Language Fuzzing Using Constraint Logic Programming

Kyle Dewey Jared Roesch Ben Hardekopf
University of California, Santa Barbara
{kyledewey, jroesch, benh}@cs.ucsb.edu

ABSTRACT

Fuzz testing builds confidence in compilers and interpreters. It is desirable for fuzzers to allow targeted generation of programs that showcase specific language features and behaviors. However, the predominant program generation technique used by most language fuzzers, stochastic context-free grammars, does not have this property. We propose the use of constraint logic programming (CLP) for program generation. Using CLP, testers can write declarative predicates specifying interesting programs, including syntactic features and semantic behaviors. CLP subsumes and generalizes the stochastic grammar approach.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Constraint and logic languages; D.2.5 [**Testing and Debugging**]: Testing tools

Keywords

Fuzzing; automated program generation; automated testing

1. INTRODUCTION

Language fuzzing is a proven strategy for testing the correctness of compilers and interpreters. However, existing language fuzzing techniques are, in general, ad-hoc and limited in their scope and their effectiveness. We propose a new method for language fuzzing based on *constraint logic programming* (CLP). Our CLP method generalizes and improves upon the existing fuzzing strategies.

Language fuzzing is used to test language implementations, i.e., compilers and interpreters. A fuzzer automatically generates programs in the language being tested and feeds them to one or more implementations of that language. The fuzzer can then use an oracle of correctness or differential testing [11] to determine if the program was executed correctly. Fuzzing is often used as a bug-finding tool, but it can also be used to increase confidence in the correctness of a language implementation. For example, if a fuzzer can generate a million programs that all stress a particular language feature in diverse ways without ever triggering a bug, then the implementor can have a high degree of confidence in the correctness of that language feature. With respect to both finding bugs and increasing confidence in correctness, the most important part of a language fuzzer is the technique used to generate programs, because that technique ultimately determines *what* is tested and *how* it is tested.

Program Generation. A desirable property of program generation for language fuzzing is the ability to *target* generation, i.e., to specify particular combinations of syntactic language features and semantic language behaviors that the generated programs should all exhibit. For example, suppose that the language implementation has been augmented with a new optimization or language feature and the implementor wants to test that the new version of the implementation is correct. In order to do so, the fuzzer should specifically generate programs that would trigger the optimization or that would use the new language feature. Purely syntactic specifications are not sufficient for this purpose; the fuzzer must be able to specify program *behaviors* to achieve the desired results.

Current State of the Art. The predominant program generation strategy for language fuzzing is based on *stochastic context-free grammars* [11]. The fuzzer takes a grammar describing the syntax of the language being tested and an assignment from the grammar's productions to probabilities. The resulting probabilistic grammar is used to generate syntactically valid programs whose various expressions conform to the given probability distribution.

Approaches based on stochastic grammars do not have the ability to target program generation for specific syntactic features and semantic behaviors. Traditional stochastic grammars are only able to guide program generation by tuning the grammar's probabilities (often one of the most laborious aspects of using stochastic grammars). Even using additional extended features such as conditional probabilities and bounds, stochastic grammars remain rooted in syntax, preventing testers from directly specifying semantic behaviors under test.

Key Insight. Our central insight is that constraint logic programming (CLP) is an ideal vehicle for language fuzzing that strictly subsumes the stochastic grammar approach; i.e., using CLP predicates we can specify stochastic gram-

This work was supported by NSF CCF-1319060.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642963 .

mars but we can also specify combinations of syntactic features and semantic behaviors in a declarative and efficient way. The nature of CLP means that these predicates can be directly used to automatically generate many satisfying programs, i.e., programs that contain the specified features and behaviors. CLP is not an alternative approach from stochastic grammars, but rather a complete replacement with strictly greater expressiveness.

