AFRL-RI-RS-TR-2019-069

CLIO: A DIGITAL CODE ASSISTANT FOR THE BIG CODE ERA

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MARCH 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-069 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. MCKEEVER
Work Unit Manager

/ S /
QING WU
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2019	FINAL TECHNICAL REPORT	SEP 2014 – SEP 2018

4. TITLE AND SUBTITLE

CLIO: A DIGITAL CODE ASSISTANT FOR THE BIG CODE ERA

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-14-2-0242

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Armando Solar-Lezama

5d. PROJECT NUMBER
MUSE

5e. TASK NUMBER
BM

5f. WORK UNIT NUMBER
IT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
77 Massachusetts Ave
Cambridge MA 02139-4301

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-069

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report summarizes the results of Project Clio. The goal of the Clio effort was to develop technologies that could help reduce the high cost of software by bringing more automation into development and maintenance tasks. Consistent with the overall goals of the Defense Advanced Research Program Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) program, a central feature of our effort was to leverage data about existing software in order to enable capabilities that were previously not available.

15. SUBJECT TERMS
Program Synthesis, Program Repair, Big Code

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM E. MCKEEVER**
U	U	U	UU	23	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

Summary

This report summarizes the results of Project Clio. The goal of the Clio effort was to develop technologies that could help reduce the high cost of software by bringing more automation into development and maintenance tasks. Consistent with the overall goals of the Defense Advanced Research Program Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) program, a central feature of our effort was to leverage data about existing software in order to enable capabilities that were previously not available.

The project was divided into a Maintenance thrust focused primarily on program repair, and a Development thrust, focused primarily on program synthesis. Among the key accomplishments for the project are the following:

- Dramatic improvements in automatic patch generation by leveraging big-code.
- First-time demonstration of "organ transplants for code." The ability to transfer functionality automatically from one application to another.
- First-time demonstration of the ability to discover how to use an Application Programming Interface (API) by exercising its functionality on an existing application.
- First-time demonstration of the automatic generation of key components of a satisfiability modulo theories (SMT) solver.
- Dramatic improvements in the ability to generate data processing pipelines for machine learning.
- First-time demonstration of the ability to learn a domain-specific language from a corpus of synthesis problems.
- Demonstration of the ability to generate code from hand-drawn diagrams.

Intro

The goal of project Clio was to develop technologies that could leverage data about existing source code and related artifacts available from the internet in order to help automate development and maintenance tasks. As part of this project, we explored a number of different approaches for learning from a corpus of software artifacts in order to support maintenance tasks such as bug repair and development tasks involving both the development of brand new algorithms, as well as the implementation of complex functionality by using existing APIs.

Each of the thrusts described in this report illustrates a different way of leveraging the data from the corpus, but one element all the projects have in common is the interaction between Machine Learning-based techniques and symbolic techniques for the analysis, manipulation and search of programs from the Programming Systems community. It is this combination of techniques from the two fields that enables the technological accomplishments outlined in this report.

Methods, Assumptions and Procedures

The two main thrusts of the project were Maintenance and Development; both thrusts shared the common philosophy of leveraging data about both the code and its execution in order to deliver automation, but we now elaborate on the detailed methods assumptions and procedures for each separately.

Maintenance

For maintenance, we focused on two main tasks: patch transfer and patch discovery.

Patch Transfer with Code Phage and CarbonCopy

Our code transfer research exploited the availability of large amounts of code to find useful code in one donor application and transfer it into another recipient application to eliminate security vulnerabilities, fix defects, are augment the functionality of the recipient application. For example, there may be two different Portable Document Format (PDF) readers that have been patched by different vendors, but only one is vulnerable to an overflow given a particularly malformed file; the other one simply rejects it with an error message. The goal of patch transfer is to identify the code in the correct PDF reader that identifies that the file is malformed and transfer it to the vulnerable reader which must then be able to reject the malformed input as well, essentially to perform an organ transplant on code.

As part of the project we tested two versions of this technology. The first developed into a prototype called Code Phage (CP), a system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Experimental results using seven donor applications to eliminate ten errors in seven recipient applications highlight the ability of CP to transfer code across applications to eliminate out of bounds access, integer overflow, and divide by zero

errors. Because CP works with binary donors with no need for source code or symbolic information, it supports a wide range of use cases. To the best of our knowledge, CP was the first system to automatically transfer code across multiple applications. The results were published in Programming Language Design and Implementation (PLDI) 2015 [1].

The second version of the technology was implemented in a prototype named CodeCarbonCopy (CCC), a system for transferring code from a donor application into a recipient application. CCC starts with functionality identified by the developer to transfer into an insertion point (again identified by the developer) in the recipient. CCC uses paired executions of the donor and recipient on the same input file to obtain a translation between the data representation and name space of the recipient and the data representation and name space of the donor. It also implements a static analysis that identifies and removes irrelevant functionality useful in the donor but not in the recipient. We evaluated CCC on eight transfers between six applications. Our results show that CCC can successfully transfer donor functionality into recipient applications, including the ability to transfer new image filtering kernels and work with new input formats. The results of this work were published in Future of Software Engineering (FSE) in 2017 [2].

Patch Generation with Prophet and Genesis

Our automatic patch generation research exploited the availability of large amounts of code to automatically derive models of correct patches that could be used to distinguish correct patches from those that merely masked the symptoms of a bug.

We developed Prophet, a novel patch generation system that works with a set of successful human patches obtained from open-source software repositories to learn a probabilistic, application-independent model of correct code. Prophet generates a space of candidate patches, uses the model to rank the candidate patches in order of likely correctness, and validates the ranked patches against a suite of test cases to find correct patches. Experimental results show that, on a benchmark set of 69 real-world defects drawn from eight open-source projects, Prophet significantly outperforms the previous state-of-the-art patch generation systems. More details can be found in our Principles of Programming Languages (POPL) 2016 paper [3].

