητα παραμέτρων

Θεωρούμε το εξής παράδειγμα ελεγχόμενου κώδικα Erlang (εικόνα 12):

```
-module(test5).
-export([f13a/2]).

-spec f13a(fun(({any(), any()}) -> any()), tuple()) -> any().
f13a(F, X) ->
      case F(X) of
            1 ->
                  case X of
                        {1, 2, 3} -> error(unreachable_bug);
                        _ -> ok
                  end;
            _ ->
                  case X of
                        {4, 2} -> error(bug);
                        _ -> ok
                  end
      end.
```

εικόνα 12: Πέρασμα ασύμβατης παραμέτρου σε συνάρτηση

Η συνάρτηση αυτή περιέχεται στο σύνολο αρχείων ελέγχου του CutEr. Εξετάζοντάς την διαπιστώνουμε πως κατά την εκτέλεσή της μπορεί να προκύψουν δύο ειδών σφάλματα.

Το ένα είναι το προφανές, δηλαδή εκείνο κατά το οποίο η F είναι μία συνάρτηση που με παράμετρο X δεν επιστρέφει 1 και επιπλέον, η μεταβλητή X ισούται με την πλειάδα που αποτελείται από τους ακεραίους 2 και 5.

Το δεύτερο λάθος, το οποίο προς το παρόν εσφαλμένα δεν εντοπίζεται από το εργαλείο ελέγχου, δεν υπαγορεύεται απευθείας από τον κώδικα, αλλά παρουσιάζεται όταν η μεταβλητή X είναι πλειάδα (σύμφωνα με τον περιορισμό τύπου), αλλά περιέχει περισσότερα ή λιγότερα από δύο στοιχεία. Τότε και μόνο η προσπάθεια να κληθεί η F (που παίρνει σαν όρισμα μία πλειάδα ακριβώς δύο στοιχείων οποιουδήποτε τύπου) με παράμετρο εισόδου X οδηγεί σε σφάλμα.

Οπότε, αν η F είναι μία σταθερή συνάρτηση που επιστρέφει πάντοτε 1 και η X είναι η πλειάδα που αποτελείται από τους ακεραίους 1, 2 και 3, το πρόγραμμα θα τερματίσει με σφάλμα, αλλά λόγω της απόπειρας αποτίμησης της $F(X)$ και όχι λόγω της πρώτης περίπτωσης της εμφωλευμένης εντολής case.

Παραδείγματα εφαρμογής σε συνάρτηση μιας μεταβλητής ως παραμέτρου με ασύμβατο τύπο εμφανίζονται και σε άλλα αρχεία ελέγχου του CutEr και για τον εντοπισμό των αντίστοιχων σφαλμάτων πρέπει το εργαλείο να εξετάζει πρώτα αν τα ορίσματα ταιριάζουν με την καλούμενη συνάρτηση και κατόπιν να προχωρεί στον υπόλοιπο κώδικα. Κατ' αυτό τον τρόπο, με την αντιστροφή της συνθήκης συμβατότητας και μόνο θα προκύπτει άμεσα μία είσοδος που θα οδηγεί σε σφάλμα, ανεξάρτητα από τον υπόλοιπο κώδικα.

Τα παραπάνω ισχύουν επακριβώς και για την περίπτωση εφαρμογής λάθος πλήθους ορισμάτων σε μία συνάρτηση. Δηλαδή αν σε μία συνάρτηση μιας μεταβλητής επιχειρήσουμε να περάσουμε δύο παραμέτρους, αναμένουμε από το CutEr να εκτυπώσει το σφάλμα στο οποίο ούτως ή άλλως οδηγείται η Erlang.

7.3.8 Εξοικονόμηση πόρων

Η κλάση του συντονιστή ανταγωνισμού, καθώς και οι υποκλάσεις της, καλούν παράλληλα διαφορετικούς επιλυτές σε ξεχωριστά νήματα. Σε κάθε νήμα αντιγράφουμε το στιγμιότυπο της κλάσης Python του αντίστοιχου επιλυτή, προκειμένου να αποκτήσει πρόσβαση στα πεδία του. Τεχνικά είναι εφικτό να μην δημιουργούμε αντίγραφο κάθε αντικειμένου εξοικονομώντας μνήμη, είτε αποθηκεύοντας τα αντικείμενα σε κοινόχρηστη μνήμη και περνώντας τα κατά αναφορά στη συνάρτηση – χειριστή, είτε δημιουργώντας τα στις διεργασίες – παιδιά. Ωστόσο η πρώτη προσέγγιση παρουσιάζει δυσκολία κατά την αποθήκευση της δομής στην κοινόχρηστη μνήμη, ενώ η δεύτερη απαιτεί υλοποίηση του συντονιστή εκ του μηδενός και όχι ως υποκλάσης της βασικής κλάσης του συντονιστή. Κατά συνέπεια αρκεστήκαμε στην παρούσα, απλούστερη υλοποίηση.

8 Παράρτημα

Στο παράρτημα παραθέτουμε ορισμένα τμήματα του κώδικα, των οποίων η παρουσίαση ξεφεύγει από το κυρίως αντικείμενο της εργασίας, καθώς αποτελούν ενδεχομένως εξεζητημένες λεπτομέρειες της υλοποίησης. Όμως αρκετές από αυτές τις λεπτομέρειες παρουσιάζουν ιδιαίτερο ενδιαφέρον ή κάποια δυσκολία στην καταγραφή τους. Ο πλήρης κώδικας της εργασίας είναι δημοσίως αναρτημένος στη σελίδα του εργαλείου CutEr.

8.1 Βοηθητική δομή

Στην Python χρησιμοποιούμε μία ενδιάμεση, βοηθητική δομή για αποθήκευση των εντολών που πρόκειται να αποσταλούν στον επιλυτή και των αποτελεσμάτων που παραλαμβάνονται από αυτόν. Η δομή είναι δενδρική, με ενδιάμεσους κόμβους λίστες και φύλλα συμβολοσειρές. Εξυπηρετεί στην εύκολη προσθήκη και ανάγνωση κόμβων, διαδικασίες που θα δυσχέραιναν το έργο αν γίνονταν απ' ευθείας πάνω στις συμβολοσειρές SMTLIB.

