Sinteza programa

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Anja Ivanišević, Ivan Ristović, Milana Kovačević, Vesna Katanić
kontakt email prvog, drugog (trećeg) autora

??. ?? 2018.

Abstract

U ovom tekstu je ukratko prikazana osnovna forma seminarskog rada. Obratite pažnju da je pored ove .pdf datoteke, u prilogu i odgovarajuća .tex datoteka, kao i .bib datoteka korišćena za generisanje literature. Na prvoj strani seminarskog rada su naslov, apstrakt i sadržaj, i to sve mora da stane na prvu stranu! Kako bi Vaš seminarski zadovoljio standarde i očekivanja, koristite uputstva i materijale sa predavanja na temu pisanja seminarskih radova. Ovo je samo šablon koji se odnosi na fizički izgled seminarskog rada (šablon koji *morate* da ispoštujete!) kao i par tehničkih pomoćnih uputstava. Molim Vas da kada budete predavali seminarski rad, imenujete datoteke tako da sadrže temu seminarskog rada, kao i imena i prezimena članova grupe (ili samo temu i prezimena, ukoliko je sa imenima predugačko). Predaja seminarskih radova biće isključivo preko web forme, a NE slanjem mejla.

Contents

1 Uvod **2**

2 Primene **2**

3 Izazovi **2**
3.1 Definisanje specifikacija 3
3.2 Prostor programa 3
3.2.1 Enumerativna pretraga 3
3.2.2 Deduktivna pretraga 4
3.2.3 Constraint Solving 4
3.2.4 Statistička pretraga 5

4 CEGIS **6**

5 Slike i tabele **6**

6 Zaključak **7**

Literatura **7**

A Dodatak **7**

1 Uvod

Uz sve novouvedene termine u zagradi naglasiti od koje engleske reči termin potiče. Naredni primeri ilustruju način uvođenja enlegskih termina kao i citiranje.

Teorema 1.1 *Problem zaustavljanja (eng.* halting problem*) je neodlučiv [4].*

Teorema 1.2 *Za prevođenje programa napisanih u programskom jeziku C može se koristiti GCC kompajler [1].*

Teorema 1.3 *Da bi se ispitivala ispravost softvera, najpre je potrebno precizno definisati njegovo ponašanje [3].*

Reference koje se koriste u ovom tekstu zadate su u datoteci *literature.bib*. Prevođenje u pdf format u Linux okruženju može se uraditi na sledeći način:

```
pdflatex TemaImePrezime.tex
bibtex TemaImePrezime.aux
pdflatex TemaImePrezime.tex
pdflatex TemaImePrezime.tex
```

Prvo latexovanje je neophodno da bi se generisao *.aux* fajl. *bibtex* proizvodi odgovarajući *.bbl* fajl koji se koristi za generisanje literature. Potrebna su dva prolaza (dva puta pdflatex) da bi se reference ubacile u tekst (tj da ne bi ostali znakovi pitanja umesto referenci). Dodavanjem novih referenci potrebno je ponoviti ceo postupak.

Broj naslova i podnaslova je proizvoljan. Neophodni su samo Uvod i Zaključak. Na poglavlja unutar teksta referisati se po potrebi.

Teorema 1.4 *U odeljku* **??** *precizirani su osnovni pojmovi, dok su zaključci dati u odeljku 6.*

Još jednom da napomenem da nema razloga da pišete:

```
\v{s} i \v{c} i \'c ...
```

Možete koristiti srpska slova

```
š i č i ć ...
```

Ovde pišem uvodni tekst. Ovde pišem uvodni tekst. Ovde pišem uvodni tekst. Ovde pišem uvodni tekst.

2 Primene

The synthesized program may be explicitly presented to the user for debugging, re-use, or for being incorporated as part of a larger workflow. However, in some cases, the synthesized program may be implicit and is simply used to automate the intended one-off task for the user, as in case of spreadsheet string transformations [43].

3 Izazovi

Pisanje programa koji može da sintetiše drugi program je veliki izazov. Naime, ovaj problem sinteze se može razložiti na dva potproblema:

- Definisanje specifikacija željenog programa,

- Pretraživanje prostora mogućih programa u potrazi za onim koji zadovoljava definisane specifikacije

Prostor programa se povećava eksponencijalno brzo, te postoje različite tehnike za njegovo pretraživanje.

3.1 Definisanje specifikacija

Generisani program treba da se ponaša na način koji to korisnik definiše. Međutim, precizno definisanje zahteva predstavlja velik izazov. Postoje različiti načini na koje se to može postići; od formalnih logičkih izraza do neformalnih opisa ili primera ulaza/izlaza programa.