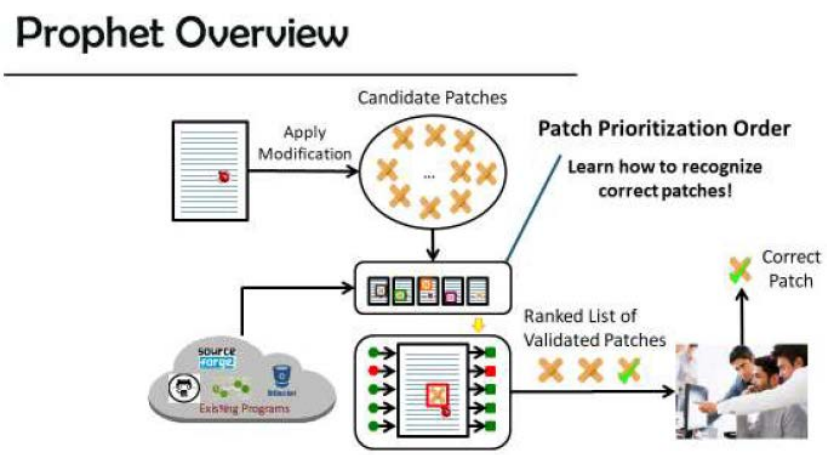

Figure 1 Overview of the Prophet System

We improved upon the prophet system with a follow-up system called Genesis. Unlike Prophet, Genesis does not rely on a set of human-crafted rules for how to transform the program. Instead, Genesis processes human patches from the corpus to automatically infer code transforms for automatic patch generation. We obtained results that characterize the effectiveness of the Genesis inference algorithms and the complete Genesis patch generation system working with real-world patches and defects collected from 372 Java projects. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from previous successful patches. More details are available in our FSE 17 paper [4].

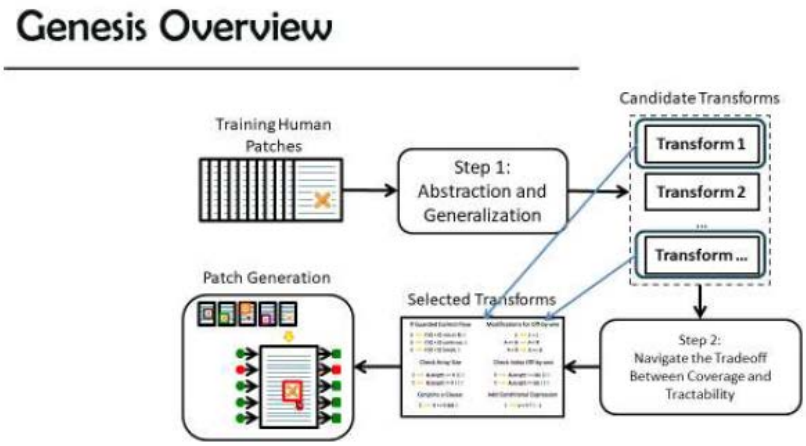

Figure 2 Overview of the Genesis System

Development

For development tasks, we developed several foundational technologies for leveraging data in order to support program synthesis.

DemoMatch: Learning to use APIs with execution data

The goal of the DemoMatch system was to allow developers to learn how to use an API when they did not know anything about its internal structure, not even the names of concepts used by the API. The starting point for DemoMatch was a database of program traces called DeLight, which captures detailed information about the execution of the API and its interaction with an application. During the setup phase, a database is constructed by running a set of applications that use the API while they are monitored by our trace capture infrastructure.

The DemoMatch tool then allows a programmer who wants to use an API to achieve some functionality to simply demonstrate that functionality in an existing application. When the functionality is demonstrated, DemoMatch will capture a short trace of the execution of the application during the demonstration that it will use to infer the functionality that the user intends to demonstrate. The short trace does not contain enough information to know what API commands had to be issued, because the short traces usually fail to capture the setup phase that enabled the behavior demonstrated by the user. Instead, the short traces are used to index into the database to find traces where similar behaviors were exercised. Unlike the short demonstration trace, the traces in the database include all the setup steps; so after some analysis, DemoMatch can return to the user with the key elements of the code that needs to be written in order to achieve the demonstrated functionality. The details of the process are explained in our Programming Language Design and Implementation (PLDI) 2017 paper [5].

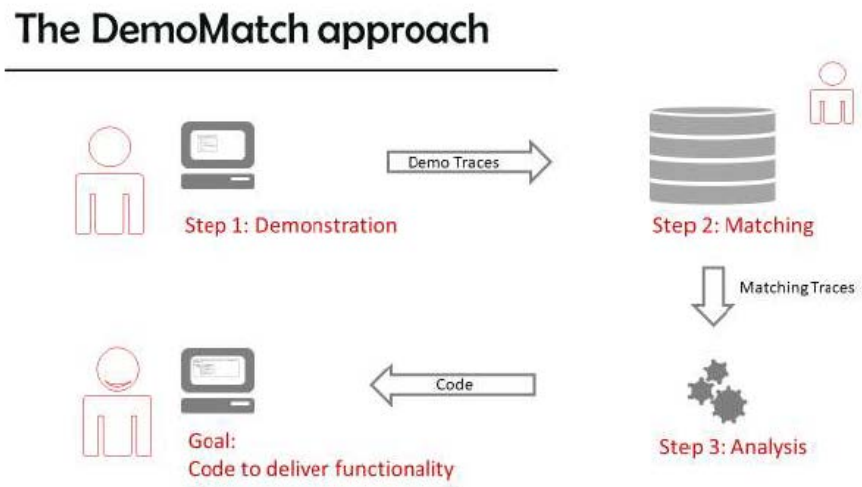

Figure 3 Overview of the DEMOMATCH Approach

Data Enhanced Sketch and Constraint Solving

Another approach we explored under this effort is the use of data to improve constraint-based synthesis. The standard approach to constraint-based synthesis is illustrated in Figure 4. The programmer writes a specification and a sketch of the desired solution to define a synthesis problem, and the problem definition is translated to a series of constraints that are solved by an SMT solver. The result of solving the constraints is then mapped back to a program that satisfies the specification.