8.1.1 Συναρτήσεις μετατροπής

Η μετατροπή από τη δενδρική δομή στη συμβολοσειρά SMTLIB που πρόκειται να γραφεί στην είσοδο του επιλυτή (εικόνα 13) πραγματοποιείται με μία απλή διάσχιση κατά βάθος (depth-first search, DFS).

```
def serialize(expr):
      if isinstance(expr, list):
            return "(" + " ".join(map(serialize, expr)) + ")"
      else:
            return expr
```

εικόνα 13: Ορισμός συνάρτησης serialize

Αντιθέτως, η κατασκευή της δενδρικής δομής από μία συμβολοσειρά SMTLIB προερχόμενη από τον επιλυτή δεν είναι τόσο απλή διαδικασία (εικόνα 14). Η συνάρτηση λαμβάνει, πέραν της εισόδου, το σημείο ως το οποίο έχει αναγνωσθεί η είσοδος. Από την άλλη, επιστρέφει, εκτός από τον κόμβο της δομής, τα δύο σημεία που οριοθετούν σε ποια υποσυμβολοσειρά της εισόδου αντιστοιχεί ο επιστρεφόμενος κόμβος. Πρώτα καταναλώνει όλους τους λευκούς χαρακτήρες από την αρχή της συμβολοσειράς εισόδου. Στη συνέχεια, εξετάζει τον επόμενο, μη λευκό, χαρακτήρα.

- Ο πρώτος μη λευκός χαρακτήρας είναι παρένθεση που ανοίγει, "(". Ο τρέχων κόμβος είναι εσωτερικός, δηλαδή θα μετατραπεί σε λίστα. Μέχρι να βρει παρένθεση που κλείνει, ")", προσπερνά κάθε λευκό χαρακτήρα, διαβάζει αναδρομικά ένα παιδί του τρέχοντος κόμβου και τον προσθέτει στη λίστα.
- Ο πρώτος μη λευκός χαρακτήρας δεν είναι παρένθεση που ανοίγει, "(" . Ο τρέχων κόμβος είναι τερματικός και η συνάρτηση θα επιστρέψει μία υποσυμβολοσειρά της εισόδου. Ελέγχουμε τον επόμενο χαρακτήρα.

- Ο χαρακτήρας είναι εισαγωγικό, """. Τότε η έκφραση είναι κάποια συμβολοσειρά SMTLIB. Στην εφαρμογή μας η μόνη περίπτωση να συμβεί αυτό το γεγονός είναι να μας επιστρέψει ένα μήνυμα σφάλματος ο επιλυτής. Τη διαβάζουμε όλη μέχρι να βρούμε εισαγωγικό που να μην ακολουθείται απευθείας από νέο εισαγωγικό (δηλαδή να μην είναι ο χαρακτήρας διαφυγής, escape character, του εισαγωγικού).
- Ο χαρακτήρας δεν είναι εισαγωγικό, """. Άρα είναι μία λέξη, την οποία διαβάζουμε μέχρι να βρούμε λευκό χαρακτήρα ή παρένθεση που κλείνει, ")", οπότε σηματοδοτείται το τέλος της λέξης.

```
def unserialize(smt, cur = None):
	if cur is None:
		return unserialize(smt, 0)[0]
	while smt[cur].isspace():
		cur += 1
	if smt[cur] == "(":
		nodes = []
		beg = cur
		cur += 1
		while True:
			while smt[cur].isspace():
				cur += 1
			if smt[cur] == ")":
				break
			node = unserialize(smt, cur)
			nodes.append(node[0])
			cur = node[2]
		end = cur + 1
		return (nodes, beg, end)
	else:
		beg = cur
		if smt[cur] == "\"":
			cur += 1
			while True:
				if smt[cur] == "\"":
					cur += 1
					if smt[cur] == "\"":
						cur += 1
					else:
						break;
				else:
					cur += 1
		else:
			while smt[cur] != ")" and not smt[cur].isspace():
				cur += 1
		end = cur
		return (smt[beg:end], beg, end)
```

εικόνα 14: Ορισμός συνάρτησης unserialize

8.1.2 Ανάπτυξη εκφράσεων

Θεωρούμε τον παρακάτω κώδικα Erlang (εικόνα 15):

```
-module(test6).
-export([f22/2]).

-spec f22(bitstring(), integer()) -> ok.
f22(X, Y) ->
     case X of
          <<42:Y>> -> error(not_ok);
          _ -> ok
     end.
```

εικόνα 15: Αναγνώριση σύνθετων απαντήσεων SMTLIB

Αν το ελέγξουμε με το CutEr και διαλέξουμε για επιλυτή το CVC4, θα λάβουμε μία από τις αναμενόμενες λύσεις, την $(x, y) = (1010100, 6)$, σε μορφή SMTLIB.

```
(get-value (x))
((x (str (
     sc
     true
     (sc false (sc true (sc false (sc true (sc false sn)))))
))))
(get-value (y))
((y (int 6)))
```

Αν διαλέξουμε αυτή τη φορά για επιλυτή τον Z3, θα λάβουμε την ίδια λύση, αλλά διατυπωμένη με τη χρήση της έκφρασης let και μιας βοηθητικής μεταβλητής:

```
(get-value (x))
((x (
     let
     ((a!1 (sc false (sc true (sc false sn)))))
     (str (sc true (sc false (sc true a!1))))
)))
(get-value (y))
((y (int 6)))
```

Οι εκφράσεις let στη γλώσσα SMTLIB [7] ορίζουν τοπικές μεταβλητές και επιτρέπεται να εμφανίζονται εμφωλευμένες η μία μέσα στην άλλη. Η εμβέλεια των τοπικών μεταβλητών περιορίζεται στο δεύτερο όρισμα της έκφρασης let. Όπως διαπιστώνουμε, κάποιοι επιλυτές συνηθίζουν να τις μεταχειρίζονται για να απλοποιούν τις εκφράσεις που εκτυπώνουν και σε αρκετές περιπτώσεις, για να αποφεύγουν την επανάληψη όμοιων εκφράσεων. Κατά την ανάγνωση των αποτελεσμάτων, όμως, από την κλάση