Contributions. Our specific contributions in this paper are the following: A general description of how to apply CLP to language fuzzing in order to yield the benefits outlined above (Section 3); a discussion of how to apply our method to fuzz JavaScript, a dynamically typed scripting language with objects, prototype-based inheritence, and higher-order functions (Section 4); and an evaluation of our method that uses our techniques for fuzzing JavaScript to compare against a purely stochastic grammar approach (Section 5).

2. RELATED WORK

We review related work with respect first to language fuzzing, then with respect to declarative test generation.

Language Fuzzing. The most common technique used to automatically generate programs for testing is the stochastic grammar approach described in Section 1. State of the art language fuzzers based on this method include `jsfunfuzz` [13], LangFuzz [5], and CSmith [19]. `jsfunfuzz` was developed internally by Mozilla specifically for JavaScript in order to test the Spidermonkey JavaScript engine. LangFuzz is targeted to dynamic languages, including JavaScript, but is not restricted to a single language. Langfuzz extends stochastic generation to include *mutation*: given an existing test suite, LangFuzz will randomly mutate the existing programs in order to generate new ones. CSmith uses stochastic generation as the basis of a fuzzing tool for C, supplemented with auxiliary C-specific techniques and analyses to handle potentially undefined behavior and other C-specific problems. We demonstrate in the rest of the paper that our proposed CLP method is able to specify the same things as stochastic grammars and also specify semantic behaviors and relations between syntactic expressions.

Declarative Test Generation. Our CLP technique uses declarative specifications to generate programs for language fuzzing. No existing language fuzzers use this approach, but there are techniques for automatic data structure generation (e.g., red-black trees) that use similar ideas.

TestEra [10, 16] allows programmers to declaratively specify data structures using predicates written in Alloy [6]. Alloy compiles the specification down to a SAT instance and uses a SAT solver to generate satisfying data structure instances. UDITA [3] allows users to specify data structure predicates in Java using a hybrid declarative/imperative style. It allows for the definition of generators which can be called nondeterministically, which is a simplification of earlier purely imperative-style generators (like those in ASTGen [2]). Korat [1] allows for purely declarative data structure predicates to be specified in Java. Korat's technique is in some ways similar to, though distinct from, a logic programming-based approach. However, it has no ability to solve symbolic constraints and cannot employ symbolic reasoning about the values contained in the data structures.

None of the above existing declarative test generation strategies use constraint logic programming, and none have been applied to program generation for language fuzzing.

3. CLP FOR PROGRAM GENERATION

In this section we discuss how to use constraint logic programming for program generation. We briefly review standard CLP syntax and semantics, but readers unfamiliar with CLP may wish to consult further references [7, 8]. We will begin with a syntactic description of a simple arithmetic expression language and express progressively more interesting properties for expression generation. Section 4 will then discuss applying these ideas to program generation specifically for JavaScript.

Syntactic Expressions. We use the following arithmetic expression language for the examples in this section:

$$e \in \mathit{ArithExp} ::= n \in \mathbb{Z} \mid e_1 + e_2$$

An arithmetic expression e is either an integer or an addition of two expressions. We can describe this grammar in CLP using Prolog-style syntax as follows:

```
exp(X) :- INTMIN <= X, X <= INTMAX.
exp(add(Y,Z)) :- exp(Y), exp(Z).
```

A *clause* consists of a head and a body and is terminated by a period, like so: "*head* `:-` *body*.". The previous example has two clauses, both for the predicate `exp`. The `:-` can be read as "if" and a comma stands for conjunction. Labels starting with capital letters are logical variables. Therefore, the first clause states "the value of variable `X` is an `exp` if the value of `X` is an integer between INTMIN and INTMAX" (where we assume INTMIN and INTMAX are variables that contain the minimum and maximum integer values, respectively). The second clause states "`add(Y,Z)` is an `exp` if `Y` is an `exp` and `Z` is an `exp`".