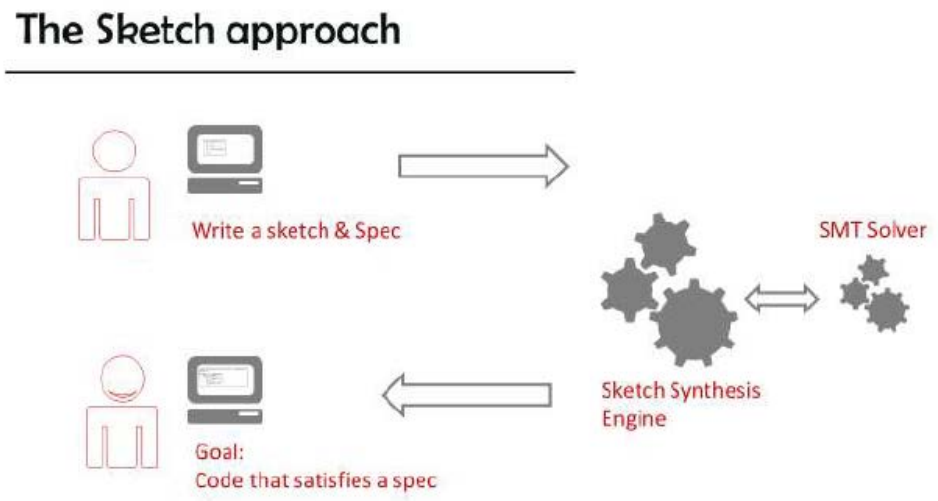

Figure 4 The Sketch Approach

A key element in this approach is the SMT solver. Our approach was to use a corpus of related problems to automatically synthesize a specialized version of the SMT solver that would be faster than the original general purpose solver. An interesting aspect of this approach was that it involved using the synthesizer to turn the data into a new synthesizer, so the synthesizer was essentially learning to improve itself.

We focused on two aspects of SMT solvers. The first aspect, illustrated by Figure 5 below was to automate the translation from high-level bit-vector constraints to CNF clauses. Given a large formula, there are many different ways of partitioning it into chunks and generating Conjunctive Normal Form (CNF) clauses for each chunk, with different solvers using different heuristics for this. Given a corpus of problems, the synthesizer will automatically generate a good way of partitioning the formula and of generating clauses for each partition. The results of this effort were published in a paper in the International Conferences on Theory and Applications of Satisfiability Testing (SAT) 2016 [6].

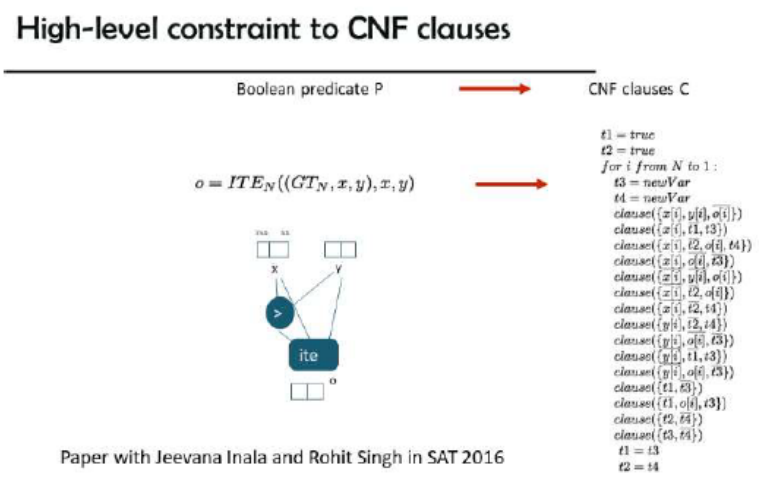

Figure 5 Bit-Vector Constraints in an SMT Formula must be translated to CNF Clauses

The second aspect of interest of the SMT solver is the simplification engine which uses conditional rewrite rules in order to attempt to simplify the problem. These rules describe how to simplify certain patterns assuming the solver can prove that some conditions are satisfied. Figure 6 illustrates one such rule and the C++ code that the solver uses to implement it. For this part of the project, we developed a tool called Swapper that given a corpus of problems, automatically generates conditional rewrite rules that are useful for that corpus and proves them correct. The details of this work were published in Formal Methods in Computer-Aided Design (FMCAD) 2016 [7].

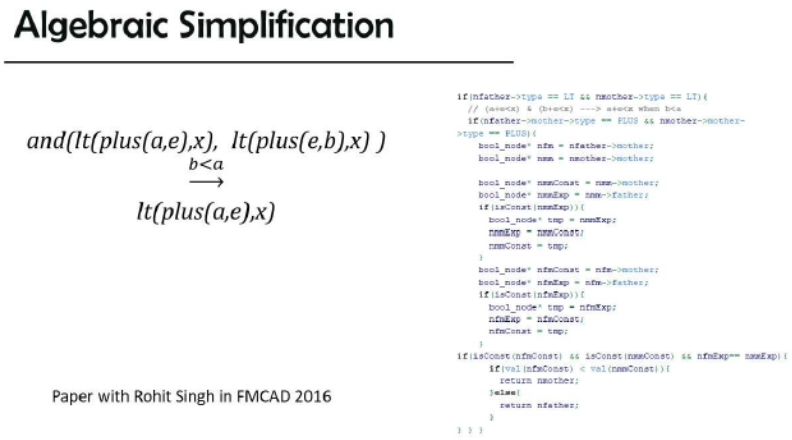

Figure 6 Conditional Rewrite Rule and the C++ Code Needed to Implement it in the Solver

Both of these elements enabled significant performance improvements in the solver.