Python του επιλυτή, θέλουμε οι εκφράσεις να είναι πλήρεις και έτοιμες για αποκωδικοποίηση, οπότε τις αναπτύσσουμε με την ακόλουθη συνάρτηση (εικόνα 15).

```
def expand_lets(expr, lets = {}):
      if not isinstance(expr, list):
            if expr in lets:
                  return lets[expr]
            else:
                  return expr
      elif not expr:
            return []
      elif expr[0] == "let":
            assert len(expr) == 3, \
                   "expand_lets({})".format(str(expr))
            lets_copy = lets.copy()
            for var in expr[1]:
                  lets_copy[var[0]] = expand_lets(var[1], lets)
            return expand_lets(expr[2], lets_copy)
      else:
            return [expand_lets(item, lets) for item in expr]
```

εικόνα 16: Ορισμός συνάρτησης expand_lets

Για την ανάπτυξη των εκφράσεων let, διαβάζουμε από την έξοδο του επιλυτή μία συμβολοσειρά, την οποία και μετατρέπουμε στην αντίστοιχη δενδρική δομή. Έπειτα καλούμε την παραπάνω συνάρτηση με όρισμα μόνο τη δομή, οπότε στη μεταβλητή lets έχουμε ένα κενό λεξικό. Το λεξικό σε κάθε αναδρομική κλήση της συνάρτησης περιέχει αντιστοιχίσεις των τοπικών μεταβλητών που είναι ορατές με τις εκφράσεις τους.

- Αν ο τρέχων κόμβος είναι τερματικός, ελέγχουμε αν αποτελεί τοπική μεταβλητή που περιέχεται στο λεξικό. Αν ναι, τότε αντικαθίσταται από την αντιστοιχισμένη έκφραση, διαφορετικά παραμένει ως έχει.
- Αν ο τρέχων κόμβος είναι λίστα που περιέχει τουλάχιστον ένα στοιχείο και το πρώτο στοιχείο είναι η συμβολοσειρά "let", πρώτα δημιουργούμε ένα αντίγραφο του λεξικού. Έπειτα αποτιμούμε τις εκφράσεις των τοπικών μεταβλητών που ορίζονται στο πρώτο όρισμα της let, καλώντας αναδρομικά την ίδια συνάρτηση με το παλιό λεξικό. Τις τοπικές μεταβλητές τις προσθέτουμε στο νέο λεξικό με τις αντίστοιχες εκφράσεις τους. Στο τέλος αντικαθιστούμε τον τρέχοντα κόμβο με το δεύτερο όρισμα, αποτιμημένο με το νέο λεξικό.
- Αν ο τρέχων κόμβος είναι οποιαδήποτε άλλη λίστα, αντικαθιστούμε τα παιδιά του υπολογίζοντάς τα αναδρομικά με το υπάρχον λεξικό.

8.2 Τύποι δεδομένων

Ακολουθεί ο κώδικας ορισμού των τύπων δεδομένων (εικόνα 17), με τους οποίους κωδικοποιούμε τις παραμέτρους της Erlang στη γλώσσα SMTLIB των επιλυτών. Οι τύποι

δεδομένων περιλαμβάνουν αναδρομή (αφού για παράδειγμα ένας όρος μπορεί να είναι λίστα όρων), επομένως ορίζουμε όλους τους χρησιμοποιούμενους τύπους σε μία δήλωση declare-datatypes και όχι σε ξεχωριστές εντολές declare-datatype.

```
(declare-datatypes () (
     (
          Term
          (int (iv Int))
          (real (rv Real))
          (list (lv TList))
          (tuple (tv TList))
          (atom (av IList))
          (str (sv SList))
          (fun (fv Int))
     )
     (
          TList
          (tn)
          (tc (th Term) (tt TList))
     )
     (
          IList
          (in)
          (ic (ih Int) (it IList))
     )
     (
          SList
          (sn)
          (sc (sh Bool) (st SList))
     )
     (
          FList
          (fn)
          (fc (fx TList) (fy Term) (ft FList))
     )
))
```

εικόνα 17: Αλγεβρικοί τύποι δεδομένων SMTLIB

Οι συναρτήσεις λήψης τιμής των αριθμητικών τύπων είναι συντομογραφίες των "ακέραια τιμή" (integer value, iv) και "πραγματική τιμή" (real value, rv) αντίστοιχα. Ομοίως, για τις υπόλοιπες περιπτώσεις όρων, οι συναρτήσεις lv, tv, av, sv και fv δίνουν την τιμή (value) για τύπο list, tuple, atom, string (bitstring) και function.

Παρόμοιο σκεπτικό για το όνομα των συναρτήσεων κατασκευής και λήψης τιμής ακολουθείται και στους βοηθητικούς τύπους δεδομένων "λίστα όρων" (term list, TList), "λίστα ακεραίων" (integer list, IList), "λίστα bits" (bit-string list, SList) και "λίστα συνάρτησης" (function list, FList). Το πρώτο γράμμα είναι ο τύπος της λίστας (t, i, s, f). Το δεύτερο γράμμα είναι n (nil) για κατασκευή φύλλου λίστας, c (cons) για κατασκευή κόμβου λίστας, h (head) για λήψη της κεφαλής της λίστας και t (tail) για λήψη της ουράς της λίστας.

Ειδικά για τον τύπο FList, τη λίστα όρων που αποτελεί τα ορίσματα ενός κόμβου τη λαμβάνουμε με το fx, ενώ την τιμή της συνάρτησης για τα συγκεκριμένα ορίσματα με το fy, όπου x η ανεξάρτητη και y η εξαρτημένη μεταβλητή στη συνήθη έκφραση $y = f(x)$.