We can use this definition in two distinct ways: to *recognize* valid expressions and to *generate* valid expressions. In the first case, we pass a potential expression as an argument to `exp` and it will return either **true** if it is a valid expression or **false** otherwise. The more interesting case for fuzzing is generation: if we want to generate valid expressions instead, we can use a query like the following:

```
:- exp(E), write(E), fail.
```

The CLP engine will attempt to find a value for the logical variable `E` that will make `exp` true; once it does it will write that value to output, then fail. Failure will cause the engine to backtrack and attempt to find a different value for `E`; this process will continue indefinitely and generate an infinite stream of valid expressions.

One caveat is that the resulting expressions will not be concrete; instead, they will contain *symbolic variables* (standing for unknown integers) that are subject to a set of *constraints* derived from the clauses. For example, one of the generated expressions would be `add(`x`,add(`y`,`z`))` where x, y, and z are symbolic variables standing for unknown integers. The CLP engine remembers the constraints INTMIN $\leq \{x, y, z\}$ and $\{x, y, z\} \leq$ INTMAX. To get a concrete expression, we simply ask the CLP engine to *label* the symbolic variables, i.e., to find concrete values that satisfy the constraints; the engine guarantees that satisfying values must

exist. Symbolic variables and constraints are how CLP implements *symbolic reasoning*, and it is the reason that CLP is strictly more powerful than logic programming.

Bounding Size. The default search strategy of most CLP engines is unbounded depth-first search. This strategy controls the order in which expressions are generated, and for an infinite stream of expressions will control which expressions are generated within a finite amount of time. However, it is simple to change this search strategy; for example, we can change the query to:

```
:- call_with_depth_limit(exp(E), 5, CurrDepth),
   CurrDepth \== depth_limit_exceeded, write(E), fail.
```

where `call_with_depth_limit` is a library function of the CLP engine, `depth_limit_exceeded` is a built-in atom, and `\==` means "not equal to". This query bounds the depth of the CLP engine's search to five; the generated expression trees are thus bounded to height five. We can build on this primitive to implement an iterative deepening search strategy as well. Using this technique we can guarantee *minimal* satisfying test cases, rather than relying on a separate test case reducer (e.g., [17, 15, 20, 21, 12, 14]) as is common with other language fuzzing techniques (e.g., [19, 13, 9, 11, 4]). We can also bound expressions by number of nodes instead of by depth, and we can specify a minimal size in addition to a maximal size.

Stochastic Grammar. CLP subsumes the stochastic grammar technique for program generation. This means that we can specify stochastic grammars using CLP, and do so quite easily. Here is our expression definition modified to emit expressions with particular probabilities:

```
exp(X) :- maybe(0.4), INTMIN <= X, X <= INTMAX.
exp(add(Y,Z)) :- exp(Y), exp(Z).
```

where `maybe` is a library function of the CLP engine that succeeds with the given probability. This definition will generate expressions whose nodes are numbers 40% of the time and additions 60% of the time. Note that we can combine bounded size with stochastic grammars to get a program generator that randomly generates programs within the given size bounds.

Arithmetic Overflow. One of the most powerful abilities of CLP is symbolic reasoning, which enables the user to specify numeric constraints as part of a predicate. These constraints are handled by an integrated constraint solver as part of the CLP engine. As an example, we can specify that we only want to generate expressions that contain at least one arithmetic overflow:

```
exp(X, X, false) :- INTMIN <= X, X <= INTMAX.
exp(add(Y,Z), N, Over) :-
  exp(Y,N1,Over1), exp(Z,N2,Over2), N = N1+N2,
  ((N > INTMAX, Over = true) ;
   (N < INTMIN, Over = true) ;
   (N >= INTMIN, N <= INTMAX, or(Over1, Over2, Over))).
```

where semicolon represents disjunction and `or` instantiates its third argument with the logical OR of its first two arguments. In this example, the first parameter of `exp` is an expression, the second parameter of `exp` represents the value of the given expression, and the third parameter indicates whether the given expression contains an overflow. The first clause states that a constant number expression has a value equal to that number and it does not contain an overflow. The second clause's first line says that `add(Y,Z)` is an expression if `Y` and `Z` are expressions, and in addition it says that the value of `add(Y,Z)` is equal to the sum of the values of `Y` and `Z`. The next three lines are a disjunction of three possibilities: either `N`, the value of `add(Y,Z)`, is greater than INTMAX and `Over` is true (i.e., this addition overflows), or `N` is less than INTMIN and `Over` is true (also meaning that this addition overflows), or, finally, `N` is between INTMIN and INTMAX and it contains an overflow only if either `Y` or `Z` contains an overflow. To generate all expressions that contain at least one overflow, we would use the following query:

```
:- exp(E,_,true), write(E), fail.
```

where underscore means "don't care". It is important to note that because CLP uses symbolic reasoning, it does not actually iterate through all possible numbers in order to find ones that overflow—rather, it generates expressions that contain symbolic variables, and then uses its constraint solver to solve for values that satisfy the given numeric constraints.

4. GENERATING JAVASCRIPT

In this section we discuss using CLP to fuzz dynamic languages. We take JavaScript as representative of that class of languages. JavaScript is an imperative, dynamically typed language with objects, prototype-based inheritance, higher-order functions, and exceptions. JavaScript is designed to be as resilient as possible and makes liberal use of implicit conversions and other idiosyncratic behaviors. Object properties can be dynamically inserted and deleted, and when performing a property access the specific property being accessed can be computed at runtime. There are several mechanisms available for runtime reflection. Object inheritance is handled via delegation: when accessing a property that is not present in a given object *obj*, the property lookup algorithm determines whether *obj* has some other object *proto* as its prototype; if so then the lookup is recursively propagated to *proto*. We omit exact details of the CLP predicates that we use for JavaScript, but they are available in the paper's supplementary materials.

Stochastic Grammar. It is fairly simple to construct a CLP predicate describing syntactically valid JavaScript programs; it is a straightforward extension of the method we used for the example arithmetic expression language in Section 3. Just as we did there, we can add probabilities to the clauses of that predicate in order to convert it to a stochastic grammar. This yields a program generator equivalent to the publically-available description of `jsfunfuzz` [13].

However, a purely stochastic grammar is not really effective for dynamic languages. The problem is that no matter what language features are present in a generated program, it is likely that there will be a runtime type error during execution before most of those features are ever reached. For example, any attempt to access a property of the **null** or **undefined** JavaScript values will raise an exception and terminate the program's execution (unless the exception is caught and handled, which again is unlikely in a randomly generated program).

Available under the `Downloads` link at `http://www.cs.ucsb.edu/~pllab`.

$$\frac{}{\vdash \textbf{null} : \texttt{nil}} \qquad \frac{}{\vdash x : \texttt{unk}} \qquad \frac{\vdash e_1 : \texttt{unk}}{\vdash e_1.e_2 : \texttt{unk}}$$

Figure 1: A fragment of an unsound type system for JavaScript to filter out programs that attempt to directly access a property of the null value.

Absence of Runtime Errors. We can use CLP to specify JavaScript programs that avoid runtime errors. The idea is to use CLP to encode a type system and compose the syntactic JavaScript predicate with a predicate that only matches on well-typed programs. One nice property of this technique is that we can *incrementally* make the type system more powerful, starting with a simple system and adding precision as needed. We can also choose to make the type system either sound (restricting the programs that can be generated, but ensuring that all such programs will avoid runtime errors) or unsound (allowing more programs to be generated, but potentially allowing some kinds of runtime errors). Thus we have two axes of freedom to work with, allowing us a great deal of flexibility to trade off between the effort required to write predicates and the kinds of programs that will be generated.

For example, we could employ a simple, unsound type system as a first effort to help avoid some runtime errors. Figure 1 shows a fragment of a simple type system that prevents programs from directly dereferencing the **null** JavaScript value; however, it still allows dereferencing a variable which itself may have the value **null**. We can encode this fragment of the type system using CLP as follows:

```
type(null, nil).
type(var(X), unk).
type(access(E1, E2), unk) :- type(E1, unk).
```

The predicate `type` has two parameters; the first is an expression and the second is the type of that expression. The first clause states that expression **null** has type `nil`; the second clause states that a variable has type `unk` (i.e., *unknown*); and the third clause states that an object access expression has type `unk` if the type of `E1` (the object being accessed) is `unk`. This predicate would disallow programs such as "`null.foo`", but still allow programs such as "`var x = null; x.foo`". If there are too many runtime errors in the resulting programs we can extend this type system to track the types of variables more closely, either soundly or unsoundly depending on how precise we want to be and how much effort we want to put into it. In the extreme we could employ a sound, precise type system using partial dependent types that utilize the CLP engine's symbolic reasoning.

Arithmetic Overflow. It turns out that arithmetic overflow is an interesting property for testing JavaScript implementations. JavaScript numeric values are technically floating point; however for performance reasons most JavaScript engines try to represent numeric values as integers wherever possible. Therefore, at runtime the engine must detect all overflows and automatically change the numeric representation from integers to floating point values. We can extend the arithmetic overflow predicate described in Section 3 to generate valid JavaScript numeric expressions that are guaranteed to contain at least one overflow, in order to test whether the engine performs this optimization correctly.

Prototype-based Inheritance. Object are the fundamental data structure in JavaScript (even functions and arrays are special kinds of objects), and so testing object inheritance is key. To do so, we must generate programs that both *construct* and *use* non-trivial prototype chains. Stochastic grammars are unlikely to generate such programs by themselves. Using CLP, we can enforce that generated programs contain the following items in sequence, in the proper scope, potentially with unrelated code in-between:

- A declaration for some function *foo*.
- A statement *foo*`.prototype.`*fld* `=` *exp*, where *fld* is some string and *exp* is a valid JavaScript expression, seeding *foo*'s prototype with the property *fld*.
- A statement `var` *x* `= new` *foo*`()`, constructing a new object whose prototype is the same as *foo*'s.
- A statement *x*`.`*fld*, triggering prototype lookup.

A high-level view of this predicate is the following:

```
proto(S) :-
  inSequence([inherFunction(Name),
              optional1(inherPrototypeSet(Name)),
              optional1(inherPrototypeAdd(Name)),
              =(Rest)], S),
  astContains(Rest, inherNew(Name), stmt, stmt).
```

where the `inSequence` predicate ensures that its arguments happen in sequence, however they are not necessarily consecutive, i.e., an arbitrary number of other expressions may occur in-between them. This sequence specifies a function declaration with a given name `Name`, followed by an optional setting of that function's `prototype` field, followed by an optional setting of a field of that function's prototype, followed by the rest of the AST.

We can extend this specification in various ways to make for even more interesting tests: to require a minimum depth for the prototype chain, to require the use of implicit conversion to convert non-objects to objects and then perform prototype lookup on the resulting objects, to construct multiple interleaved prototype chains, etc.

With + Closures. JavaScript has some very obscure and idiosyncratic behaviors that implementations must get correct. One example is the combination of the `with` expression with closures. The `with` expression changes the current scope, affecting variable lookup. Closures should preserve the current scope, but closures and `with` have complex interactions that make the proper behavior unclear and difficult to get correct. A particularly tricky case is when a closure is created inside of a `with` expression, and then subsequently called outside of that `with` expression.

We can specify a CLP predicate that generates JavaScript programs with the following requirements:

- The program contains a `with` expression that itself contains a function definition.
- The function definition contains multiple free variables that are defined in various scopes with respect to the `with` scope and the scope outside of the `with`.
- The closure value formed from that function is passed outside of the `with` and subsequently called.