Automatically Synthesizing Learning Pipelines

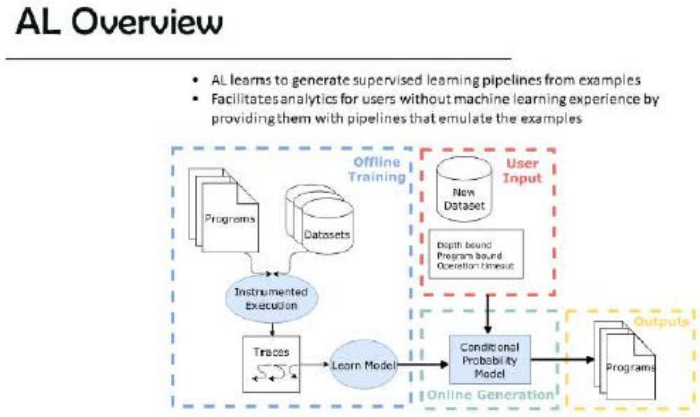

Figure 7 Overview of the Automatically Synthesizing Learning (AL) System for Learning to Generate New Supervised Learning Pipelines

We developed a system to automatically generate supervised learning pipelines from a set of APIs by learning from existing examples.

Our system uses a probabilistic model of pipeline likelihood to guide the generation process. This model is trained on prior example pipelines and their input data, characterizing the likelihood of each step in a pipeline given previous steps. For our experiments, we instrument and learn from 500 different Python data science programs that use 9 publicly available datasets. Each of these programs implements a supervised learning pipeline as part of its functionality. We evaluate our tool by generating pipelines for 25 previously unseen datasets collected from various repositories (OpenML, Kaggle, Scikit-Learn, UCI). Our evaluation showed that our system can produce pipelines in under 5 minutes that are comparable to those produced by other systems in 1 hour. Our tool is also able to generalize to more datasets, which require additional transformations of the input for correct execution [8].

Learning a Synthesizer

In recent years, many important synthesis problems have been solved not by using a general synthesis infrastructure such as Sketch, but rather by building a custom synthesizer, which searches programs in a Domain-Specific Language (DSL) using a specialized search heuristic, so the question for this project was whether we could learn a DSL and a search heuristic from a corpus. This was done with a new algorithm called Explore-Compress-Compile (EC^2). The algorithm works by iterating through the three phases mentioned in the name. The algorithm is seeded with a very basic language with only a small number of constructs. During the Exploration phase, the algorithm uses its current version of the language and the search heuristic to try to solve as many problems from the corpus as it can; during early iterations, it will succeed in solving only a few problems, but as it learns a more sophisticated language with a more sophisticated policy, its capabilities increase. From Exploration, the algorithm moves to Compression, where the algorithm analyzes all the solutions generated during Exploration and

infers new DSL components that would have made those solutions more compact. Finally, during the Compilation phase, the algorithm "dreams" new programs based on the new components it derived and uses these "dreamed" programs to train a new exploration policy. The algorithm then cycles back to the Exploration phase armed with the improved policy and improved DSL, allowing it to solve new problems in the corpus. This work was presented at Neural Information Processing Systems (NeurIPS) 2018 [9].

From Hand Drawings to Graphics Programs

As part of this work, we developed a system that is able to take as input hand-drawn images and produce a tikz program that when executed generates the desired image, but which also captures all the symmetries and regularity of the drawing. The system works through a two-stage pipeline: during the first stage, a neural network translates the hand-drawn images into a specification of the diagram to be generated, and then in the second phase, a program synthesizer generates the code with the goal of minimizing the program complexity. The high-level process is illustrated in Figure 8. This work was presented at NeurIPS 2018 [10]. Additionally, this work led to a new sponsored collaboration with Siemens and follow-on work expanding the approach to 3D, which was recently published in the Conference and Exhibition on Computer Graphics and Interactive Techniques SIGGRAPH ASIA [11].

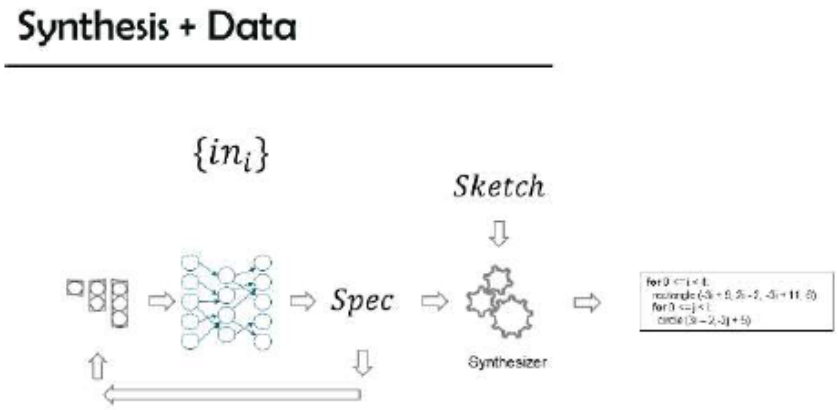

Figure 8 General Approach for Converting Hand-Drawn Figures into Programs

Selecting representative examples for synthesis

The high-level observation behind this effort was that for many synthesis problems, the goal is to learn a function from a large number of input-output examples, but in some cases, there are significantly more input-output examples than are necessary to fully specify the task, making the synthesis process unnecessarily slow. We were able to show that given a corpus of related synthesis problems, you could train a neural network to identify whether an additional example will add any information to the problem, taking into account the biases built in to the synthesizer. By using this technique, the neural network can be used to pre-filter the set of examples to a much smaller set that leads to a simpler synthesis problem that is much easier to

solve. The results of this work were published in the International Conference on Machine Learning (ICML) 2018 [12].

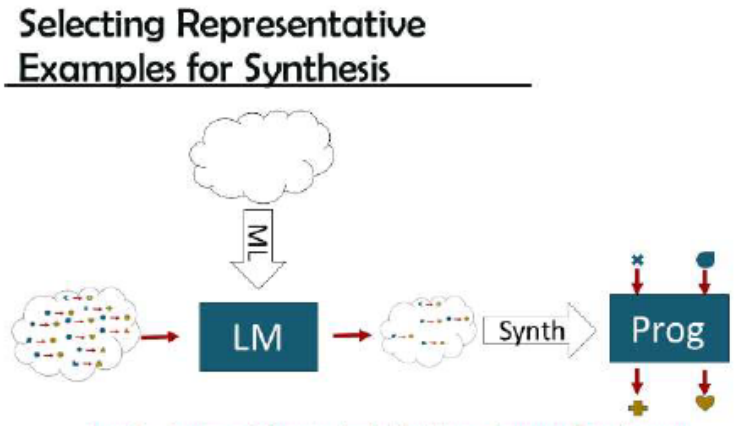

Figure 9 Overview of Selecting Representative Examples for Synthesis

Results and Discussion

We now elaborate on some of the key results from the projects described earlier; for additional details, we refer the reader to the full papers cited earlier.

Maintenance

Patch Generation with Prophet and Genesis

Some of the headline results for Prophet are as follows: After running Prophet on a benchmarks set, it was able to generate validated patches for 38/69 bugs; after manual validation, we were able to confirm that it had generated correct patches for 18/69 bugs and for 15 out of those 18 bugs, the correct patch was ranked first. The results compared to alternative patching systems are summarized in Figure 10.

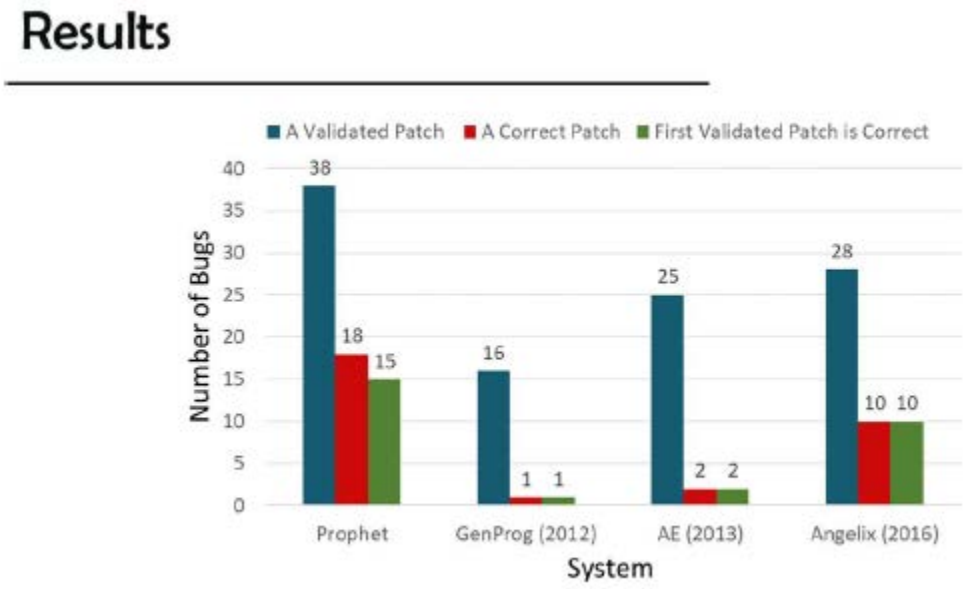

Figure 10 Comparison Between Prophet and Other Patch Generation Systems

In the case of Genesis, we compared it with the Pattern-based Automatic program Repair (PAR) system from ICSE 2013 [13], which uses manual transformations (Genesis and Prophet cannot be compared side-by-side because Genesis operates on Java code while Prophet operates on C code). Those results are summarized in Figure 11.

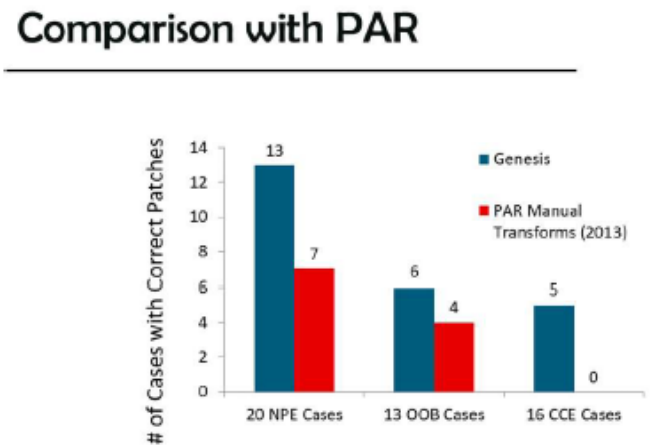

Figure 11 Comparison Between Genesis and PAR

Development

DemoMatch: Learning to use APIs with execution data

All the results for DemoMatch were published in the PLDI 2017 paper mentioned earlier, but some of the highlights are as follows. First, we were able to generate a database of traces from 130 Java Swing tutorial programs and dozens of "how-to" sample programs which totaled 20,000 lines of code and 250 million events. For 28 demos, we found that in 26 of the 28 demos captured from 3 applications, the right code snippet containing the key implementation methods was in the top 10 suggestions produced by the tool.

We also constructed a Database of traces from 5 Eclipse Plugins. We used it to explore behaviors including editor folding, auto completion, auto edit, outline navigation. In a controlled experiment we found that even for such complex APIs, the system produced most of the necessary code for these tasks with 0 irrelevant statements.

Data Enhanced Sketch and Constraint Solving

In Figure 12 we illustrate the results of replacing the native encoding layer in CVC 4 with an encoding layer synthesized using our data-driven technique. For each of the problem domains, we trained a different encoder and show that the generated encoding is able to solve problems that could not be solved by the original. We also noted that there is significant domain specificity in the encoders. Encoders trained for one domain actually do quite poorly when used in another domain.