Formalno definisanje zahteva često izgleda komplikovano (možda čak deluje i komplikovanije neko pisanje samog programa). Nasuprot tome, neformalne metode su mnogo prirodnije korisniku, ali dovode do drugih problema. Na primer, neka se željeni program definiše na osnovu primera njegovog ulaza i izlaza *"John Smith"* -¿ *"Smith, J."*. Ovaj program vrši nama intuitivnu transformaciju niski, ali da bi se on automatski generisao korišćenjem FlashFill [2] programa, on treba da pretraži prostor koji sadrži milione mogućih rešenja. Problem je u tome što programi nemaju ljudsku intuiciju, već se preprilagođavaju datim primerima ulaza i izlaza. Uz dodatnu komunikaciju sa korisnikom FlashFill usmerava pretragu te na kraju uspeva da nađe rešenje.

Većina programa koji se danas koriste su previše komplikovani da bi se potpuno opisali bilo formalnim bilo neformalnim metodama. Čak i ako bi se to nekako uspelo, opis programa bi bio toliko obiman kao i sama implementacija programa. Kako bi senteza ovakvih, realnih programa bila moguća, potrebno je omogućiti korisniku da ne definiše program do kraja, već da se interaktivno sa njim tokom sinteze postepeno dolazi do rešenja.

3.2 Prostor programa

Svaka uspešna sinteza programa vrši neki vid pretrage prostora mogućih programa. Ovo je težak kombinatorni problem. Broj mogućih rešenja raste eksponencijalno sa veličinom programa, te pretraga svih kandidata nije moguća u realnom vremenu. Potrebno je pažljivo vršiti odsecanja dela prostora pretrage kako bi se došlo do rešenja u realnom vremenu.

Tehnike pretrage se mogu zasnivati na enumerativnoj pretrazi, dedukciji, constraint solving, statističkim tehnikama, kao i na kombinaciji nekih od nih.

3.2.1 Enumerativna pretraga

An enumerative search technique enumerates programs in the underlying search space in some order and for each program checks whether or not it satisfies the synthesis constraints. While this might appear simple, it is often a very effective strategy. A naïve implementation of enumerative search often does not scale. Many practical systems that leverage enumerative search innovate by developing various optimizations for pruning the search space or by ordering it.

3.2.2 Deduktivna pretraga

Specifications on the formal end of this spectrum (traditionally required by deductive synthesis techniques) often appear to the user as complex as writing the program itself.

The deductive top-down search [113] follows the standard divide-and-conquer technique, where the key idea is to recursively reduce the problem of synthesizing a program expression e of a certain kind and that satisfies a certain specification fiii to simpler sub-problems (where the search is either over sub-expressions of e or over sub-specifications of fiii), followed by appropriately combining those results. The reduction logic for reducing a synthesis problem to simpler synthesis problems depends on the nature of the involved expression e and the inductive specification fiii. In particular, if e is of the form F(e1, e2), the reduction logic leverages the inverse semantics of F to push constraints on e down through the grammar into constraints on e1 and e2. While enumerative search is bottom-up (i.e., it enumerates smaller sub-expressions before enumerating larger expressions), the deductive search is top-down (i.e., it fixes the top-part of an expression and then searches for its sub-expressions). Enumerative search can be seen as finding a programmatic path (within an underlying grammar that connects inputs and outputs) starting from the inputs to outputs. Deduction does the same, but it searches for the programmatic path in a backward direction starting from the outputs leveraging the operator inverses. If the underlying grammar allows for a rich set of constants, the bottom-up enumerative search can get lost in simply guessing the right constants. On the other hand, the top-down deductive technique can deduce constants based on the accumulated constraints as the last step in the search process.

3.2.3 Constraint Solving

The constraint solving based techniques [132, 135] involve two main steps: constraint generation, and constraint resolution.

Constraint generation refers to the process of generating a logical constraint whose solution will yield the intended program. Generating such a logical constraint typically requires making some assumption about the control flow of the unknown program and encoding that control flow in some manner. Three different kinds of methods have been used in the past for constraint generation: invariant-based, path-based, and input-based. On one extreme, we have invariant-based methods that generate constraints that faithfully assert that the program satisfies the given specification [133]. Such methods also end up synthesizing an inductive proof of correctness in addition to the program itself. A disadvantage of such methods is that the generated constraints may be very sophisticated since the inductive invariants are often much more complicated and over a richer logic than the program itself. On the other extreme, we have inputbased methods that generate constraints that assert that the program satisfies the given specification on a certain collection of inputs [132]. Such constraints are usually much simpler in nature than the ones generated by the invariant-bases method. Unless paired with a sound counterexample guided inductive synthesis strategy (CEGIS), described in §3.2, this method trades off soundness for efficiency. A middle ground is achieved by path-based methods that generate constraints that assert that the program satisfies the given specification on all inputs that execute a certain set of paths [134]. Compared to input-based methods,

these methods may achieve a faster convergence, if paired up with an outer CEGIS loop. Constraint solving involves solving the constraints outputted by the constraint generation phase. These constraints often involve secondorder unknowns and universal quantifiers. A general strategy is to first reduce the second-order unknowns to first-order unknowns and then eliminate universal quantifiers, and then solve the resulting first-order quantifier-free constraints using an off-the-shelf SAT/SMT solver. The second-order unknowns are reduced to first-order unknowns by use of templates. The universal quantifiers can be eliminated using a variety of strategies including Farkas lemma, cover algorithms, and sampling.