A high-level view of this predicate is the following:

```
withclo(S) :- inSequence([withCloWith(Name),
                          withCloCall(Name)], S).
```

where `withCloWith` ensures that the generated program contains a `with` expression containing a closure and `withCloCall` ensures that the closure is called outside of the `with`. This is an example of how CLP allows the tester to concentrate on a specific set of language features and a specific interaction between those language features.

5. EVALUATION

In this section we evaluate the effectiveness of CLP-based program generation versus a stochastic grammar approach. We first explain our experimental methodology, then we present and discuss our results for JavaScript.

5.1 Methodology

We compare two different approaches to program generation for language fuzzing: **sto** (the stochastic grammar approach) and **clp** (our CLP-based approach). The comparison is a bit misleading because CLP can implement stochastic grammars; in fact, all of the approaches are implemented in SWI-Prolog [18], a publically-available Prolog engine that contains a constraint solver to implement CLP. The comparison does showcase the additional expressiveness and efficiency of CLP over purely stochastic grammars. While we requested implementations of the existing JavaScript fuzzers `jsfunfuzz` [13] and LangFuzz [5] from Mozilla for comparison with our technique, our request was ignored. Our entire infrastructure, including our implementation of stochastic grammars and the exact predicates that we use for JavaScript, is available in the supplementary materials located under the `Downloads` link at `http://www.cs.ucsb.edu/~pllab`. Our experiments are run on a system with 12 Intel Xeon@1.9 Ghz cores and 32 GB memory. All experiments are single-threaded.

The metric that we use to measure program generation effectiveness is *generation rate*, in terms of programs per second. We measure three distinct kinds of generation rate: **total**, **unique**, and **interesting**. The **total** rate is a measure of how quickly each approach can generate programs, without regard to what kinds of programs are being generated. The **unique** rate is a measure of how quickly each approach can generate unique programs, i.e., ignoring duplicate ASTs. For this metric, unless otherwise stated, variable names and the values of number, string, and boolean literals are irrelevant. For example, the programs `x + 6` and `y + 7` would be considered identical. The **interesting** rate is a measure of how quickly each approach can generate programs that match some tester-given criteria for being interesting; in our case, *interesting* means that the program is accepted by one of our predicates discussed in Section 4. To compute the generation rate, we allow each approach to generate programs for five minutes and then divide the resulting number of programs (of each category, **total**, **unique**, and **interesting**) by 300 seconds to get units of programs per second. We report separate **total**, **unique**, and **interesting** numbers for the **sto** approach; the **clp** approach only generates unique, interesting programs and so we only report the **interesting** number for that approach.

To be clear, our generation metric is not intended to focus on how fast programs can be generated by the various techniques, but rather to reveal how well the techniques can be tuned to generate interesting programs for various definitions of "interesting", and how easily they can be targeted for different interesting properties.

Details of the sto Approach. For JavaScript, we create a predicate `exp` that describes syntactically valid programs. We then add probabilities as discussed in Section 3 to create a stochastic grammar, carefully tuned to favor the generation of unique programs. Querying this predicate using "`:- exp(E), write(E), fail.`" will yield a stream of randomly generated, syntactically valid programs. This method is equivalent to the current state of the art for stochastic grammar-based approaches [2, 5, 13, 19].

Details of the clp Approach. The CLP approach uses the predicates specifying interesting programs directly in order to generate satisfying programs; thus the generated programs are guaranteed to be both unique and interesting. We set the search strategy used by the CLP approach for all predicates to bounded depth-first search. Again, while our experiments treat **sto** and **clp** as distinct approaches, we should be clear that in reality **clp** subsumes **sto** and that these approaches can all be easily combined together in various ways.

5.2 JavaScript Program Generation

We evaluate four predicates for generating JavaScript programs: `js-err`, generating programs with fewer runtime errors; `js-overflow`, generating programs that contain at least one arithmetic overflow; `js-inher`, generating programs that construct and use prototype-based inheritance chains; and `js-withclo`, generating programs that construct closures inside of `with` expressions and then call them outside of those `with` expressions. With the `js-overflow` predicate, we strengthen our definition of uniqueness to consider programs which have overflows at the same positions to be the same. For the `js-err` predicate, we use a simple, unsound type system (however, one that is more extensive than the example shown in Section 4) that attempts to ensure that (1) expressions that are used for property access or as the subject of a `delete` are neither `null` nor `undefined`; and (2) expressions that are called as functions, either directly or via `new`, are actually function values. Table 1 gives the generation rates for **sto** compared to **clp**.