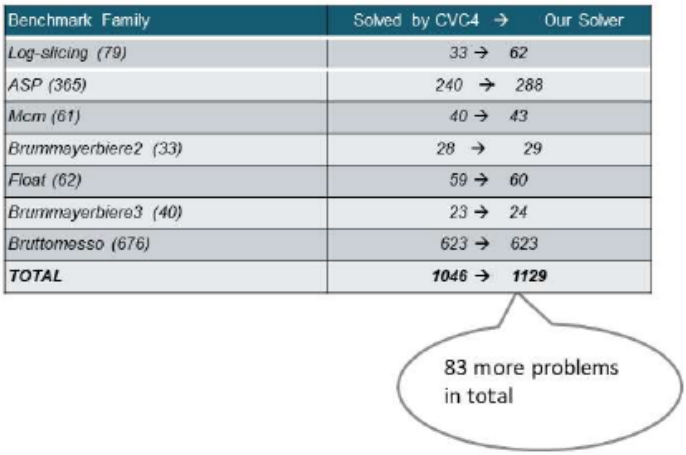

Figure 12 Results of Synthesized Encoder Vs. Native Encoder for CVC4

In the case of the synthesized simplification rules, we were also able to show that synthesizing the rules both helped us generate faster solvers and also led to solvers that were more specialized for their domain. It is important to point out that these are also among the largest artifacts ever generated using constraint-based synthesis: Some of the encoders are over 50K Lines of Code (LOC).

Automatically Synthesizing Learning Pipelines

In order to evaluate this work, we compared it to Autosklearn and Tree-Based Pipeline Optimization Tool (TPOT), two automated Machine Learning (ML) tools with pre-defined search spaces for components. We also compared the AL tool to a simple baseline and a default value predictor. All tools use Python's data analysis ecosystem (pandas, numpy, scikit-learn, xgboost). The system was evaluated against 20 benchmark datasets. Autosklearn, TPOT run for 1 hour on each benchmark dataset, whereas our tool runs in less than 5 min. Our AL tool was trained on 500 scripts targeting 9 datasets (collected through Kaggle, a data science website). From the experiments, we drew some important observations; first, the search space extracted from example programs enabled execution on more datasets, whereas the other systems need to be extended manually to account for new components. We also observe the pipelines generated are comparable in performance to existing systems (for inputs where all systems execute). The results are summarized in Figure 12.

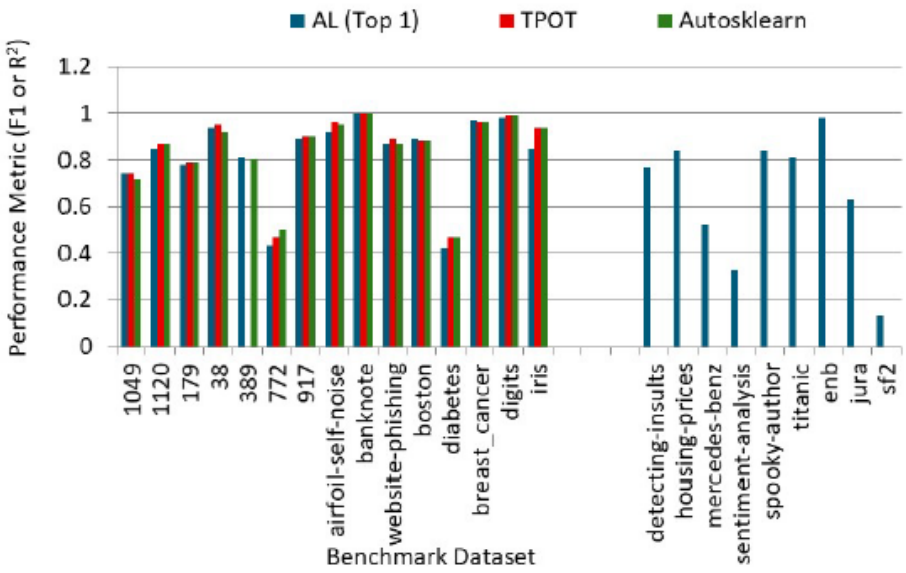

Figure 13 Results From AL Tool Compared with TPOT and Autosklearn

Learning a Synthesizer

The key result from this effort is that starting with a very high-level language, we are able to synthesize interesting DSL components, including high-order functions such as "filter" directly from the corpus, as illustrated in Figure 14.

Figure 15 shows how after every iteration of the algorithm, the system is able to solve more synthesis problems and learn them faster. The orange lines are the results including the learned search policy; the green line is without learning the search policy, leading to lower quality results.