8.3 Μαθηματικές πράξεις

Η γλώσσα SMTLIB υποστηρίζει bitwise πράξεις μεταξύ ακεραίων μέσω της μετατροπής τους σε bitvectors σταθερού μήκους. Ο περιορισμός του σταθερού μήκους θέτει ένα όριο στο εύρος τιμών που μπορούν να λάβουν οι ακέραιοι, οπότε επιχειρήσαμε να υλοποιήσουμε τις πράξεις με χρήση αναδρομικών συναρτήσεων. Αφού υλοποιήσουμε την αναδρομή με τις συνθήκες τερματισμού, για να περιορίσουμε την πολυπλοκότητα επίλυσης του προβλήματος που προκύπτει, προσθέτουμε επιπλέον περιορισμούς που φράσσουν τις τιμές των αγνώστων παραμέτρων, ανάλογα με το είδος της πράξης και τις τιμές των υπολοίπων τελούμενων. Η διαδικασία που ακολουθήσαμε περιγράφεται ακολούθως.

8.3.1 Bitwise AND και OR

Κατ' αρχάς, σημειώνουμε πως παρακάτω η χρήση των συμβόλων $\wedge$, $\vee$ και $\neg$ γίνεται μεταξύ bits ή λογικών μεταβλητών και εννοούν τη λογική πράξη "και" (boolean and), τη λογική πράξη "ή" (boolean or) και τη λογική πράξη "όχι" (boolean not) αντίστοιχα. Μεταξύ ακεραίων τα σύμβολα $\odot$, $\oplus$ και ο τελεστής $\overline{n}$ δηλώνουν τις ίδιες λογικές πράξεις bitwise (bitwise and, bitwise or και bitwise not).

Θεωρούμε ότι η δυαδική αναπαράσταση των αριθμών είναι $(x, a, b) = (\dots x_2 x_1 x_0, \dots a_2 a_1 a_0, \dots b_2 b_1 b_0)$. Η ισότητα bitwise and, $x = a \odot b$, συνεπάγεται τις σχέσεις $x_i = a_i \wedge b_i \; \forall i \geq 0$. Γράφουμε την αναδρομική συνθήκη ως:

$$x = a \odot b \Leftrightarrow (x_0 = a_0 \wedge b_0) \wedge (\dots x_2 x_1 = \dots a_2 a_1 \odot \dots b_2 b_1)$$

η οποία μαθηματικά είναι ισοδύναμη με την

$$x = a \odot b \Leftrightarrow (\mathrm{mod}(x, 2) = \mathrm{mod}(a, 2) \wedge \mathrm{mod}(b, 2)) \wedge (\mathrm{div}(x, 2) = \mathrm{div}(a, 2) \odot \mathrm{div}(b, 2))$$

όπου $\mathrm{div}(n, 2) = \cdots n_2 n_1$ και $\mathrm{mod}(n, 2) = n_0$ το πηλίκο και το υπόλοιπο της ακέραιας διαίρεσης του $n = \cdots n_2 n_1 n_0$ με το 2, αντίστοιχα. Οι τελεστές $\mathrm{div}(\cdot, 2)$ και $\mathrm{mod}(\cdot, 2)$, ακόμα και στους αρνητικούς αριθμούς, εξακολουθούν να χωρίζουν το τελευταίο ψηφίο από τα υπόλοιπα ψηφία. Άρα οι αναδρομική συνθήκη δεν επηρεάζεται από τα πρόσημα των εμπλεκόμενων αριθμών.

Σημειώνουμε πως η μετατροπή ενός bit n_0 σε SMTLIB λογική μεταβλητή γίνεται εύκολα με την έκφραση $n_0 = 1$ ή την ισοδύναμη $\neg(n_0 = 0)$. Επίσης, $n = \sum_i n_i \cdot 2^i \ \forall n \geq 0$, με το άθροισμα να συγκλίνει, αφού τα πιο σημαντικά ψηφία κάθε μη αρνητικού ακεραίου είναι 0 (δηλαδή $\exists\ i_0 \geq 0$ τέτοιο ώστε $n_i = 0\ \forall\ i \geq i_0$).

Ομοίως, η ισότητα bitwise or, $x = a \oplus b$, υποδηλώνει $x_i = a_i \vee b_i\ \forall i \geq 0$. Η αναδρομική συνθήκη γράφεται:

$$x = a \oplus b \Leftrightarrow (x_0 = a_0 \vee b_0) \wedge (\dots x_2 x_1 = \ \dots a_2 a_1 \oplus \dots b_2 b_1)$$

δηλαδή

$$x = a \oplus b \Leftrightarrow (\mathrm{mod}(x,2) = \mathrm{mod}(a,2) \vee \mathrm{mod}(b,2)) \wedge (\mathrm{div}(x,2) = \mathrm{div}(a,2) \oplus \mathrm{div}(b,2))$$

Στην πράξη bitwise and οι συνθήκες τερματισμού της αναδρομής είναι:

$$(a = 0) \vee (b = 0) \Rightarrow x = 0$$

$$x = -1 \Rightarrow (a = -1) \wedge (b = -1)$$

ενώ στην πράξη bitwise or είναι:

$$(a = -1) \vee (b = -1) \Rightarrow x = -1$$

$$x = 0 \Rightarrow (a = 0) \wedge (b = 0)$$

αφού $n \odot 0 = 0\ \forall n, n \oplus -1 = -1\ \forall n,$

$a \odot b \neq -1\ \forall\ (a,b) \neq (-1,-1)$ και $a \oplus b \neq 0\ \forall (a,b) \neq (0,0)$.

Καταγράφουμε, τέλος, ορισμένες συνθήκες ανισότητας που φράσσουν τις τιμές των παραμέτρων.

Στην πράξη bitwise and, αν ένα από τα τελούμενα είναι μη αρνητικός ακέραιος, το αποτέλεσμα θα είναι και αυτό μη αρνητικός ακέραιος, αφού τα πιο σημαντικά ψηφία είναι 0. Επιπροσθέτως, από τον πίνακα αλήθειας της λογικής πράξης, αν $x_i = a_i \wedge b_i\ \forall i$, προκύπτει εύκολα (για $a \geq 0$ ή $b \geq 0$ αντίστοιχα):

$$x_i \leq a_i\ \forall i \Rightarrow \sum_i x_i \cdot 2^i \leq \sum_i a_i \cdot 2^i \Rightarrow 0 \leq x \leq a$$

$$x_i \leq b_i\ \forall i \Rightarrow \sum_i x_i \cdot 2^i \leq \sum_i b_i \cdot 2^i \Rightarrow 0 \leq x \leq b$$

Άρα $a \geq 0 \Rightarrow 0 \leq x \leq a$ και $b \geq 0 \Rightarrow 0 \leq x \leq b$.

