POLITECHNIKA POZNAŃSKA

Iwo Błądek

Machine Learning and Formal Verification for Acquisition of Knowledge in Heuristic Program Synthesis

Streszczenie rozprawy doktorskiej

Promotor: prof. dr hab. inż. Krzysztof Krawiec

Poznań, 2022

Tytuł w języku polskim:

Uczenie maszynowe i formalna weryfikacja dla pozyskiwania wiedzy w heurystycznej syntezie programów

1 Wprowadzenie

Automatyczna synteza programów komputerowych to problem polegający na znalezieniu programu komputerowego spełniającego pewien zestaw wymagań/ograniczeń, który nazywamy *specyfikacją*. Specyfikacja może przyjmować różne formy, przykładowo zbiór przykładów wejście-wyjście, formalne ograniczenia wyrażone w języku logiki, demonstracja etapów pośrednich wykonania programu, czy też opis zadania w języku naturalnym. Zbiór dopuszczalnych programów określony jest *językiem programowania*, który także jest, obok specyfikacji, częścią opisu danego problemu syntezy.

W literaturze można znaleźć wiele algorytmów rozwiązujących różne warianty problemu syntezy. Podejścia te możemy zaklasyfikować do czterech głównych paradygmatów [10]:

- *Metody pełnego przeglądu*, w których przestrzeń możliwych programów jest systematycznie przeszukiwana.
- *Podejścia oparte o rozwiązywanie ograniczeń*, w których problem syntezy zostaje przetransformowany do innego znanego problemu, na przykład spełnialności formuł logicznych (SAT), i rozwiązany za pomocą dedykowanych dla tego problemu metod.
- *Podejścia dedukcyjne*, w których specyfikacja jest stopniowo przekształcana do oczekiwanego programu, najczęściej za pomocą szeregu predefiniowanych reguł.
- *Techniki statystyczne i stochastyczne*, które oparte są na uczeniu maszynowym lub algorytmach metaheurystycznych, takich jak programowanie genetyczne.

Jako że przestrzeń przeszukiwania możliwych rozwiązań problemu syntezy jest bardzo duża i rośnie wykładniczo wraz z rosnącą długością programów, podejścia dokładne potrzebują dużo czasu by zwrócić rozwiązanie dla bardziej złożonych problemów. Alternatywnym podejściem są algorytmy heurystyczne, które co prawda nie gwarantują rozwiązania problemu syntezy ani znalezienia globalnie najlepszego programu w przypadku optymalizacyjnego wariantu tego problemu (w którym szukamy programu najlepszego według pewnej funkcji

celu), ale w zamian są w stanie zwrócić relatywnie szybko aproksymowane rozwiązanie. W rozprawie skupiamy się na *programowaniu genetycznym* (GP) [17, 22], które jest heurystycznym podejściem do syntezy w którym utrzymywana jest populacja programów wraz z presją selekcyjną na ich jakość, i poprzez mechanizm selekcji oraz operatory wariacji (mutacja, krzyżowanie) znajdowane są rozwiązania o coraz wyższej jakości.

Wspólną cechą wszystkich prac opisanych w niniejszej rozprawie jest wzbogacenie programowania genetycznego o dodatkową informację (wiedzę), która jest następnie wykorzystywana przez różne elementy algorytmu (selekcja, operatory wariacji) do zwiększenia efektywności przeszukiwania, albo wręcz umożliwienia rozwiązywania przez GP problemu syntezy dla danego rodzaju specyfikacji. W rozprawie przedstawione są następujące podejścia:

- *ewolucyjne szkicowanie programów* [5],
- *programowanie genetyczne kierowane kontrprzykładami* [7, 18, 19],
- *regresja symboliczna kierowana kontrprzykładami* [6],
- *programowanie genetyczne wspierane sztuczną siecią neuronową* [20].