	List Functions	Text Editing	Symbolic Regression
Programs & Tasks	[7 2 3]→[7 3] [1 2 3 4]→[3 4] [4 3 2 1]→[4 3] $f(\ell) = (f_1\ \ell\ (\lambda\ (x)\ (> x\ 2)))$ [7 3]→False [3]→False [9 0 0]→True [0]→True [0 7 3]→True $f(\ell) = (f_3\ \ell\ 0)$ [2 7 8 1]→8 [3 19 14]→19 $f(\ell) = (f_2\ \ell)$	+106 769-438→106.769.438 +83 973-831→83.973.831 $f(s) = (f_0$ "." "-" $(f_0$ "." " " (cdr s))) Temple Anna H →TAH Lara Gregori→LG $f(s) = (f_2\ s)$	 $f(x) = (f_1\ x)$  $f(x) = (f_6\ x)$  $f(x) = (f_4\ x)$  $f(x) = (f_3\ x)$
DSL	$f_1(\ell,p) = (\text{foldr}\ \ell\ \text{nil}\ (\lambda\ (x\ a)$ (if (p x) (cons x a) a))) (f_1: *Higher-order filter function*) $f_2(\ell) = (\text{foldr}\ \ell\ 0\ (\lambda\ (x\ a)$ (if (> a x) a x))) (f_2: *Maximum element in list* ℓ) $f_3(\ell,k) = (\text{foldr}\ \ell\ (\text{is-nil}\ \ell)$ $(\lambda\ (x\ a)\ (\text{if a a}\ (= k\ x))))$ (f_3: *Whether* ℓ *contains* k)	$f_0(s,a,b) = (\text{map}\ (\lambda\ (x)$ (if (= x a) b x)) s) (f_0: *Performs character substitution*) $f_1(s,c) = (\text{foldr}\ s\ s\ (\lambda\ (x\ a)$ (cdr (if (= c x) s a)))) (f_1: *Drop characters from* s *until* c *reached*) $f_2(s) = (\text{unfold}\ s\ \text{is-nil car}$ $(\lambda\ (z)\ (f_1\ z\ "\ ")))$ (f_2: *Abbreviates a sequence of words*)	$f_0(x) = (+\ x\ \text{real})$ $f_1(x) = (f_0\ (*\ \text{real}\ x))$ $f_2(x) = (f_1\ (*\ x\ (f_0\ x)))$ $f_3(x) = (f_0\ (*\ x\ (f_2\ x)))$ $f_4(x) = (f_0\ (*\ x\ (f_3\ x)))$ (f_4: *4th order polynomial*) $f_5(x) = (/\ \text{real}\ x)$ $f_6(x) = (f_4\ (f_0\ x))$ (f_6: *rational function*)

Figure 14 Some Examples of DSL Components We Were Able to Learn For Each Domain

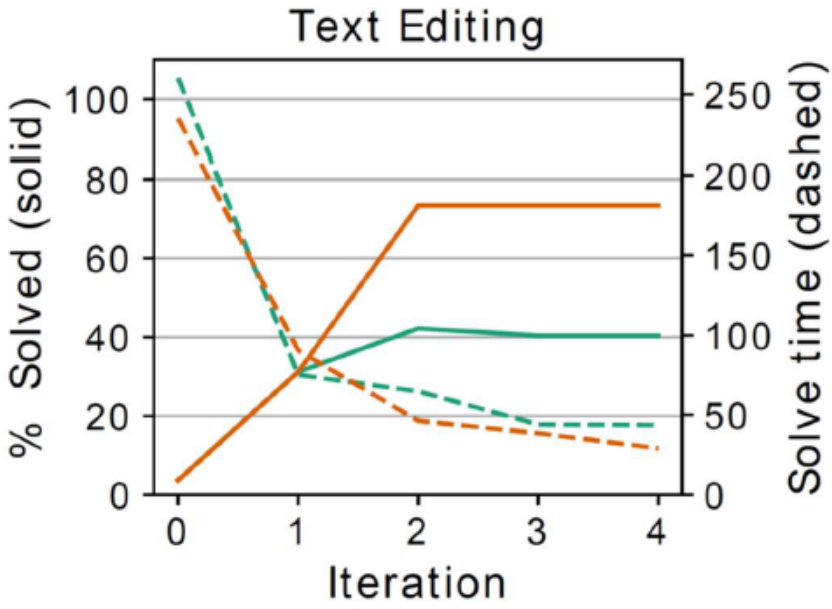

Figure 15 Results for the Text Domain. The Green Line is Without the Learned Policy (Only Learning the DSL)

From Hand Drawings to Graphics Programs

We were able to generate images for large numbers of diagrams and show that the generated diagrams generalize to different numbers of elements in the diagram as illustrated in figure 16.

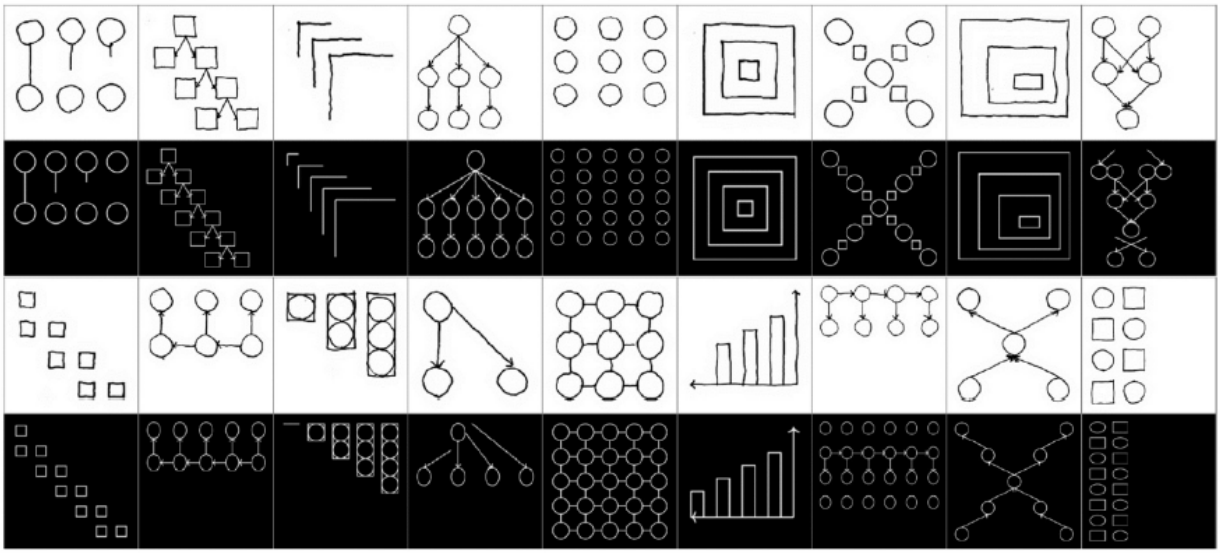

Figure 16 Example of Hand Drawings to Graphics

Selecting representative examples for synthesis

For the key experiment in the paper we were able to show that the proposed approach outperforms competing approaches for selecting examples. An interesting observation was that some potential baselines were in fact much worse than adding all the examples at once, which was still significantly worse than our approach.

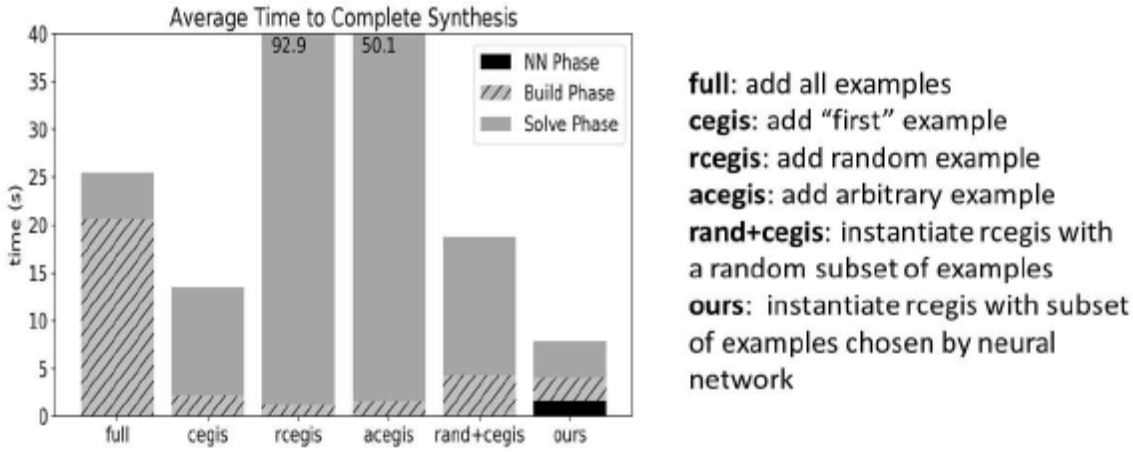

Figure 17 Selecting Representative Examples for Synthesis

Conclusions

This report provides a comprehensive, cumulative and substantive summary of the progress and significant accomplishments achieved during the period covered by the agreement. Additional information on the research accomplishments highlighted in this report are available in 12 publications that were produced as part of this project. We believe these results clearly demonstrate some of the original hypothesis behind the MUSE program. First that the interaction between programming systems and data-driven machine learning can lead to powerful new capabilities, and two that focusing on enclaves, that is, programs from similar domains is beneficial for many tasks. For example, the work on data enhanced sketching and constraint solving showed that solvers trained in one domain generalized really well to new problems in that domain, but would do badly on very different domains.

Another measure of the success of the program is in the interest from industry; our group is now doing follow-up work with Intel and Siemens.

References

[1] E. L. F. L. M. R. Stelios Sidiroglou-Douskos, "Automatic error elimination by horizontal code transfer across multiple applications.," in PLDI, 2015.

[2] E. L. A. E. F. L. M. R. Stelios Sidiroglou-Douskos, "CodeCarbonCopy," in ESEC/SIGSOFT FSE 2017, 2017.

[3] M. R. Fan Long, "Automatic patch generation by learning correct code," in POPL, 2016.

[4] P. A. M. R. Fan Long, "Automatic inference of code transforms for patch generation," in ESEC/SIGSOFT FSE, 2017.

[5] I. K. A. S.-L. Kuat Yessenov, "DemoMatch: API discovery from demonstrations," in PLDI, 2017.

[6] R. S. A. S.-L. Jeevana Priya Inala, "Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers," in SAT, 2016.

[7] A. S.-L. Rohit Singh, "SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules.," in FMCAD, 2016.

[8] M. R. Jose Cambronero, "Generating Component-based Supervised Learning Programs From Crowdsourced Examples," MIT-CSAIL-TR-2017-015, https://dspace.mit.edu/handle/1721.1/112949, 2017.

[9] L. M. M. S.-M. A. S.-L. J. T. Kevin Ellis, "Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Induction," in NeurIPS, 2018.

[10] D. R. A. S.-L. J. T. Kevin Ellis, "Learning to Infer Graphics Programs from Hand-Drawn Images," in NeurIPS, 2018.

[11] J. P. I. Y. P. A. S. A. S. D. R. A. S.-L. W. M. Tao Du, "InverseCSG: automatic conversion of 3D models to CSG trees," ACM Transactions on Graphics, Volume 37, pp. 213:1-213, 2018.

[12] Z. M. A. S.-L. L. P. K. Yewen Pu, "Selecting Representative Examples for Program Synthesis," in ICML, 2018.

[13] D. Kim, J. Nam, J. Song and S. Kim, "Automatic patch generation learned from human-written patches," 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, 2013, pp. 802-811.

List of Acronyms

AL: Automatically Synthesizing Learning

API: Application Programming Interface

CCC: Code Carbon Copy System

CNF: Conjunctive Normal Form

CP: Code Phage

DARPA: Defense Advanced Research Program Agency

DSL: Domain Specific Language

EC^2: Explore-Compress-Compile Algorithm

FMCAD: Formal Methods in Computer-Aided Design

FSE: Future of Software Engineering

ICSE: International Conference on Software Engineering

ICML: International Conference on Machine Learning

LOC: Lines of Code

MIT: Massachusetts Institute of Technology

ML: Machine Learning

MUSE: Mining and Understanding of Software Enclaves

NeurIPS: Neural Information Processing Systems

PAR: Pattern-based Automatic program Repair

PDF: Portable Document Format

PI: Principal Investigator

PLDI: Programming Language Design and Implementation

POPL: Principles of Programming Languages

PS: Programming Systems

SAT: Satisfiability Problem

SAT: International Conferences on Theory and Applications of Satisfiability Testing

SIGGRAPH: Conference and Exhibition on Computer Graphics and. Interactive Techniques

SMT: Satisfiability Modulo Theories

TPOT: Tree-Based Pipeline Optimization Tool

UCI: University of California Irvine