Στην πράξη bitwise or, αν το αποτέλεσμα είναι μη αρνητικός ακέραιος, τότε και τα δύο τελούμενα είναι μη αρνητικοί ακέραιοι, αφού τα πιο σημαντικά ψηφία είναι 0. Η πρόταση

ισχύει και αντιστρόφως κατά προφανή τρόπο. Παράλληλα, από τον πίνακα αλήθειας, αν $x_i = a_i \vee b_i \; \forall i$ και $x \geq 0$, έχουμε:

$$a_i \leq x_i \; \forall i \Rightarrow \sum_i a_i \cdot 2^i \leq \sum_i x_i \cdot 2^i \Rightarrow 0 \leq a \leq x$$

$$b_i \leq x_i \; \forall i \Rightarrow \sum_i b_i \cdot 2^i \leq \sum_i x_i \cdot 2^i \Rightarrow 0 \leq b \leq x$$

$$x_i \leq a_i + b_i \; \forall i \Rightarrow \sum_i x_i \cdot 2^i \leq \sum_i a_i \cdot 2^i + \sum_i b_i \cdot 2^i \Rightarrow 0 \leq x \leq a + b$$

Άρα $(a \geq 0) \wedge (b \geq 0) \Rightarrow x \geq 0$ και $x \geq 0 \Rightarrow 0 \leq a, b \leq x \leq a + b$.

Εξετάζουμε τώρα τους αρνητικούς ακεραίους. Οι αρνητικοί κωδικοποιούνται με το συμπλήρωμα ως προς δύο, δηλαδή $-n = \overline{n} + 1 \; \forall n < 0 \Rightarrow \overline{n} = \sum_i \overline{n_i} = -n - 1$.

Στην πράξη bitwise and, αν το αποτέλεσμα είναι αρνητικός ακέραιος, τότε και τα δύο τελούμενα είναι αρνητικοί ακέραιοι, αφού τα πιο σημαντικά ψηφία είναι 1. Η πρόταση ισχύει και αντιστρόφως, ενώ από τον πίνακα αλήθειας:

$$x_i \leq a_i \; \forall i \Rightarrow \overline{x_i} \geq \overline{a_i} \; \forall i \Rightarrow \sum_i \overline{x_i} \cdot 2^i \geq \sum_i \overline{a_i} \cdot 2^i \Rightarrow \overline{x} \geq \overline{a} \geq 0 \Rightarrow$$
$$-x - 1 \geq -a - 1 \geq 0 \Rightarrow -x \geq -a \geq 1 \Rightarrow x \leq a < 0$$

Ομοίως, $x_i \leq b_i \; \forall i \Rightarrow x \leq b < 0$. Εφαρμόζοντας τον κανόνα De Morgan:

$$a \odot b = x \Rightarrow \overline{a \odot b} = \overline{x} \Rightarrow \overline{a} \oplus \overline{b} = \overline{x} \Rightarrow \overline{a_i} + \overline{b_i} \geq \overline{x_i} \; \forall i \Rightarrow$$
$$\sum_i \overline{a_i} \cdot 2^i + \sum_i \overline{b_i} \cdot 2^i \geq \sum_i \overline{x_i} \cdot 2^i \Rightarrow \overline{a} + \overline{b} \geq \overline{x} \geq 0 \Rightarrow$$
$$-a - 1 - b - 1 \geq -x - 1 \geq 0 \Rightarrow a + b < x < 0$$

Άρα $(a < 0) \wedge (b < 0) \Rightarrow x < 0$ και $x < 0 \Rightarrow a + b < x \leq a, b < 0$.

Στην πράξη bitwise or, αν τουλάχιστον ένα από τα τελούμενα είναι αρνητικός ακέραιος, το αποτέλεσμα θα είναι και αυτό αρνητικός ακέραιος, αφού τα πιο σημαντικά ψηφία είναι 1, ενώ από τον πίνακα αλήθειας:

$$a_i \leq x_i \; \forall i \Rightarrow \overline{a_i} \geq \overline{x_i} \; \forall i \Rightarrow \sum_i \overline{a_i} \cdot 2^i \geq \sum_i \overline{x_i} \cdot 2^i \Rightarrow \overline{a} \geq \overline{x} \geq 0 \Rightarrow$$
$$-a - 1 \geq -x - 1 \geq 0 \Rightarrow -a \geq -x \geq 1 \Rightarrow a \leq x < 0$$

Ομοίως, $b_i \leq x_i \; \forall i \Rightarrow b \leq x < 0$.

Άρα $a < 0 \Rightarrow a \leq x < 0$ και $b < 0 \Rightarrow b \leq x < 0$.

Συνολικά, ο κώδικας SMTLIB της συνθήκης $x = a \odot b$ καλείται ως (int-and x a b), όπου:

```
(define-fun-rec int-and-rec ((x Int) (a Int) (b Int)) Bool (
      or
      (= a x 0)
      (= b x 0)
      (= a b x (- 1))
      (
            and
            (= (
                  and
                  (not (= (mod a 2) 0))
                  (not (= (mod b 2) 0))
            ) (not (= (mod x 2) 0)))
            (int-and-rec (div x 2) (div a 2) (div b 2))
      )
))
(define-fun int-and ((x Int) (a Int) (b Int)) Bool (
      and
      (=> (>= a 0) (<= 0 x a))
      (=> (>= b 0) (<= 0 x b))
      (=> (and (< a 0) (< b 0)) (< (+ a b) x 0))
      (=> (< x 0) (and (< a 0) (<= x a) (< b 0) (<= x b)))
      (int-and-rec x a b)
))
```

εικόνα 18: Συνάρτηση υπολογισμού bitwise AND

Ενώ ο κώδικας της $x = a \oplus b$ καλείται ως (int-or x a b), όπου:

```
(define-fun-rec int-or-rec ((x Int) (a Int) (b Int)) Bool (
      or
      (= a x (- 1))
      (= b x (- 1))
      (= a b x 0)
      (
            and
            (= (
                  or
                  (not (= (mod a 2) 0))
                  (not (= (mod b 2) 0))
            ) (not (= (mod x 2) 0)))
            (int-or-rec (div x 2) (div a 2) (div b 2))
      )
))
(define-fun int-or ((x Int) (a Int) (b Int)) Bool (
      and
      (=> (< a 0) (and (<= a x) (< x 0)))
      (=> (< b 0) (and (<= b x) (< x 0)))
      (=> (and (>= a 0) (>= b 0)) (<= 0 x (+ a b)))
      (=> (>= x 0) (and (<= 0 a x) (<= 0 b x)))
      (int-or-rec x a b)
))
```

εικόνα 19: Συνάρτηση υπολογισμού bitwise OR

8.3.2 Bitwise XOR

Ορίζουμε επιπροσθέτως το σύμβολο $*$ για τη λογική πράξη "αποκλειστικό ή" (boolean xor) μεταξύ λογικών μεταβλητών ή bits και το $\circledast$ για την ίδια λογική πράξη μεταξύ ακεραίων, bitwise (bitwise xor). Για την πράξη bitwise xor ισχύει, πέραν της αντιμεταθετικής ιδιότητας και η κυκλική συμμετρία $x = a \circledast b \Leftrightarrow a = b \circledast x \Leftrightarrow b = x \circledast a$.

Θεωρώντας την ίδια αναπαράσταση για τους ακεραίους (x, a, b), η ισότητα $x = a \circledast b$ συνεπάγεται τις σχέσεις $x_i = a_i * b_i \; \forall i$ και η αναδρομική σχέση γράφεται:

$$x = a \circledast b \Leftrightarrow (x_0 = a_0 * b_0) \wedge (\dots x_2 x_1 = \dots a_2 a_1 \circledast \dots b_2 b_1)$$

ή

$$x = a \circledast b \Leftrightarrow (\mathrm{mod}(x, 2) = \mathrm{mod}(a, 2) * \mathrm{mod}(b, 2)) \wedge (\mathrm{div}(x, 2) = \mathrm{div}(a, 2) \circledast \mathrm{div}(b, 2))$$

Οι συνθήκες τερματισμού της αναδρομής προκύπτουν και πάλι όταν κατόπιν διαδοχικών ακεραίων διαιρέσεων ένας αριθμός (χωρίς βλάβη της γενικότητας ας θεωρήσουμε τον

x) γίνεται $x = 0$ ή $x = -1$, οπότε ισχύουν οι σχέσεις $0 = a \circledast b \Leftrightarrow a = b$ και $-1 = a \circledast b \Leftrightarrow a = \overline{b} \Leftrightarrow a + b = -1$.

Συνολικά, ο κώδικας της $x = a \circledast b$ καλείται ως (int-xor x a b), όπου:

```
(define-fun-rec int-xor ((x Int) (a Int) (b Int)) Bool (
     or
     (and (= x 0) (= a b))
     (and (= a 0) (= b x))
     (and (= b 0) (= x a))
     (and (= x (- 1)) (= (+ a b) (- 1)))
     (and (= a (- 1)) (= (+ b x) (- 1)))
     (and (= b (- 1)) (= (+ x a) (- 1)))
     (
          and
          (= (
               xor
               (not (= (mod a 2) 0))
               (not (= (mod b 2) 0))
          ) (not (= (mod x 2) 0)))
          (int-xor (div x 2) (div a 2) (div b 2))
     )
))
```

εικόνα 20: Συνάρτηση υπολογισμού bitwise XOR

8.3.3 Ακέραια δύναμη

Περιορίζουμε τη γνωστή μαθηματική πράξη της δύναμης, $p = b^e$, σε πραγματική βάση (base, b) με ακέραιο εκθέτη (exponent, e). Δεν υπάρχει συνάρτηση SMTLIB υπολογισμού της δύναμης, οπότε θα την υλοποιήσουμε με αναδρομή στον ακέραιο εκθέτη:

$$p = b^e, (b \neq 0) \wedge (e \in \mathbb{Z}) \Leftrightarrow \begin{cases} p \cdot b = b^{e+1} & e < 0 \\ p = 1 & e = 0 \\ p/b = b^{e-1} & e > 0 \end{cases}$$

Για να περιορίσουμε το εύρος αναζήτησης του επιλυτή, αναλύουμε την αναδρομή σε επιπλέον υποπεριπτώσεις, ώστε να φράξουμε και τις παραμέτρους με ανισότητες.

Αρχικά ελέγχουμε αν η βάση παίρνει τιμές από το $\{-1, 0, 1\}$:

- $b = -1$ και
  - $\mathrm{mod}(e, 2) = 0$: τότε $p = 1$
  - $\mathrm{mod}(e, 2) = 1$: τότε $p = -1$
- $b = 0$: τότε $p = 0$ και $e \neq 0$
- $b = 1$: τότε $p = 1$

Ύστερα, για τις υπόλοιπες τιμές της βάσης, εξετάζουμε την τιμή του εκθέτη:

- $e = 0$: τότε $p = 1$, αρκεί $b \neq 0$.
- $e = 1$: τότε $b = p$.
- $e > 1$: τότε $p/b = b^{e-1}$ και
    - $1 < b < p$
    - $0 < p < b < 1$
    - $-1 < b < 0$
        - $\mathrm{mod}(e, 2) = 0$ και $0 < p < -b < 1$
        - $\mathrm{mod}(e, 2) = 1$ και $-1 < b < p < 0$
    - $b < -1$
        - $\mathrm{mod}(e, 2) = 0$ και $1 < -b < p$
        - $\mathrm{mod}(e, 2) = 1$ και $p < b < -1$
- $e = -1$: τότε $p \cdot b = 1$
- $e < -1$: τότε $p \cdot b = b^{e+1}$ και
    - $0 < p < 1 < b$
    - $0 < b < 1 < p$
    - $-1 < b < 0$
        - $\mathrm{mod}(e, 2) = 0$ και $0 < -b < 1 < p$
        - $\mathrm{mod}(e, 2) = 1$ και $p < -1 < b < 0$
    - $b < -1$
        - $\mathrm{mod}(e, 2) = 0$ και $0 < p < 1 < -b$
        - $\mathrm{mod}(e, 2) = 1$ και $b < -1 < p < 0$

Οπότε τελικά ο κώδικας SMTLIB για τον περιορισμό $p = b^e$ είναι (real-pow p b e). Η συνάρτηση real-pow υλοποιείται ως εξής:

```
(define-fun-rec real-pow ((p Real) (b Real) (e Int))
Bool (
     or
     (and (= b 0) (not (= e 0)) (= p 0))
     (and (= b 1) (= p 1))
     (and (= b (- 1)) (= (mod e 2) 0) (= p 1))
     (and (= b (- 1)) (= (mod e 2) 1) (= p (- 1)))
     (and (> e 1) (< 1 b p) (real-pow (/ p b) b (- e
1)))
     (and (> e 1) (< 0 p b 1) (real-pow (/ p b) b (- e
1)))
     (and (> e 1) (= (mod e 2) 0) (< (- 1) b 0 p (- b)
1) (real-pow (/ p b) b (- e 1)))
     (and (> e 1) (= (mod e 2) 0) (< b (- 1) 1 (- b) p)
(real-pow (/ p b) b (- e 1)))
     (and (> e 1) (= (mod e 2) 1) (< (- 1) b p 0) (real-
pow (/ p b) b (- e 1)))
     (and (> e 1) (= (mod e 2) 1) (< p b (- 1)) (real-
pow (/ p b) b (- e 1)))
     (and (= e 1) (= b p))
     (and (= e 0) (not (= b 0)) (= p 1))
     (and (= e (- 1)) (= (* b p) 1))
     (and (< e (- 1)) (< 0 p 1 b) (real-pow (* p b) b (+
e 1)))
     (and (< e (- 1)) (< 0 b 1 p) (real-pow (* p b) b (+
e 1)))
     (and (< e (- 1)) (= (mod e 2) 0) (< (- 1) b 0 (- b)
1 p) (real-pow (* p b) b (+ e 1)))
     (and (< e (- 1)) (= (mod e 2) 0) (< b (- 1) 0 p 1
(- b)) (real-pow (* p b) b (+ e 1)))
     (and (< e (- 1)) (= (mod e 2) 1) (< p (- 1) b 0)
(real-pow (* p b) b (+ e 1)))
     (and (< e (- 1)) (= (mod e 2) 1) (< b (- 1) p 0)
(real-pow (* p b) b (+ e 1)))
))
```

εικόνα 21: Συνάρτηση υπολογισμού ακέραιας δύναμης πραγματικού αριθμού

8.4 Λοιπές συναρτήσεις

Για την κωδικοποίηση περιορισμών πάνω σε αναδρομικούς όρους στον επιλυτή, ορίσαμε άλλες τρεις αναδρομικές συναρτήσεις SMTLIB.

8.4.1 Τύπος atom

Τα atoms στην Erlang αποτελούνται από κανέναν ή περισσότερους 8-bit Latin-1 (extended ASCII) χαρακτήρες [26], επομένως η απλούστερη αναπαράστασή τους σε SMTLIB είναι η λίστα ακεραίων. Κάθε στοιχείο της λίστας περιέχει τον κώδικα του αντίστοιχου χαρακτήρα. Για να είναι συνεπές το μοντέλο, η τιμή αυτή επιτρέπεται να κυμαίνεται στο εύρος από 0 μέχρι 255. Ο περιορισμός για έναν όρο atom με όνομα t εφαρμόζεται δίνοντας:

```
(assert (and (is-atom t) (ilist-spec (av t))))
```

όπου

```
(define-fun-rec ilist-spec ((l IList)) Bool (
     or
     (is-in l)
     (
          and
          (is-ic l)
          (<= 0 (ih l) 255)
          (ilist-spec (it l))
     )
))
```

εικόνα 22: Περιορισμός τύπου atom

Σημειώνουμε πως από την έκδοση Erlang/OTP 20.0 οι χαρακτήρες ενός atom είναι Unicode [27], συνεπώς έχουν ως κωδικό έναν ακέραιο που μπορεί να λαμβάνει μη αρνητικές τιμές, μικρότερες του φράγματος 1,114,112 (0x000000 – 0x10FFFF). Η τροποποίηση των προδιαγραφών της γλώσσας μπορεί πολύ εύκολα να ενσωματωθεί στο CutEr αλλάζοντας το άνω όριο του κριτηρίου σύγκρισης στην παραπάνω αναδρομική συνάρτηση.

8.4.2 Τύπος bitstring

Τα bitstings στην Erlang είναι ακολουθίες bits. Ο περιορισμός που τίθεται για το μήκος l της ακολουθίας έχει τη μορφή $l = m + n \cdot k, k \in \mathbb{N}$, όπου m και n σταθεροί, μη αρνητικοί ακέραιοι. Τότε, για έναν όρο t τύπου bitstring εξασφαλίζουμε ότι αποτελείται από τουλάχιστον m bits και ότι το υπόλοιπο τμήμα έχει μήκος πολλαπλάσιο του n. Η δεύτερη συνθήκη εκφράζεται με τη βοήθεια της ακόλουθης συνάρτησης:

```
(define-fun-rec slist-spec (
      (l SList)
      (n Int)
      (r Int)
) Bool (
     or
     (and (is-sn l) (= r 0))
     (
          and
          (is-sc l)
          (slist-spec (st l) n (- (ite (= r 0) n r) 1))
     )
))
```

εικόνα 23: Περιορισμός τύπου bitstring

Ας υποθέσουμε, για παράδειγμα, ότι $m = 3$ και $n = 4$. Τότε ο περιορισμός για τον όρο t τύπου bistring είναι:

```
(
     and
     (is-str t)
     (is-sc (sv t))
     (is-sc (st (sv t)))
     (is-sc (st (st (sv t))))
     (slist-spec (st (st (st (sv t)))) 4 0)
)
```

8.4.3 Τιμή συνάρτησης

Υποθέτουμε ότι έχουμε έναν όρο t, για τον οποίο θέλουμε να εκφράσουμε τον περιορισμό πως είναι συνάρτηση που απεικονίζει το ζεύγος ακεραίων $(9, 4)$ στην πραγματική τιμή 2.25. Το εύκολο μέρος είναι να θέσουμε περιορισμό ότι ο όρος t είναι συνάρτηση δύο μεταβλητών. Έπειτα μένει το πρώτο ζεύγος λίστας όρων με όρο του σώματος της συνάρτησης που σαν λίστα όρων έχει την $(9, 4)$ να έχει ως αντίστοιχη τιμή τον 2.25. Ο περιορισμός αυτός επιβάλλεται με την αναδρομική συνάρτηση flist-equals (εικόνα 24).

```
(define-fun-rec flist-equals (
     (f FList)
     (x TList)
     (y Term)
) Bool (
     or
     (and (is-fc f) (= (fx f) x) (= (fy f) y))
     (
          and
          (is-fc f)
          (not (= (fx f) x))
          (flist-equals (ft f) x y)
     )
))
```

εικόνα 24: Επιβολή τιμής συνάρτησης

Επομένως, σε SMTLIB γράφουμε:

```
(assert (
     and
     (is-fun t)
     (= (fa (fv t)) 2)
     (
          flist-equals
          (fm (fv t)) (tc (int 9) (tc (int 4) tn))
          (real 2.25)
     )
))
```

Η παραπάνω εντολή επιλύεται χωρίς πρόβλημα από τον CVC4. Όμως ο Z3 αδυνατεί να βρει αποτέλεσμα, οπότε στην υποκλάση του τροποποιήσαμε τις σχετιζόμενες μεθόδους, ώστε η αναζήτηση για ζεύγη λίστας όρων με όρο να πραγματοποιείται μέχρι ένα συγκεκριμένο βάθος. Κάθε φορά που προσθέτουμε περιορισμό τιμής συνάρτησης, η μεταβλητή του βάθους αυξάνεται κατά μία μονάδα. Με την συγκεκριμένη τεχνική ο επιλυτής δεν αναζητεί επ' άπειρον την τιμή για ταίριασμα της λίστας όρων μέσα στη λίστα των ζευγών. Η αναδρομική συνάρτηση πλέον καλείται με μία επιπλέον παράμετρο, αυτή του μέγιστου βάθους αναζήτησης (εικόνα 25).

```
(define-fun-rec flist-equals (
     (f FList)
     (x TList)
     (y Term)
     (d Int)
) Bool (
     or
     (and (>= d 0) (is-fc f) (= (fx f) x) (= (fy f) y))
     (
          and
          (> d 0)
          (is-fc f)
          (not (= (fx f) x))
          (flist-equals (ft f) x y (- d 1))
     )
))
```

εικόνα 25: Επιβολή τιμής συνάρτησης με μέγιστο βάθος αναζήτησης

Βιβλιογραφία

[1] *Satisfiability modulo theories --- Wikipedia, The Free Encyclopedia,* 2017.

[2] C. Barrett, P. Fontaine and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB),* 2016.

[3] A. Giantsios, N. Papaspyrou and K. Sagonas, "Concolic Testing for Functional Languages," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, New York, NY, USA, 2015.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, Berlin, 2011.

[5] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, 2008.

[6] C. Barrett, P. Fontaine and C. Tinelli, "The SMT-LIB Standard: Version 2.5," 2015.

[7] C. Barrett, P. Fontaine and C. Tinelli, "The SMT-LIB Standard: Version 2.6," 2017.

[8] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed Automated Random Testing," *SIGPLAN Not.,* vol. 40, pp. 213-223, 6 2005.

[9] M. Berkelaar, K. Eikland and P. Notebaert, *lpsolve 5.5, Open source (Mixed-Integer) Linear Programming system,* 2004.

[10] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proceedings of the 11th International Conference on Compiler Construction*, London, 2002.

[11] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Washington, 2008.

[12] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, 2014.

[13] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, T. Ball and R. B. Jones, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 419-423.

[14] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, Mississauga, 1999.

[15] H. Palikareva and C. Cadar, "Multi-solver Support in Symbolic Execution," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, N. Sharygina and H. Veith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 53-68.

[16] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification, 19th International Conference, {CAV} 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, 2007.

[17] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, Berlin, 2009.

[18] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *{Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)}*, Palo, 2004.

[19] *17.1. subprocess — Subprocess management — Python 2.7.14 documentation,* 2017.

[20] *Z3 - Guide,* 2017.

[21] A. Cimatti, A. Griggio, B. Schaafsma and R. Sebastiani, "The MathSAT5 SMT Solver," in *Proceedings of TACAS*, 2013.

[22] F. Corzilius, G. Kremer, S. Junges, S. Schupp and E. Ábrahám, "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving," in *Theory and Applications of Satisfiability Testing -- SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, M. Heule and S. Weaver, Eds., Cham, : Springer International Publishing, 2015, pp. 360-368.

[23] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe and P. Fontaine, "veriT: An Open, Trustable and Efficient SMT-Solver," in *Automated Deduction -- CADE-22: 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, R. A. Schmidt, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 151-156.

[24] P. Gustafsson and K. Sagonas, "Adaptive Pattern Matching on Binary Data," in *Programming Languages and Systems, 13th European Symposium on Programming, {ESOP} 2004, Held as Part of the Joint European Conferences on Theory and Practice*

of Software, {ETAPS} 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, 2004.

[25] P. Gustafsson and K. Sagonas, "Efficient manipulation of binary data using pattern matching," *J. Funct. Program.,* vol. 16, pp. 35-74, 2006.

[26] *Erlang -- External Term Format,* 2017.

[27] *Erlang/OTP 20.0 is released,* 2017.