We can group the predicates based on their behaviors:, with `js-err` in one class and `js-overflow`, `js-inher`, and `js-withclo` in the other class. For `js-err`, fully 85% of the unique programs are interesting. While **clp** generates slightly fewer total programs than **sto**, it generates 2.3× as many interesting programs during the same period of time. This is the best that **sto** does in comparison to **clp**, and the two major reasons are that (1) for this predicate we are using an unsound type system, thus it is easier for the stochastic grammar to generate programs matching the predicate; and (2) for this predicate we bound the search space for both **sto** and **clp** to programs whose abstract syntax trees are at most height seven. Both of these facts favor **sto** in our experiments; by making the type system more sound or by increasing the bound on program size, **clp** would do progressively better than **sto**.

The remaining three predicates have a very different story. The **interesting** generation rate for **sto** is 0 or very close to 0, while the generation rate for **clp** for two of the pred-

Predicate	total	unique	interesting	clp	$\frac{\text{clp}}{\text{interesting}}$
`js-err`	53,335	28,614	24,210	56,658	2.3
`js-overflow`	49,635	22,631	0.303	1,229	4,056
`js-inher`	20,677	11,195	0	304,890	∞
`js-withclo`	31,458	17,132	0.057	302,472	5,306,526

Table 1: Generation rate of sto versus clp, in units of programs per second. We report total, unique, and interesting separately for sto; they are all the same number for clp. The last column is the ratio between the clp and sto interesting program generation rates (higher is better for clp).

icates is 10–15× higher than **sto**'s **total** generation rate. The reason that **clp** is so much faster than **sto** in this case is because **sto** is randomly searching the space, i.e., each time it finds a program it restarts the search from scratch. In contrast, **clp** is using bounded depth-first search, i.e., when it finds a program it searches the nearby space to find other programs as well. The behavior of **sto** is characteristic of stochastic grammars, not specific to our implementation. The **clp** generation rate for `js-overflow` is much lower than the **clp** generation rate for the other predicates because `js-overflow` heavily exercises the constraint solver.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have demonstrated that constraint logic programming is a promising technique to automatically generate programs for language fuzzing. CLP-based program generation has several desirable properties, including the ability to target program generation for specific combinations of syntactic features and semantic behaviors. We have empirically demonstrated those properties and their benefits in comparison to stochastic grammars, focusing on JavaScript program generation. Our results show that the CLP-based approach does significantly better than stochastic grammars for generating interesting programs.

There are several avenues for advancing this work in the future. One idea is to apply CLP to generate interesting programs in other kinds of languages, such as strongly statically typed (e.g., Scala) or weakly statically typed (e.g., C). Another possibility is to modify the CLP engine to replace the numeric constraint solver with a more powerful SMT solver, with access to more theories such as the theory of strings or the theory of bit-vectors. An application of this improved engine might be to automatically generate interesting hardware designs to fuzz hardware synthesis tools.

7. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[2] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

[3] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM.

[4] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE*, pages 621–631. IEEE Computer Society, 2007.

[5] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[6] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Apr. 2002.

[7] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM.

[8] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19:503–581, 1994.

[9] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE '05, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society.

[10] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[11] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, December 1998.

[12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA, 2012. ACM.

[13] J. Ruderman. Introducing jsfunfuzz, 2007.

[14] J. Ruderman. Introducing lithium, a testcase reduction tool, 2007.

[15] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[16] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 133–142, New York, NY, USA, 2004. ACM.

[17] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[18] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[20] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

[21] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.