2 Ewolucyjne szkicowanie programów

Ewolucyjne szkicowanie programów (ang. Evolutionary Program Sketching, EPS) [5] bazuje na paradygmacie *syntezy programów przez szkicowanie* [24, 25], w którym algorytm syntezy oprócz specyfikacji działania dostaje również na wejście częściowo napisany program („szkic") wraz ze wskazanymi miejscami ("luki"), które powinny zostać wypełnione brakującymi fragmentami programu. Szkicowanie pozwala uniknąć pełnego przeszukiwania ogromnej przestrzeni rozwiązań, a jednocześnie naturalnie wpisuje się w praktyczne zastosowania syntezy programów w sytuacji, kiedy programista/użytkownik ma dość dobre pojęcie jak w ogólności powinna wyglądać struktura programu, ale chce skorzystać z algorytmu syntezy żeby uzupełnić, często wymagające dłuższej analizy, detale.

EPS zwalnia użytkownika z obowiązku przygotowania szkicu, ponieważ są one generowane automatycznie na drodze ewolucji, a następnie uzupełniane przez metodę syntezy opartą o rozwiązywanie ograniczeń. Problem syntezy sprowadzany jest do problemu spełnialności formuł logicznych modulo teorie (ang. Satisfiability Modulo Theories, SMT) [3, 9]. Specyfikacją działania programu są przykłady wejście-wyjście, a zadaniem jest znalezienie programu spełniającego jak najwięcej z nich. Do rozwiązania tego zadania optymalizacji wykorzystany został solwer Z3 [8], który udostępnia narzędzie służące do optymalizacji bazujące na SMT [4].

EPS działa na podobnej zasadzie jak algorytmy memetyczne [21], to znaczy rozwiązania znalezione przez algorytm ewolucyjny są następnie oceniane i potencjalnie modyfikowane na podstawie tego, jak dobrze solwer SMT dał radę je uzupełnić. Zaproponowano dwa warianty podejścia: w wariancie Baldwinowskim (EPS-B) ma miejsce jedynie przypisywanie do danego rozwiązania (szkicu) wartości przystosowania jego uzupełnionej przez solwer SMT wersji. Z kolei w wariancie Lamarckowskim (EPS-L) rozwiązanie jest także zastępowane w populacji przez uzupełniony szkic. Nowe luki dodawane są do programów przez operatory mutacji i inicjalizacji, które traktują luki jako element terminalny

gramatyki języka programowania o przypisanym typie wskazującym na rodzaj wartości zwracanej przez brakujący fragment kodu.

Przeprowadzono wstępne eksperymenty obliczeniowe, w których zostały ze sobą porównane EPS-B i EPS-L w różnych konfiguracjach, a także sprawdzono czy radzą sobie one lepiej niż samo GP bez mechanizmu szkicowania. Wyniki pokazały, że szkicowanie rzeczywiście daje lepsze wyniki niż standardowe GP, o ile luki są wypełniane przez stałe a nie tylko zmienne wejściowe programu. Oba warianty EPS okazały się znacząco dominować nad GP przy tej samej liczbie pokoleń, przy czym wariant Baldwinowski okazał się zdecydowanie lepszy od Lamarckowskiego. Żeby skompensować długi rzeczywisty czas działania EPS, przetestowane zostały również dwie dodatkowe konfiguracje w których GP miało do dyspozycji taki sam budżet czasu obliczeniowego oraz znacznie większą populację – po tych zmianach EPS-B pozostał zdecydowanie dominującym wariantem jeżeli chodzi o skuteczność, jednak EPS-L okazał się być w tych warunkach gorszym algorytmem niż GP.

3 Programowanie genetyczne kierowane kontrprzykładami

Programowanie genetyczne kierowane kontrprzykładami (ang. Counterexample-Driven Genetic Programming, CDGP) [7, 18, 19] jest próbą zastosowania GP do problemów syntezy programów, w których zadanie jest określone wyłącznie przez formalną specyfikację złożoną z logicznych ograniczeń wyrażonych w logice pierwszego rzędu rozszerzonej teoriami. Teorie pozwalają wzbogacić semantykę wyrażeń logicznych i znacząco ułatwić przygotowywanie formalnych specyfikacji; w tej pracy skupiliśmy się na dwóch z nich: liniowej arytmetyce liczb całkowitych (Linear Integer Arithmetic, LIA), oraz operacjach na ciągach znaków i liczbach (Strings with Linear Integer Arithmetic, SLIA).

Problem z bezpośrednim użyciem GP do zadań syntezy na podstawie formalnej specyfikacji polega na trudności w skonstruowaniu odpowiedniej funkcji celu, gdyż GP realizuje zadanie optymalizacji, podczas gdy synteza na podstawie formalnej specyfikacji to problem przeszukiwania. Problem ten próbowano rozwiązać poprzez zliczanie spełnionych indywidualnych ograniczeń [11, 12], wyróżnianie poziomu spełnienia danego ograniczenia [13, 14], czy też wykorzystanie kontrprzykładów pochodzących z nieudanych weryfikacji [15, 16].

CDGP również jest oparte o zbieranie kontrprzykładów, które po przekształceniu do regularnych testów wykorzystywane są do obliczania miary przystosowania programów w populacji. W odróżnieniu od wcześniej opisanych prac, wykorzystujemy to podejście do rozwiązywania bardziej złożonych problemów i wprowadzamy dodatkowy warunek który musi spełnić program zanim zostanie poddany weryfikacji. Początkowo zbiór testów w CDGP jest pusty, i może ulec powiększeniu gdy program w populacji spełni procent α (parametr algorytmu) już zebranych testów i zostanie poddane formalnej weryfikacji przy użyciu solwera SMT. Celem weryfikacji jest formalne udowodnienie poprawności programu dla każdego wejścia; proces ten przeprowadzany jest z pomocą solwera SMT, który pozwala uniknąć testowania programu na wszystkich możliwych wejściach. Jeżeli weryfikacja zakończy się sukcesem, to CDGP kończy działanie, ponieważ program spełniający formalną specyfikację został znaleziony. W przeciwnym wypadku, tworzony jest nowy test

na bazie kontrprzykładu zwróconego przez solwer SMT i dodawany jest do zbioru testów. Zastosowany tutaj mechanizm pozwala ograniczyć liczbę kosztownych weryfikacji kiedy posiadamy informację, że program jest niepoprawny (nie spełnia wszystkich testów). Z drugiej strony niższe wartości parametru α pozwalają uzyskać więcej testów i dostarczać tym samym więcej informacji ewolucyjnemu algorytmowi przeszukiwania.

Eksperymenty obliczeniowe wykazały, że w dziedzinie LIA CDGP działa gorzej niż formalne metody syntezy z którymi go porównywaliśmy (pełnoprzeglądowy EUSolver [1], oraz oparty o rozwiązywanie ograniczeń CVC4 [23]) – osiąga gorsze rezultaty w znacznie gorszym czasie. Jednak LIA to relatywnie prosta teoria, dla znacznie trudniejszej SLIA to CDGP okazało się uzyskiwać lepsze rezultaty. Programy znalezione przez CDGP były też często, dla obu problemów, wielokrotnie krótsze niż te znalezione przez metody dokładne. Inne wnioski z eksperymentów obliczeniowych to bardzo wysoka skuteczność algorytmu selekcji lexicase [26] w porównaniu do selekcji turniejowej, lekka przewaga wartości $\alpha = 0.75$ nad innymi testowanymi wartościami, a także obserwacja, że kontrprzykłady generowane przez solwer SMT pozwalają uzyskać lepsze rezultaty niż te generowane losowo.

4 Regresja symboliczna kierowana kontrprzykładami

Regresja symboliczna kierowana kontrprzykładami (ang. Counterexample-Driven Symbolic Regression, CDSR) [6] to efekt adaptacji CDGP do rozwiązywania problemów regresji symbolicznej. Regresja symboliczna polega na znalezieniu wyrażenia matematycznego, które możliwie dobrze wyjaśni zbiór przykładów wejście-wyjście (zbiór uczący). Jest to problem z dziedziny uczenia maszynowego, ponieważ oczekujemy że zaproponowana formuła będzie również trafnie przewidywać wartości dla przykładów spoza, potencjalnie obarczonego szumem i błędami (np. pomiarowymi), zbioru uczącego. W rozważanym przez nas scenariuszu, poza zbiorem testów użytkownik dostarcza także zbiór ograniczeń logicznych (przykładowo, wymagając od formuły symetrii albo monotoniczności), i tak zdefiniowane ogólniejsze zadanie nazywamy *regresją symboliczną z formalnymi ograniczeniami* (Symbolic Regression with Formal Constraints, SRFC) [6].

W porównaniu do CDGP, w CDSR zmienione zostały głównie dwa elementy. Po pierwsze, ze zbioru uczącego wydzielony został zbiór walidacyjny, który zapobiega przeuczeniu i kończy działanie algorytmu kiedy błąd na zbiorze uczącym nie ulega poprawie przez pewną liczbę pokoleń. Po drugie, musieliśmy przyjąć próg błędu (domyślnie 5% odchyłu od oczekiwanego wyjścia) przy którym uznajemy testy za spełnione dla kryterium weryfikacji opartym na parametrze α. Opracowaliśmy również CDSR_p, wariant CDSR w którym spełnienie indywidualnych ograniczeń jest uwzględnione w wektorze przystosowania rozwiązań, dzięki czemu spełnianie ograniczeń ma bezpośredni wpływ na presję selekcyjną.

Spośród testowanych wariantów CDSR, w eksperymentach CDSR_p okazał się zdecydowanie najlepszy jeżeli chodzi o spełnianie ograniczeń, podczas gdy pod względem błędu średniokwadratowego na zbiorze testowym nieco lepszy okazał się standardowy wariant CDSR. Przeprowadziliśmy również kompleksowe porównanie CDSR z szeregiem klasycznych algorytmów uczenia maszynowego dla problemu regresji. Algorytmy te nie przyjmują formalnych ograniczeń jako danych uczących, i celem eksperymentu było sprawdzenie, czy pomimo to potrafią te ograniczenia spełnić poprzez naukę na przykładach. Nieco wbrew

naszym oczekiwaniom okazało się, że najlepsze algorytmy regresji spełniają średnio więcej ograniczeń niż CDSR_p. CDSR_p jednak wyraźnie częściej daje radę spełnić wszystkie ograniczenia na raz oraz jest znacznie skuteczniejszy w spełnianiu pewnych rodzajów ograniczeń. Z kolei standardowy wariant CDSR zdołał uzyskać mniejszy błąd na zbiorze testowym.

5 Programowanie genetyczne wspierane sztuczną siecią neuronową

Prace nad programowaniem genetycznym wspieranym sztuczną siecią neuronową (ang. Neuro-Guided Genetic Programing) [20] były zainspirowane podejściem DeepCoder [2], w którym problem syntezy programów rozwiązywany jest w dwóch fazach. W pierwszej fazie, model uczenia maszynowego (sztuczna sieć neuronowa) uczony jest rozkładu prawdopodobieństwa wystąpienia instrukcji w programie pod warunkiem przykładów wejście-wyjście. Co istotne, proces uczenia przeprowadzany jest tylko raz dla wybranej domeny problemów. W drugiej fazie, nauczony model jest wykorzystywany jako wsparcie dla zewnętrznego algorytmu przeszukiwania potrafiącego wykorzystać predykcje sieci neuronowej przy rozwiązywaniu konkretnych problemów syntezy (potencjalnie innych, niż na których sieć była uczona w pierwszej fazie). Przykładowo, takim algorytmem może być zmodyfikowany wariant DFS (ang. depth-first search), który konstruuje drzewa reprezentujące programy metodą priorytetyzowanego przeglądu, dając pierwszeństwo przeszukiwaniu gałęzi odpowiadających bardziej prawdopodobnym instrukcjom według wskazań sieci neuronowej.

Nasz przyczynek w zakresie tego podejścia do syntezy programów miała charakter eksperymentalny, i polegała na jego zastosowaniu razem z GP oraz przetestowaniu, jak parametryzacja GP wpływać będzie na skuteczność tego podejścia. Eksperymenty przeprowadzone zostały przy pomocy naszej własnej implementacji sieci neuronowej oraz modułu generującego dane uczące, bazujących jednak mocno na tych zastosowanych w DeepCoder [2]. Wykazały one, że jest istotna poprawa efektywności dla GP wspieranego przez sieć neuronową w porównaniu do wariantów bez takiego wsparcia lub z prostym obciążeniem faworyzującym instrukcje najczęściej występujące w zbiorze uczącym. Zaobserowano także, że zastosowanie predykcji sieci także podczas inicjalizacji GP daje bardzo wyraźną poprawę w porównaniu do używania predykcji sieci wyłącznie podczas mutacji.

6 Podsumowanie

Niniejsza rozprawa proponuje cztery heurystyczne podejścia do syntezy programów oparte na GP i zdobywające dodatkową wiedzę o zadaniu syntezy za pomocą formalnej weryfikacji/optymalizacji przy użyciu solwera SMT lub uczenia maszynowego. Najważniejsze rezultaty rozprawy to:

- Ewolucyjne podejście do syntezy programów przez szkicowanie, w którym szkice są generowane automatycznie przez GP.

- Podejście do syntezy programów na podstawie formalnej specyfikacji przy użyciu GP, oparte na kontrprzykładach uzyskanych z formalnej weryfikacji niepoprawnych programów.
- Definicja zadania regresji symbolicznej z formalnymi ograniczeniami.
- Opracowanie algorytmu wykorzystującego GP do rozwiązywania tego zadania, i dogłębne porównanie go z klasycznymi algorytmami uczenia maszynowego dla problemu regresji.
- Implementacja i przetestowanie podejścia do rozwiązywania problemu syntezy użytego w DeepCoder [2] w reżimie obliczeń ewolucyjnych.

Bibliografia

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 319–336, 2017.

[2] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *Proceedings International Conference on Learning Representations 2017*. OpenReviews.net, April 2017.

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In C. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications*, chapter 12, pages 825–885. IOS Press, 2009.

[4] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[5] Iwo Błądek and Krzysztof Krawiec. Evolutionary Program Sketching. In Mauro Castelli, James McDermott, and Lukas Sekanina, editors, *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 3–18, Amsterdam, 19-21 April 2017. Springer Verlag.

[6] Iwo Błądek and Krzysztof Krawiec. Solving Symbolic Regression Problems with Formal Constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 977–984, New York, NY, USA, 2019. ACM.

[7] Iwo Błądek, Krzysztof Krawiec, and Jerry Swan. Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications. *Evolutionary Computation*, 26(3):441–469, Fall 2018.

[8] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and*

Analysis of Systems, volume 4963 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[9] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, September 2011.

[10] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[11] Pei He, Lishan Kang, Colin G. Johnson, and Shi Ying. Hoare logic-based genetic programming. *SCIENCE CHINA Information Sciences*, 54(3):623–637, March 2011.

[12] Colin Johnson. Genetic Programming with Fitness based on Model Checking. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124, Valencia, Spain, 11-13 April 2007. Springer.

[13] Gal Katz and Doron Peled. *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*, pages 33–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[14] Gal Katz and Doron Peled. MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *8th International Symposium on Automated Technology for Verification and Analysis, ATVA 2010*, volume 6252 of *Lecture Notes in Computer Science*, pages 359–364, Singapore, September 21-24 2010. Springer.

[15] Gal Katz and Doron Peled. Synthesis of Parametric Programs using Genetic Programming and Model Checking. In Lukas Holik and Lorenzo Clemente, editors, Proceedings 15th International Workshop on *Verification of Infinite-State Systems,* Hanoi, Vietnam, 14th October 2013, volume 140 of *Electronic Proceedings in Theoretical Computer Science*, pages 70–84. Open Publishing Association, 2014.

[16] Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer*, 19:449–464, 2016.

[17] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[18] Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. Counterexample-Driven Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 953–960, New York, NY, USA, 2017. ACM.

[19] Krzysztof Krawiec, Iwo Błądek, Jerry Swan, and John H. Drake. Counterexample-Driven Genetic Programming: Stochastic Synthesis of Provably Correct Programs. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5304–5308. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[20] Paweł Liskowski, Iwo Błądek, and Krzysztof Krawiec. Neuro-Guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1143–1150, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Pablo Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms. Technical report, California Institute of Technology, 1989.

[22] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Published via `http://lulu.com`, 2008.

[23] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-Based Synthesis in SMT. *Formal Methods in System Design*, Feb 2017.

[24] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008.

[25] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, San Jose, CA, USA, 2006.

[26] Lee Spector. Assessment of Problem Modality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, page 401–408, New York, NY, USA, 2012. Association for Computing Machinery.