3.2.4 Statistička pretraga

Postoji veliki broj statističkih metoda koje mogu da se upotrebe za pretragu. Neke od njih su:

- Mašinskog učenje Machine learning techniques can be used to augment other search methodologies based on enumerative search or deduction by providing likelihood of various choices at any choice point. One such choice point is selection of a production for a nonterminal in a grammar that specifies the underlying program space. The likelihood probabilities can be function of certain cues found in the input-ouput examples provided by the user or the additionally available inputs [89]. These functions are learned in an offline phase from training data.
- Genetičkog programiranje Genetic programming is a program synthesis method inspired by biological evolution [72]. It involves maintaining a population of individual programs, and using that to produce program variants by leveraging computational analogs of biological mutation and crossover. Mutation introduces random changes, while crossover facilitates sharing of useful pieces of code between programs being evolved. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. The success of a genetic programming based system crucially depends on the fitness function. Genetic programming has been used to discover mutual exclusion algorithms [68] and to fix bugs in imperative programs [146]
- MCMC sampling MCMC sampling has been used to search for a desired program starting from a given candidate. The success crucially depends on defining a smooth cost metric for Boolean constraints. STOKE [124], a superoptimization tool, uses Hamming distance to measure closeness of generated bit-values to the target on a representative test input set, and rewards generation of (almost) correct values in incorrect locations.
- Genetičkog programiranje Probabilistic inference has been used to evolve a given program by making local changes, one at a time. This relies on modeling a program as a graph of instructions and states, connected by constraint nodes. Each constraint node establishes the semantics of some instruction by relating the instruction with the state immediately before the instruction and the state immediately after the instruction [45]. Belief propagation has been used to synthesize imperative program fragments that execute polynomial computations and list manipulations [62].

4 CEGIS

What is CEGIS? Synthesis tasks often have the same structure: an implementation is sought that behaves correctly under all possible inputs (with the help of some extra variables, i.e. helper variables).

It is absolutely legal to pass such a term to an SMT solver like Z3. A big drawback is the universal quantifier though.

For many real world problems it is not necessary to consider all inputs to derive an implementation that behaves correctly for all of them. Following this observation, the problem was just moved to another position: which is the minimal subset of inputs one have to consider to ensure a correct synthesis?

This is the point where CEGIS comes into play. CEGIS is a loop looking for exactly this minimal subset of inputs and performing the implementation synthesis as a "by-product". Therefore CEGIS uses one satisfiability solver to generate new implementations based on all the inputs considered so far (starting with zero); and another one to generate counter examples that uncover incorrect behavior in the latest synthesised implementation. Eventually there will be no more implementations possible, i.e. the specification is not realisable, or no more counter examples possible, i.e. the latest implementation must be correct.

This CEGIS library works with the SMT solver Z3 and requires insight in the synthesis task to be executed, as it has to be specified which variables belong to implementation, inputs, etc. Boundary conditions are to be specified manually as well.

5 Slike i tabele

Slike i tabele treba da budu u svom okruženju, sa odgovarajućim naslovima, obeležene labelom da koje omogućava referenciranje.

Teorema 5.1 *Ovako se ubacuje slika. Obratiti pažnju da je dodato i* `\usepackage{graphicx}`

Figure 1: Pande

Na svaku sliku neophodno je referisati se negde u tekstu. Na primer, na slici 1 prikazane su pande.

Teorema 5.2 *I tabele treba da budu u svom okruženju, i na njih je neophodno referisati se u tekstu. Na primer, u tabeli 1 su prikazana različita poravnanja u tabelama.*

Table 1: Razlčita poravnanja u okviru iste tabele ne treba koristiti jer su nepregledna.

centralno poravnanje	levo poravnanje	desno poravnanje
a	b	c
d	e	f

6 Zaključak

Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak.

References

[1] Free Software Foundation. GNU gcc, 2013. on-line at: http://gcc.gnu.org/.

[2] Sumit Gulwani. *Automating string processing in spreadsheets using input-output examples*. Symposium on Principles of Programming Languages, POPL, Austin, TX, USA, 2011.

[3] J. Laski and W. Stanley. *Software Verification and Analysis*. Springer-Verlag, London, 2009.

[4] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe.