Developing a small but useful set of inputs for tests is challenging. We show that a domain-specific language backed by a constraint solver can help the programmer with this process. The solver can generate a set of test inputs and guarantee that each input is different from other inputs in a way that is useful for testing.

This paper presents Iorek: a tool that empowers the programmer with the ability to express to any SMT solver what it means for inputs to be different. The core of Iorek is a rich language for constraining the set of inputs, which includes a novel bounded enumeration mechanism that makes it easy to define and encode a flexible notion of difference over a recursive structure. We demonstrate the flexibility of this mechanism for generating strings.

We use Iorek to test real services and find that it is effective at finding bugs. We also build Iorek into a random testing tool and show that it increases coverage.

CCS Concepts: •Software and its engineering → Domain specific languages; Software testing and debugging; Constraints;

Additional Key Words and Phrases: solver-aided languages, test input generation, generators

ACM Reference format:
DOI: 10.1145/3133915

1 INTRODUCTION

Coming up with an interesting set of test inputs for a software service is a time-consuming, tedious, and error-prone task. This is precisely the kind of task that constraint solvers are ideal for. In the case of testing, solvers can generate a set of test inputs for some code and guarantee that each input is both a legal input according to a specification and different from other inputs in a way that is useful for testing.

Consider, for example, the purchasing workflow for a retail website. The API for this code specifies legal inputs: Items have names, categories, and prices. A programmer who is testing this code likely does not want to test every single item—there may be hundreds of millions of items. The programmer can instead specify what it means for two items to be different for the sake of testing: Two items may be different if they are in different categories (say, Music and Shoes) or have prices at least $50 apart. A constraint solver can guarantee...
that every individual item it generates for this test has a name, category, and price, and that every pair of items are in different categories or have prices at least $50 apart.

Our tool Iorek (pronounced your-ik) helps programmers express this. Iorek combines a testing framework with a domain-specific language backed by a constraint solver—a solver-aided language (Torlak and Bodik 2013). A Iorek programmer writes an incomplete test and leaves some inputs blank. For every blank input, Iorek generates many inputs that satisfy a specification and are different in the way the programmer defines.

Iorek exposes a constraint language to express what it means for any two inputs to be different. In this way, Iorek empowers the programmer with control over the space of inputs. The constraint language can express relations (for example, strings of different lengths, or numbers at a certain distance), combinations of constraints, and a novel bounded enumeration mechanism. The enumeration mechanism makes it possible to express the notion of a representative set of values for a recursive structure over an infinite domain and encode it for any SMT solver. This enables programmers to define expressive combinations of constraints over structured data. It also provides a level of abstraction: Programmers can leave reasoning about SMT-supported datatypes like numbers and strings to the solver.

We demonstrate the power and flexibility of the enumeration mechanism in generating string inputs. To accomplish this, we extend Rosette (Torlak and Bodik 2013) with strings and regular expressions (regexes), which are recent additions to SMT (Bjørner et al. 2012; Liang et al. 2014; Trinh et al. 2014; Uhler and Dave 2013; Zheng et al. 2013). For example, our evaluation suggests that using the structure of a regex to define inputs can sometimes increase code coverage. That is, a programmer generating three tests with inputs constrained by [a-z]+[0-9]+_ may prefer "abc", "1234", and "_" over "123", "4567", and "89012". The enumeration mechanism makes it easy to express this to a solver. The use of an SMT solver allows programmers to combine this mechanism with other notions of difference (such as strings of different lengths) and leave reasoning about strings to the solver.

We show that Iorek is effective at finding bugs, increases code coverage, and scales to complex queries: We integrate a prototype into a framework at Amazon and use it to find four bugs in real services in development—developers have accepted three of these bugs.1 We also use Iorek to generate inputs for the fully-automatic testing tool Randoop (Pacheco and Ernst 2007) and find that the inputs Iorek generates increase code coverage on 15 of 18 benchmarks in a string benchmark suite. Finally, we show that compared to three alternative ways of querying a solver for many different values, the approach that Iorek uses is the only approach that leads to acceptable performance across all the kinds of queries Iorek makes.

In summary, we contribute the following:

1. We design and implement Iorek: a solver-aided language and tool for generating test inputs that lets programmers define what makes test inputs different.
2. We introduce a novel bounded enumeration mechanism that generates a representative set of values and formulate a way to express this mechanism to a solver. We demonstrate the power and flexibility of this mechanism in the domain of strings.
3. We build a prototype of Iorek into a testing framework at Amazon and use it to find bugs in real services in development.
4. We use Iorek to generate inputs for an automatic testing tool and show that it increases code coverage and that the flexibility it provides is useful.
5. We evaluate the performance of different mechanisms for querying a solver for many different values and show that Iorek scales to large and complex queries.

1The other does not cause unexpected behavior for Amazon customers.
2 USING IOREK

The core of Iorek is a solver-aided intermediate representation (IR) that is designed to interface easily with existing languages. This way, we can integrate Iorek with testing frameworks in many different languages. So far, we have implemented an interface (Figure 1) that lets programmers write Iorek tests in JUnit.

To see Iorek in action, consider the calendar service API specified in Figure 2. Every calendar event has a name, which may end in a time string. The calendar can infer information about the time of an event from its name. The calendar also interacts with a messaging system, and can import and display flight information and adjust the time zone of events accordingly. That is, if Alice flies to Vancouver at 10:00 AM on October 21st, it shows all of her events before 10:00 AM on October 21st in her local time, but all events after the time of arrival in Pacific Daylight Time.

We want to make sure that the calendar still passes a sanity check when Alice is abroad: If Alice creates two events, one that contains a time as the suffix of its name (for example, "meeting 7:00 PM" with an empty time field) and another that contains the same name and time as separate fields ("meeting" with the time field set to 7:00 PM that same day), the two events should be identical:

```java
assertEquals(create(name + suffix, none), create(name, of(suffixTime)));
```

We would like to make sure that this works when Alice flies anywhere in the world. We do not want to exhaustively test every event name and flight location (there are infinitely many event names). Instead, we want to generate event names with times that exercise different branches in the time string specification:

```
{{05:00}, {21:22}, {12:00 AM}, {5:00 PM}, ...}
```

And we want to pair each time with flight locations that are in different time zones:

```
{{05:00, Pittsburgh}, {05:00, Amsterdam}, {05:00, Vancouver}, ...}
```

We can write this test in the Iorek JUnit framework (Figure 1). We pass Iorek three things:

1. **The input specification** (Figure 2), which describes legal service inputs (events).
2. A number \( n \) (Figure 3) of test inputs to generate (in this case, 100).
3. **A test outline** (Figure 3) that defines:
   a. a JUnit test with blank inputs, and
   b. what makes any two test inputs different.

The input specification for the service API from Figure 2 exists already, as is common in industry. To write the test outline in Figure 3, we write a normal JUnit test, but in place of concrete times and locations like "05:00" and "Pittsburgh", we write blank
inputs: ?timeSuff and ?Location. These are the blank inputs that Iorek generates 100 (the argument to @Generate) concrete inputs for. We use these blank inputs to create two events, which we assert (using JUnit’s assertEquals) are the same. We also define (using constrain) what makes any two times and locations different.

We annotate the test outline with two top-level annotations: @Specification tells Iorek where to find the specification file. @RunWith(Iorek.class) tells JUnit to run the outline with the Iorek TestRunner.² The TestRunner uses the specification file, number n, and test outline to run tests in two passes:

1. Prepare, which generates inputs.
2. Run, which runs the outline with the generated inputs.

Prepare translates the specification from Figure 2 and the outline from Figure 3 into a Iorek program (in fact, the one in Figure 8 in Section 4.2). It then runs the Iorek program (not the code under test, or CUT), which queries a solver for 100 time-location pairs that satisfy the specification from Figure 2 and are different in the way we specify in Figure 3. It saves the 100 time-location pairs as an input file. Run invokes the test outline with the CUT 100 times (once with each input from the input file), replacing every blank input with a generated input. Each run produces a result (success or failure).

In this calendar example, we took a black-box view of testing, thinking about inputs rather than properties of the CUT. We could just as well use specifications for properties of the CUT. Ultimately, Iorek is agnostic to whether the approach is black-box or white-box. Iorek is a tool to express what it means for test inputs to be different—it gives the programmer control of what “different” means.

3 OUR CONSTRAINT PROBLEM, CONTRASTED

We formulate test input generation as a constraint solving problem: A solver-aided testing tool generates many different satisfying assignments (a set of inputs) for a specification. This means that every individual input should satisfy the specification and every pair of inputs should be different. That is, for some set of values $v_1, \ldots, v_n$, some assertion $A$ about each individual value, and some constraint $C$ relating values:

$$\forall 1 \leq i \leq n, A(v_i) \land \forall 1 \leq j \leq n, (i \neq j \Rightarrow C(v_i, v_j))$$

This problem is different from the problem traditional synthesis (Solar Lezama 2008) and angelic execution (Bodik et al. 2010) tools solve. These tools find a single satisfying assignment—a program that is correct for all inputs. We use an interface that is inspired by the synthesis tool SKETCH (Solar Lezama 2008) because it is clean and simple to use.

Some solver-aided languages such as Kaplan (Köksal et al. 2012) can enumerate all satisfying assignments to a specification. Bounded exhaustive testing (BET) tools (Daniel et al. 2007; Goodenough and Gerhart 1975; Khurshid and Marinov 2004; Rosner et al. 2014; Senni and Fioravanti 2012) generate all structures up to a bound. Relational logic solvers such as Kodkod (Torlak and Jackson 2007) are useful for this style of testing. While Iorek can express BET, it is not our goal—many inputs will be redundant. We expose a new mechanism for enumeration that allows the programmer to define what it means for structures to be different. This is similar to what SciFe (Kuraj et al. 2015) accomplishes, except we can express our enumerators to any SMT solver. This means that we can use established solvers, abstract away reasoning about data, and support constraints on strings (which most similar tools do not support).

Iorek is a tool to help programmers write tests, not a fully-automatic test generation tool. The core of Iorek is a DSL for controlling input generation. In this regard, it is similar to QuickCheck (Claessen and Hughes 2000), a widely used DSL for property-based testing. QuickCheck programmers guide the search process for test inputs by writing constructive generators (Claessen and Hughes 2000; Kuraj et al. 2015) which assign weights to all of the possibilities of the input space; these generators are probabilistic. The DSL Luck (Lampropoulos et al. 2017) makes it easier to write this style of generator. Iorek exposes constraints to specify what makes inputs different—these constraints are not probabilistic, and can be used structurally to limit the search space like constructive generators, but are concise like declarative generators (Boyapati et al. 2002; Senni and Fioravanti 2012). Furthermore, Iorek queries existing SMT solvers, which abstracts details of SMT-supported datatypes and frees the developer to focus on structural ways to constrain those datatypes, or on the high-level structure of datatypes in which they are embedded.

Some random testing tools such as Randoop (Pacheco and Ernst 2007) use feedback from executing tests to rule out redundant inputs. Many tools (Anand et al. 2007; Cadar et al. 2006; Chipounov et al. 2011; Fraser and Arcuri 2011; Godefroid et al. 2005; Sen et al. 2005; Tillmann and De Halleux 2008) combine symbolic execution with automatic testing techniques to produce useful tests that explore different paths. Iorek is close in spirit to Grammar-Based Whitebox Fuzzing (Godefroid et al. 2008), which uses grammar-based...
specifications to generate interesting inputs that trigger invalid program states. Unlike these tools, Iorek gives the programmer control over what it means to generate an interesting set of inputs, and is thus agnostic to whether the approach is black-box or white-box. Iorek also provides a way for programmers to generalize existing test suites to create better tests. Iorek is ultimately a tool for writing better and more expressive tests and can be used alongside fuzzing and testing tools. We demonstrate the use of Iorek with Randoop in Section 7. We discuss more related work in Section 9.

4 IOREK DESIGN AND IMPLEMENTATION

Iorek allows programmers to specify not only properties about individual inputs, but also properties about the space of generated inputs. Iorek encodes this information for a solver and ensures that all sets of inputs returned by the solver are different in the specified way.

Iorek includes an IR for generating inputs (described in Section 4.1 and formalized in Section 4.2), an embedded language of built-in constraints on the space of inputs (Section 4.3), and combinators to define new expressive constraints (Section 4.4), including a novel combinator for enumeration (Section 4.5).

4.1 The Iorek IR

The Iorek language centers around the notion of a model. A model maps symbolic values (as in some integer \( v \)) to concrete values (as in the integer 3). The Iorek environment (Figure 4) is a triple \((V, A, C)\). \( V \) maps variables to (possibly symbolic) values. \( A \) is an assertion about each individual model. \( C \) is a constraint that relates every pair of models.

The Iorek IR is implemented as a deep embedding in Rosette (Torlak and Bodik 2013). Rosette extends Racket with symbolic values that can be used interchangeably with concrete values. For example, in Rosette a programmer can add a symbolic integer \( v \) to the concrete integer 3. Iorek uses Rosette’s angelic execution primitives (to find some \( v \) for which an assertion holds).

We use the syntax \( M[v] \) and \((\text{evaluate } v \text{ model})\) (the latter from Rosette) to denote the value of \( v \) in the model \( M \) when \( v \) is symbolic, and \( v \) itself when \( v \) is concrete. For convenience, we define algebraic identities \( \text{true}_A \) for assertions and \( \text{never} \) for constraints (Figure 5). We define ++ to combine variable maps, and we lift \( \land \), \( \lor \), and \( \neg \) to work with assertions and constraints. Figure 5 shows \( \land \) for assertions and \( \lor \) for constraints; the rest are analogous.

In an environment \((V, A, C)\), a Iorek test outline evaluates to a set of models \(\{M_1, \ldots, M_n\}\) that satisfies the environment (when possible): The assertion \( A \) holds for every individual model, and the constraint \( C \) holds for every pair of models. When \( n \) is the size of the model space and \( C \) is inequality, this evaluates to the set of all models satisfying \( A \), and is deterministic. Otherwise, \(\{M_1, \ldots, M_n\}\) can be any set of satisfying models in any order.
default(V, M_x, M_y) = \exists v \in \text{range}(V), M_x[v] \neq M_y[v]

\begin{align*}
\text{models}(M_1, \ldots, M_n, V, A) & \equiv \\
\forall 1 \leq i \leq n, \\
& (\text{dom}(M_i) \subseteq \text{range}(V)) \land A(M_i)
\end{align*}

different(M_1, \ldots, M_n, V, C) = \\
\forall 1 \leq i, j \leq n, i \neq j \Rightarrow \\
\text{default}(V, M_i, M_j) \land \\
((C = \text{never}) \lor C(M_i, M_j))

M_1, \ldots, M_n \models (V, A, C) \equiv \\
\text{models}(M_1, \ldots, M_n, V, A) \land \\
different(M_1, \ldots, M_n, V, C)

Fig. 6. The satisfiability criteria, which checks that all assertions in an environment hold for each individual model, and that all constraints hold for every pair of models. The default criteria ensures no two models are the same, even when the programmer does not define C.

Fig. 7. Pseudocode for Iorek’s generation loop, which implements the satisfiability criteria from Figure 6. The loop finds a model for the assertion in the environment, making sure (via coerce) that every symbolic value in the model maps to some concrete value. It then adds an assertion that the next model should be different (defaulting to the default constraint defined in Figure 6).

Figure 6 formalizes this satisfiability criteria. Figure 7 implements the criteria in a loop, generating results over a sequence of queries.

The key insight to the generation loop is that while a constraint maps (Model, Model) → boolean, we can also think of it in its curried form, Model → Model → boolean. Any constraint holds vacuously with one model. For each subsequent model, we can encode the requirement that the model is different from all previous models as a Model → boolean—in other words, an assertion.

The generation loop (Figure 7) implements this: It curries the constraint in the environment with each model it receives from the solver and asserts that the next model should be different. These assertions accumulate into a single assertion that consists of a linear number of constraints on the model with respect to n for any given query, yet guarantees that all generated models are different. This loop is built to be efficient and generalizable. There are many alternative ways to express this notion of difference between all models to a solver; we evaluate them in Section 8.

4.2 Syntax and Semantics

Figure 8 shows an example Iorek program, and Figures 9 and 10 show the grammar and semantics. A program is a sequence of global statements ⟨e⟩* (which typically correspond to a service specification) followed by ⟨model-gen⟩s (which correspond to individual test outlines to generate values for). Iorek ⟨val⟩s are values (for example, strings, lists, and objects), which for now come from Java values.

Both ⟨a⟩ and ⟨c⟩ evaluate in embedded languages (⇓a and ⇓c). Primitive assertions ⟨p-a⟩ and primitive constraints ⟨p-c⟩ map ⟨val⟩ to assertions ⟨A⟩ and constraints ⟨C⟩, respectively.
That is, if (we omit a straightforward formalization of objects and fields). The highlighted lines correspond to the PACM Progr. Lang., Vol. 1, No. OOPSLA, Article 91. Publication date: October 2017.

Both embedded languages can add to the environment: The embedded assertion language can add variables to \( V \) and assertions to \( A \). This is necessary for the powerful enumeration construct we define in Sections 4.4 and 4.5.

There are three commands that affect the environment: The declare command defines a \( \langle \text{var} \rangle \) (Racket symbol) and maps it to a \( \langle \text{val} \rangle \) in \( V \). The specify command evaluates an assertion \( \langle a \rangle \) about a variable and adds the resulting \( \langle A \rangle \) to \( A \) as a conjunction. The constrain-solutions command evaluates a constraint \( \langle c \rangle \) about a variable and adds the resulting \( \langle C \rangle \) to \( C \) as a disjunction. All three can occur in the main program body (in which case they are global) or within a \( \langle \text{model-gen} \rangle \) (in which case they are local).

---

Fig. 8. A lorek program that generates 100 times and locations, where the inputs either exercise different branches of the time specification or have locations in different time zones. Location and different-time-zones are primitives in an extensible language. The with syntax defines object fields (we omit a straightforward formalization of objects and fields). The highlighted lines correspond to the test outline and the rest correspond to the service specification.

\[
\begin{align*}
\text{calendarTests} & \quad \text{event}\ 7\text{Object} \\
\text{12hr} & \quad \text{event}\ \text{Name}\ ?\text{String} \\
\text{24hr} & \quad \text{time}\ (\text{gte}\ 0) \\
\text{name} & \quad \text{duration}\ (?\text{Integer}) \\
\text{timeSuff} & \quad \text{startTime}\ (?\text{Option Date}) \\
\text{event} & \quad \text{location}\ (?\text{Location}) \\
\text{duration} & \quad \text{generate}\ \text{testSanity}\ 100 \\
\text{startTime} & \quad \text{loc}\ ?\text{Location} \\
\text{suffix} & \quad \text{constrain-solutions}\ \text{suffix} \\
\text{timeSuff} & \quad \text{cover-regexp}\ \text{time} \\
\text{time} & \quad \text{constrain-solutions}\ \text{loc} \quad \text{different-time-zones} \\
\text{duration} & \\
\end{align*}
\]

\[
\begin{align*}
\text{calendarTests} & \quad \text{event}\ 7\text{Object} \\
\text{12hr} & \quad \text{event}\ \text{Name}\ ?\text{String} \\
\text{24hr} & \quad \text{time}\ (\text{gte}\ 0) \\
\text{name} & \quad \text{duration}\ (?\text{Integer}) \\
\text{timeSuff} & \quad \text{startTime}\ (?\text{Option Date}) \\
\text{event} & \quad \text{location}\ (?\text{Location}) \\
\text{duration} & \quad \text{generate}\ \text{testSanity}\ 100 \\
\text{startTime} & \quad \text{loc}\ ?\text{Location} \\
\text{suffix} & \quad \text{constrain-solutions}\ \text{suffix} \\
\text{timeSuff} & \quad \text{cover-regexp}\ \text{time} \\
\text{time} & \quad \text{constrain-solutions}\ \text{loc} \quad \text{different-time-zones} \\
\text{duration} & \\
\end{align*}
\]

That is, if \( \langle p-a \rangle \) is \( (\lambda\ h\ (\lambda\ M\ (M[h]\ 0))) \), passing numHours returns \( (\lambda\ M\ (M[\text{numHours}]\ 0)) \), an assertion \( \langle A \rangle \) that numHours is non-negative. We introduce the notation \( a(val) \) and \( c(val) \) to denote this.

The embedded language for assertions and the primitive assertions currently come from the specification language for an existing framework in use at the company Amazon. We extend the language so that developers can express additional properties. We discuss the embedded language for constraints \( \langle c \rangle \) and some primitive constraints \( \langle p-c \rangle \) in Section 4.3. Both embedded languages can add to the environment: The embedded assertion language can add variables to \( V \), and the embedded constraint language can add both variables to \( V \) and assertions to \( A \). This is necessary for the powerful enumeration construct we define in Sections 4.4 and 4.5.

Fig. 9. lorek grammar

\[
\begin{align*}
\langle \text{program} \rangle & \ ::= \langle e \rangle^{*} \langle \text{model-gen} \rangle + \\
\langle \text{model-gen} \rangle & \ ::= \text{generate} \ (\langle \text{outline-name} \rangle) \ (\langle e \rangle)^{*} \\
\langle \text{outline-name} \rangle & \ ::= \langle \text{racket-symbol} \rangle \\
\langle e \rangle & \ ::= (\langle \text{declare} \rangle | \langle \text{specify} \rangle | \langle \text{constrain} \rangle) \\
\langle \text{declare} \rangle & \ ::= \text{declare} \ (\langle \text{var} \rangle) \ (\langle t \rangle) \\
\langle \text{specify} \rangle & \ ::= \text{specify} \ (\langle \text{var} \rangle) \ (\langle a \rangle) \\
\langle \text{constrain} \rangle & \ ::= \text{constrain-solutions} \ (\langle \text{var} \rangle) \ (\langle c \rangle) \\
\langle t \rangle & \ ::= \ (\langle \text{var} \rangle) | \ (\langle \text{val} \rangle) \\
\langle \text{var} \rangle & \ ::= \ (\langle \text{racket-symbol} \rangle) \\
\langle \text{val} \rangle & \ ::= \ (\langle \text{symbolic} \rangle) \ | \ (\langle \text{concrete} \rangle) \\
\langle \text{symbolic} \rangle & \ ::= \ldots \\
\langle \text{concrete} \rangle & \ ::= \ (\langle n \rangle) \ | \ \ldots \\
\langle a \rangle & \ ::= \ \text{and} \ (\langle a \rangle) \ (\langle a \rangle) \ | \ \text{or} \ (\langle a \rangle) \ (\langle a \rangle) \\
\langle \text{ite} \rangle & \ ::= \ \text{ite} \ (\langle t \rangle) \ (\langle a \rangle) \ | \ \text{not} \ (\langle a \rangle) \ | \ \langle p-a \rangle \ (\langle t \rangle) + \\
\langle p-a \rangle & \ ::= \ldots \\
\langle c \rangle & \ ::= \ \text{all-of} \ (\langle c \rangle) \ (\langle c \rangle) \ | \ \text{some-of} \ (\langle c \rangle) \ (\langle c \rangle) \\
\langle p-c \rangle & \ ::= \ldots / \text{see Figure 12} \\
\end{align*}
\]

Every \( \langle \text{model-gen} \rangle \) takes an \( \langle \text{outline-name} \rangle \), a number \( \langle n \rangle \), and a body \( \langle e \rangle^* \) and generates a result: either UNSAT, or a mapping from its name to a set of models \( M_1, \ldots, M_n \) that satisfies the environment that \( \langle e \rangle^* \) evaluates to.

### 4.3 Controlling the Space of Solutions

Programmers can define what it means for models to be different using \text{constrain-solutions}. We provide a language of built-in constraints for this (Figures 11 and 12).

By default, Iorek ensures that every generated model is not equal to any other generated model—that is, that at least one symbolic value is mapped to a different concrete value for every pair of generated models.

For example, if we are testing an authentication service with different usernames, we add the following assertion to the environment between subsequent runs of the \text{gen} loop:

\[
(\neq \text{username} \ (\text{evaluate username model})).
\]

The default constraint ensures that the solver does not return the same model twice. When \( n \) is the size of the model space, this amounts to BET using an assertion as a bound. This is not always sufficient for testing: Modern solvers purposely explore close spaces by undoing as few choices as possible (Barrett et al. 2008a,b). That is, if the first username is "aaa", the second username may be "aab", which is likely not different enough from "aaa" to test a different case.

We can instead generate usernames of different lengths:

\[
(\neq \text{string-length username} \ (\text{evaluate username model})).
\]
The enumerate* combinator works as follows: It takes a list of assertions as and a list of constraints cs, and returns a new environment extended with an assertion that chooses among as and a constraint that the next model is different. Each ci in cs describes what it means for two models that satisfy ai in as to be different. A different choice is a model that either satisfies ci, or a different assertion aj from as, where cj describes what it means for two models that satisfy aj to be different. In the simplest case, when every ci is never (the identity constraint), we generate one model for each assertion:

\[(\text{define} \ (\text{enumerate-linear as}) \ (\text{enumerate* as} \ (\text{map} \ (\lambda (a) \text{never}) \ as)))\]

That is, \(\text{(enumerate-linear (list a1 a2 a3))}\) asserts a1, a2, and a3 in some order.

---

3Optimizations are possible for transitive relations (strictly increasing string length, for example).
4The \(*\) denotes dynamic choice, in keeping with Rosette syntax.
Programmers can use \texttt{enumerate*} to define difference for groups of inputs conditionally. For example, we can define numeric inputs as different if they are less than 100 and not equal to each other, or greater than or equal to 100 and at least 50 apart:

\begin{verbatim}
(define (prefer-small x)
  (enumerate*
    (list (< x 100) (≥ x 100))
    (list (relation x not-equal) (relation x (numeric-distance 50))))
)
\end{verbatim}

When the bound \(n\) is the size of the model space, this has the effect of prioritization—that is, if we generate all satisfying positive numbers less than 1000, we are guaranteed to prefer small numbers.

The real power of \texttt{enumerate*} comes from its support for structured data: Programmers can use \texttt{enumerate*} to write constraints over structured data and combine it with other constraints recursively from within the structure. This makes it easy to define what it means to generate a representative set of values for a specification.

For example, suppose we are testing a purchasing service with different items. To determine whether items are different, we consider the structure of the service’s product tree, which places items into categories and subcategories. We want to test our service with a few different items from every subcategory. We define two items as different if they are in different subcategories, or if they are in the same subcategory but have prices that are far enough apart:

\begin{verbatim}
(define (cover-products item tree)
  (match tree
    [(or ((root) l) ((cat _) l))
      (define cs (map (curry cover-products item) l))
      (enumerate* (map (λ (c) true) cs) cs))
    [(subcat l)
      (enumerate*
        (list (in-subcat item l))
        (list (relation item different-price))))]
)
\end{verbatim}

By recursively calling \texttt{cover-products}, we make nested \texttt{enumerate*} choices at every position of the tree. At the subcategory level, we assert that the item must be in that subcategory, with the constraint that any other item in that subcategory must have a different price. At the category level, we recurse to every subcategory and \texttt{enumerate*} the resulting assertions (the item is in that subcategory) and constraints (any other item in that subcategory has a different price). At the root level, we recurse to every category and \texttt{enumerate*} the resulting assertions (choices of categories) and constraints (always choose different categories). When we evaluate this constraint and call the \texttt{gen} loop with the resulting environment, we get what we want: Within a subcategory we generate only items that have prices that are far enough apart, and if we generate all models, then we generate at least one item in every subcategory.

4.5 Implementing \texttt{enumerate*}

We implement \texttt{enumerate*} the same way that we implement other relations, with a single constraint that relates any pair of models.

Rosette makes it easy to choose from a set of assertions. The challenge in implementing \texttt{enumerate*} is asking the solver to make a different choice. When there are finitely many values that satisfy our specification, if we always ask for a different value, we will enumerate
all values (Köksal et al. 2012), but many of these values will be redundant. And we cannot just assert \( \neq a \) (evaluate a M), since this is equivalent to \( \neg a \), which does not tell the solver to pick a different assertion.

This constraint that \texttt{enumerate*} evaluates to (Figure 13) ensures that the solver always picks a different assertion. Consider, for example, enumerating over three assertions:

\[
\texttt{(enumerate* (list a1 a2 a3) (list never never never))}
\]

This will \texttt{choose*} from the three assertions, which will create new symbolic booleans \texttt{b0} and \texttt{b1} and will evaluate to an \texttt{ite} statement in Rosette:

\[
\texttt{(ite b0 a1 (ite b1 a2 a3))}
\]

The \texttt{enumerate*} constraint will extend the environment with \texttt{b0} and \texttt{b1}, and guarantee that the solver maps each of them to some concrete value. Suppose the solver chooses the following assignment:

\[
\texttt{(ite \#f a1 (ite \#t a2 a3))}
\]

Then the assertion for that iteration is \texttt{a2}. The constraint that \texttt{enumerate*} evaluates to tells the solver to choose a \texttt{different assertion} next time:

\[
\texttt{(ite (evaluate b0 M) (~ b0) (ite (evaluate b1 M) (~ b1) b1))}
\]

In other words, next iteration, the solver will either flip \texttt{b0} to \texttt{#t} or flip \texttt{b1} to \texttt{#f}. It will also ensure that no redundant paths are taken. That is, if \texttt{b0} is true, it does not matter what \texttt{b1} is, since for any \texttt{b1}, the resulting assertion evaluates to \texttt{a1}:

\[
\texttt{(ite \#t a1 (ite b1 a2 a3))}
\]

The second part of the constraint guarantees that the solver only flips \texttt{b1} if \texttt{b0} is false, that way Iorek does not generate two satisfying assignments to \texttt{a1}.
Passing the constraints \(c_1\) and \(c_2\) to \texttt{enumerate*} allows the developer to nest constraints (including \texttt{enumerate*} statements) within \texttt{enumerate*} statements. To enumerate all possibilities, the constraints should be \texttt{never} (identity). Otherwise, \(c_1\) is the constraint for the if branch, and \(c_2\) is the constraint for the else branch.

If we always start with \texttt{never}, then nested \texttt{enumerate*} statements are guaranteed to visit each assertion exactly once. This gives us the following theorem about the \texttt{enumerate-linear} function we defined earlier:

\textbf{Theorem 4.1.} For any set of \(n\) distinct assertions, in the empty environment, \texttt{enumerate-linear} visits each assertion exactly once. That is, whenever the resulting environment is satisfiable, there are at most \(n\) models that satisfy it, and each of those models satisfies a different assertion.

\textit{Proof Outline.} The proof follows by induction on \(n\). When \(n = 1\), it is trivial. When \(n = 2\), then \texttt{choose*} introduces exactly one fresh symbolic boolean \(b_0\). By the definition of satisfiability and the fact that there are only two possible values for \(b_0\) (true and false), we visit each assertion exactly once. In the inductive case, for any \(n + 1\), we can define the satisfiability criteria in terms of the criteria for \(n\) with exactly one fresh boolean \(b_1\). Then the models are exactly those from \(n\) with \(b_1\) mapped to false, plus exactly one new model with \(b_1\) mapped to true. The rest follows by definition.

5 STRINGS AND REGULAR EXPRESSIONS

Service inputs are commonly strings bound by regexes. While Iorek spans more than just this application domain, many of our implementation decisions are inspired by it. To make this possible, we extend Rosette with the Racket \texttt{string} and \texttt{regexp} types so that they work with symbolic values. We support some \texttt{regexp} literals by building an interpreter from those literals into SMT-LIB. Rosette is agnostic to the back-end solver, and so Iorek can be used with any SMT-LIB solver with string and regex support (we do not yet support domain-specific solvers like Hampi since Rosette does not support them). In practice we use Z3, which has some string and regex support.\(^5\)

The idea of \texttt{enumerate*} is particularly useful for regex-bounded service inputs. When generating strings that match a regex, we do not want every single string—for many regexes, this would be an infinite set of strings. Rather than generate a finite arbitrary subset of strings or strings up to a given length, the \texttt{enumerate*} combinator allows us to define coverage strategies that generate a representative set of strings \textit{for that regex}. We show in Section 7 that this can increase coverage of the CUT.

5.1 Enumerating Inputs for a Regex

We define a \texttt{cover-regexp} constraint that allows us to generate a representative set of strings for a regex. Consider a simple authentication service in which usernames must match the regex \([a-zA-Z0-9_.]+\). One way to generate usernames that cover this regex is to consider short and long strings, and treat all character classes as different from each other, but each character within a character class as the same:\(^6\)

\begin{verbatim}
"a" matches [a-z], "C" matches [A-Z], "7" matches [0-9], "." matches [_.], "z0aC__137" matches [a-z][a-zA-Z0-9-9]+, "Jx_oMn10" matches [A-Z][a-zA-Z0-9-9]+, "9ax9__123" matches [0-9][a-zA-Z0-9-9]+, ".aAg_0m__" matches [_.][a-zA-Z0-9-9]+..
\end{verbatim}

\(^5\)https://github.com/Z3Prover/z3
\(^6\)In this example, we vary only the first letter of the strings. The real implementation does more than this.
Using `enumerate*`, we can write this constraint as a recursive function on a simple subset of regexes that has just character classes, unions, and repetition:

```
(define (cover-regexp s r)
  (match r
    [(char-class r1)
      (enumerate* (list (matches s r1)) (list never))]
    [(re-union r1 r2)
      (enumerate* (list true true) (list (cover-regexp s r1) (cover-regexp s r2)))]
    [(re-plus r1)
      (enumerate* (list (matches s (re-+ r1))) (list (cover-regexp s r1)))])
)
```

We can use this function as a primitive constraint to cover *any* regex that has this structure.

### 5.2 Why Enumerate Over the Regex Structure?

When generating strings that match a regular expression, a simple and effective approach is to compile the regular expression to a minimal DFA and then enumerate over the structure of the DFA itself. We find `enumerate*` much more expressive than this: It allows us to make explicit the notion of difference we are expressing. Consider a regex for full names which may have prefixes or suffixes:

```
((Ms|Mr|Mrs|Dr)\s+)? [A-Z][a-z]+\s+ ([A-Z][a-z]+\s+)? [A-Z][a-z]+\s+ (Jr|Sr)?
```

The minimal DFA for this regex is complex since there are multiple ways to interpret strings like "Dr John Smith Jr" ("Dr" can be a prefix or a first name). The way the regex is written, its structure (as it is used within the CUT) is much more transparent.

The function we define to cover a regex does not always generate the same set of strings for two regexes that accept equivalent languages. This is by design. The way a regex is written can communicate more information than the language it accepts alone, and two regexes that accept equivalent languages do not always communicate the same intent.

The name regex, for example, calls out prefixes and suffixes separately. For testing purposes, it makes sense to consider strings with different prefix-suffix combinations, as there are few and they may exercise different code behavior. But if we treat character classes as equivalent to unions, then there are infinitely many names. Even if we restrict the length of the names that we generate, we still end up with redundant strings (such as "Joe Smith" and "Jon Smith"). By treating elements within a character class as the same, we generate a small set of interesting inputs.

This is, of course, a heuristic. One major benefit of `enumerate*` is that it makes it easy to define several coverage mechanisms backed by different heuristics (including one that does not distinguish between character classes and unions) and to switch among them easily. The `enumerate*` combinator is also in no way limited to regular domains—many services take JSON as input, and we can use `enumerate*` to cover the structure of a JSON schema.

In practice, `enumerate*` helps us define non-obvious but useful notions of differences for strings. Our default implementations take feedback from potential users into account: We treat character classes differently from unions since character classes express a notion of sameness. We generate strings of a few different lengths when covering an infinite regex to...
find length-sensitive bugs. JSON accepts arbitrary amounts of whitespace; potential users expressed the desire to exercise different amounts of whitespace when generating JSON. The enumerate* combinator makes it easy to express these notions of difference.

5.3 Combining Constraints
Since enumerate* is a combinator, we can combine enumeration with other constraints and take advantage of the power of SMT.

For example, Racket regexps have a notion of matching a single literal character. A simple way to express this to the solver (rather than giving it the entire alphabet of possible Racket characters) is to assert that the string has length one. We can extend cover-regexp to handle this:

\[
\text{(define (cover-regexp s r)}
\text{(match r)}
\text{; ...}
\text{[(any-char) (enumerate* (list (= (string-length s) 1)) (list never))])}
\]

6 INDUSTRIAL CASE STUDIES
We integrated a Iorek prototype into a testing framework at Amazon and used it to find bugs in five real services in development for later use in production. We chose services from a list of services that were being tested by security testers. We selected services with specifications that exercised interesting functionality in our tool, such as bounded numbers (numeric values for dates, for example) and strings that match patterns (paths, account IDs, and so on).

We found four code and specification bugs in three of these services. Developers accepted three of these bugs; the other was identified as an intentional hole in the specification that does not cause unexpected behavior. We found one of the bugs that developers accepted using a custom solution constraint; for the other three bugs, we used the default constraint. In all five services, we tested previously untested behavior.

6.1 Integration and Deployment
We integrated a preliminary version of Iorek into a testing framework at Amazon. The goal of Iorek is to give developers the benefit of a test generation framework without any additional cost, given that developers are resistant to tools that disrupt development flow (Johnson et al. 2013). To use Iorek, the developer simply writes tests as she normally would, but leaves some inputs in the tests blank. We achieved this by reusing existing specifications, writing a Iorek Java front-end, and integrating the front-end with JUnit.

Existing Specifications. Developers at Amazon use a framework that allows them to write input specifications for their service APIs. We wrote an interpreter that translates these specifications (which exist for every service) into Iorek so that the developer does not need to write a specification file (like the file in the example in Figure 2). These specifications are often partial, so we allow developers to add stronger specifications in the test outline, but additional specifications are optional.

Java Front-End. We implemented symbolic inputs using an Input object rather than the ? syntax seen in Figure 3 so that test outlines compile as Java files with no changes to the compiler. We used the builder pattern so that developers do not need to learn new syntax. That is, ?timeSuffix becomes:
new Input<>.Builder(timeSuffix, String.class)
    .withCoverageMode(coverRegexp(time))
    .build();

JUnit Integration. Test outline files run with JUnit with no changes to the build process. Programmers can write test outlines alongside JUnit tests and use the testing patterns they are accustomed to, such as mock objects, which allow developers to stub out sensitive or irrelevant behavior (Mackinnon et al. 2001). To achieve this, we implemented a custom JUnit TestRunner. The prepare phase of the TestRunner temporarily replaces symbolic inputs with dummy objects of the correct types and evaluates the test outline file and service specification to a Iorek program. It then runs the program and evaluates it to a result, which it saves to a JSON file. The run phase runs each test \( n \) times for the specified \( n \), evaluating each symbolic input in a given test outline to the corresponding value in the result.

6.2 Writing Tests
While we are so far the only users, we found it straightforward to use Iorek, even on services with code and specifications that were unfamiliar to us. We used specifications and test files that already existed for these services. We modified each test file to run with the Iorek TestRunner and passed it the name of the specification. The tool fetched the specification from the internal codebase and translated it into Iorek.

We then generalized existing tests, turning concrete inputs that were constrained by the specification into symbolic inputs. When there were an insufficient number of tests, we wrote new test outlines that were similar to existing tests. The specification and existing tests alone made it clear in all but a few cases which inputs were good candidates for symbolic inputs without any need to look at the CUT. Most outlines had just one symbolic input, but some had multiple. When inputs were underconstrained, we added new specifications from within the test, using the names of specifications (for example, accountID) to infer what they represent. We chose a difference constraint that seemed most appropriate for the data type. We defaulted to the not-equal coverage strategy, but also used cover-regexp and different-lengths for strings constrained by regexes as well as a distance metric for numbers.

For each test outline, we started by choosing a small number \( n \) of tests to generate (typically 10). If this produced a bug, then we did not increase \( n \). Otherwise, we increased \( n \) in small increments to a maximum of 100. The largest \( n \) that we needed to produce a bug in the services we tested was 50.

6.3 Running Iorek
We ran the TestRunner on each of the test outline files. The TestRunner spent most of its execution time in the prepare pass, where it translated the outlines into Iorek, queried Z3, and generated inputs. We found the performance overhead reasonable for all of our tests, especially since in practice, \( n \) was small and so there were not many inputs to generate (we evaluate the performance of Iorek for large queries in Section 8). There was no significant performance overhead in the run pass, which ran each outline \( n \) times with the generated inputs.

A simple and effective way to minimize the performance overhead of prepare is to reuse saved inputs when there are no changes in a test (rather than querying the solver again). This can also help with regression testing. The framework already generates inputs as JSON before parsing them back into Java, so this is just a matter of implementation.
6.4 Finding Bugs

In total, we wrote 42 test outlines, 35 of which were generalizations of existing tests. Of the four bugs we found, two were from generalizing tests and two were from creating new tests. Three used additional specifications, although we could avoid adding specifications for two of these three by providing default specifications for common service input types (for example, account IDs).

In three of the bugs we found, we used the default solution constraint, while in one we used a custom solution constraint (cover-regexp). While we did not evaluate these services with any other tools, we suspect that existing BET tools could find the bugs that we found using the default solution constraint, since the default constraint amounts to BET when $n$ is the size of the input space: It generates all $n$ inputs that satisfy the assertion. We suspect that existing BET tools would not find the bug that we needed a custom solution constraint to find, since the custom constraint is beyond the scope of what BET tools can express.

Two of the bugs we found were in services that handled some inputs poorly, and two were in services that were overly lenient. We found untested scenarios in all five services. We discuss these bugs and scenarios below.

Poorly handled inputs. Two of the bugs that we found were in services that did not correctly handle some inputs. For example, one service takes a string which represents a colon-delimited path. We generalized an existing test and changed this input to a symbolic input. Using cover-regexp, we found that the service handles most characters correctly, but crashes on paths that contain underscores. This bug has been accepted by the developer.

Overly lenient services. Two of the bugs that we found were in services that took inputs that cannot be valid. For example, one service takes a string identifier that ought to be numeric. The identifier is used widely enough in other services that we were able to infer a Iorek specification for it. In doing so, we found that the service accepted invalid identifiers. The developers have accepted this bug.

Untested scenarios. In all five services, we exercised scenarios that were not yet tested. Some of these scenarios may be useful for regression testing. For example, one service restricts an input to one of over thirty Enum values. Using Iorek, we were able to automatically test all inputs. Another service correctly errors on an empty input. A test case expressing this as desired may be useful.

7 CODE COVERAGE EVALUATION

We used Iorek to generate string inputs for the random testing tool Randoop (Pacheco and Ernst 2007) to evaluate the effectiveness of Iorek at increasing code coverage. We focused on strings because they are common in services and exercise interesting Iorek functionality. We found the following:

1. Iorek increased code coverage. The inputs that Iorek produced increased the coverage of Randoop tests on 15 of 18 benchmarks.
2. Specifications were useful. Using a specification to generate strings increased coverage on 14 of 18 benchmarks.
3. Having multiple coverage strategies was useful. On 4 benchmarks, not-equal performed best; on 5 benchmarks, cover-regexp performed best.
7.1 Using Randoop

Randoop randomly generates sequences of method calls, executes these sequences, and uses feedback to automatically produce unit tests for Java programs. Randoop has limited built-in string support: It relies on simple strings and on hard-coded test strings in the CUT. We wrote a small (24 LOC) Racket program that runs Iorek programs and compiles the results into Randoop inputs.

We chose Randoop because it is an established tool with an interface for extending input generation. This made it possible to measure the effect of using Iorek to generate inputs while controlling for expertise in writing test outlines. We chose Randoop over enumeration tools like Kaplan (Köksal et al. 2012) and SciFe (Kuraj et al. 2015) since we focused our implementation efforts on constrained strings, which those tools do not support. We did not compare to BET tools such as Korat (Boyapati et al. 2002) since Iorek’s not-equal strategy with maximal \( n \) amounts to BET using an assertion as a bound.

7.2 Evaluating Coverage

We selected our string benchmarks\(^7\) from prior testing work (McMinn et al. 2012; Shahbazi and Miller 2016). We ran off-the-shelf Randoop for a minute for each benchmark without Iorek strings to produce baseline tests.

We wrote a basic string regex to control for the effect of generating unconstrained strings:

\[
\text{(declare basic-string #rx"([a-zA-Z0-9-]|/|\|;|_|,|=|=|=|@|!|'|%|#| (\.)| )")}
\]

While this is more restrictive than ".*", it is important for sane results, since querying Z3 for strings with no specification at all only returns sequences of null characters.

For each benchmark, we wrote two regexes: a black-box regex to demonstrate model-based input generation and a white-box regex to demonstrate the plausibility of using Iorek for white-box input generation. We found all of the black-box regexes on free online sources (such as StackOverflow and RegExLib) using only descriptions of the datatypes and comments in the CUT, and modified them only as necessary to run Iorek. We manually inspected the CUT to write white-box regexes that were likely to exercise different cases.

For example, one benchmark validates 24 hour times. We found the black-box regex by searching RegExLib, and we wrote our white-box regex after inspecting the CUT and finding that it special-cases zero, invalid characters, and invalid times:

\[
\text{black-box}
\begin{align*}
(\text{declare hh #rx"((\[0-1\]\d)|(2[0-3])")} \\
(\text{declare mm #rx"[0-5]\d"}) \\
(\text{declare 24Hour (concat ":" hh mm)})
\end{align*}
\]

\[
\text{white-box}
\begin{align*}
(\text{declare hh #rx"(0|1-2|3-9|X)(0|1-3|4-9)"}) \\
(\text{declare mm #rx"(0|1-5|6-9)(0|1-9)"}) \\
(\text{declare 24Hour (concat ":" hh mm)})
\end{align*}
\]

We then produced as many strings as we could in a minute using the following regexes and strategies:

1. Basic string regex with not-equal
2. Black-box regex with not-equal
3. Black-box regex with cover-regexp
4. White-box regex with not-equal
5. White-box regex with cover-regexp

For example, we wrote the following Iorek program to generate strings covering the black-box time:

\[
(\text{define-program 24HourBlackbox} \\
(\text{... black-box regex from above, which can produce at most two inputs with cover-regexp}) \\
(\text{generate testCoverRegExp 2 (constrain-solutions time (cover-regexp 24Hour)))})
\]

\(^7\)We did not evaluate the LGOL benchmark because the code was no longer available.
This produced a file with the strings "02:04" and "20:22". Each Iorek program produced an input file; we ran Randoop for a minute for each file and measured line coverage of generated tests using JaCoCo.\(^8\)

### 7.3 Results

We summarize our results in Figure 14. Iorek enhanced the performance of Randoop on 15 of the 18 benchmarks for both black-box and white-box testing. This effect was not just from generating many strings—using a regex specification increased coverage compared to generating underconstrained strings for 14 of the 15 benchmarks. Having multiple coverage strategies for the same specification was useful, as no single coverage strategy performed uniformly best: Both coverage strategies performed equally well in 9 of 18 benchmarks, \texttt{not-equal} with a specification outperformed \texttt{cover-regexp} in 4 benchmarks, and \texttt{cover-regexp} outperformed \texttt{not-equal} in the remaining 5 benchmarks. The performance of different coverage strategies depended on the specification and the CUT.

**Hard-coded Strings.** The CUT included hard-coded test strings in 4 benchmarks. Randoop used these test strings to generate tests and performed well in all 4 cases. Using Iorek provided no additional benefit in 3 of these cases. Iorek was still sometimes useful even when these hard-coded test strings existed: For the 4th benchmark, Iorek produced International Bank Account Numbers that passed a validity check, which the hard-coded test strings in the code failed to do.

**No Best Strategy.** Neither coverage strategy performed best in 5 of the benchmarks that did not have hard-coded strings. In 1 of these benchmarks, even \texttt{not-equal} without a specification produced a single valid year string that hit the only case necessary to achieve full coverage. In the other 4, the \texttt{not-equal} strategy always produced more strings than

\(^8\)http://www.jacoco.org/jacoco/trunk/index.html
cover-regexp in a minute, but both strategies produced strings for either the black-box or white-box regex that exercised enough cases to achieve the maximum coverage possible (excluding private and protected methods, which Randoop does not call).

**Hide Cases.** The not-equal strategy outperformed the cover-regexp strategy in 4 benchmarks. In these cases, the black-box regex hid information about cases that the CUT treated differently and that were not obvious from the datatype itself (for example, "00:00" in 24 hour times). The white-box regexes were able to generate these invalid cases using both strategies.

**Reveal Cases.** The cover-regexp strategy outperformed the not-equal strategy in 5 of the benchmarks. In these cases, one or both of the regexes revealed important semantic information about the CUT. The not-equals strategy did not always manage to produce strings that exercised these different cases despite producing more values than cover-regexp. For example, the black-box regex we found for an International Standard Book Number is \((97(8|9))?\d\d\d\d\d\d\d(\d|X)\). This handles different cases in combination with each other that the code treats differently: values that end in X, values that end in digits, 10 digit values, and 13 digit values. The cover-regexp strategy achieved 77.0% coverage with 6 inputs, while the not-equal strategy achieved only 63.5% coverage with 544 inputs. The white-box regex further considered invalid lengths and characters as well as dash-separated values—white-box cover-regexp achieved 91.9% coverage in 73 values while white-box not-equal did not improve.

### 8 SOLVER QUERY PERFORMANCE TRADEOFFS

By default, to generate \(n\) inputs, Iorek's generation loop (Figure 7) queries the solver \(n\) times, asking for values one at a time with \(n\) total constraints. We compared this to three other approaches for evaluating Iorek's satisfiability criteria (Figure 6):

1. Using \(\forall \forall\):
\[
\forall 1 \leq i \leq n, A(v_i) \land \forall 1 \leq j \leq n, (i \neq j \Rightarrow C(v_i, v_j))
\]
Using `distinct?` when possible:

\[ \text{distinct?}(v_1, \ldots, v_n) \]

(3) All at once, in a single query with \( n^2 \) constraints:

\[ C(v_1, v_2) \land C(v_1, v_3) \land \ldots \land C(v_{n-1}, v_n) \]

We used Z3 as the solver and set a timeout of 600 seconds. When covering a regexp, we used a constraint that treats + the same as concatenation ("aaa" is always distinct from "aaaa" and so on) so that we could generate many values for the sake of evaluation.

Iorek’s generation loop is the only approach we found that scaled to large and complex queries. Using “∀” timed out for the smallest and simplest query. Using `distinct?` was by far the fastest approach for simple queries when it was possible, but it was very slow when it required Z3 to invert a function, and it could express only two of four queries. Asking for values all at once performed well for small \( n \) for simple queries, but timed out for complex queries and for large \( n \).

Given that the Iorek semantics are independent of the encoding, it may be worth using heuristics to switch between different encodings for simple queries and small \( n \). Furthermore, it is possible that different solvers have different optimal encodings for the same queries—while we are yet to investigate this, it may also be worth switching encodings based on what is optimal for a particular solver. We summarize the trade-offs of the quantifier-free approaches with Z3 below.

`distinct?` The simplest query (Table 1) asked for \( n \) different Java integers. For this query, using `distinct?` was by far the fastest for all \( n \). However, using `distinct?` to ask for many strings of different lengths matching a regex timed out for all queries (Table 3), likely because this query required Z3 to invert a function (\texttt{str.len}). The other two queries cannot be expressed using `distinct?`. Using `distinct?` appears to be optimal when it can express a query and when it does not require Z3 to invert a function.

All at once. For both of the integer queries, asking for values all at once was faster than one at a time for small values of \( n \), but slower for large values of \( n \) (Tables 1 and 2). For both of the string queries, all at once timed out for all queries (Tables 3 and 4). For simple queries with small \( n \), it was faster to ask for values all at once rather than one at a time. Given that strings are new to Z3, we suspect that heuristics are yet to be implemented, and so string queries are currently more complex than integer queries.

One at a time. For both of the integer queries, asking for values one at a time was slower than all at once for small \( n \), but outpaced all at once when \( n \) became sufficiently large (Tables 1 and 2). For both of the string queries, one at a time was uniformly the fastest across all \( n \) (Tables 3 and 4). One at a time was the only approach that scaled to all four queries, and was optimal for large \( n \) and complex queries.

9 RELATED WORK

We have already discussed angelic execution and synthesis (Bodik et al. 2010; Solar Lezama 2008), relational logic solvers (Torlak and Jackson 2007), BET tools and generators (Boyapati et al. 2002; Goodenough and Gerhart 1975; Khurshid and Marinov 2004; Rosner et al. 2014; Senni and Fioravanti 2012), DSLs for property-based testing (Claessen and Hughes 2000; Lampropoulos et al. 2017), test generation and symbolic execution tools (Anand et al. 2007; Boyapati et al. 2002; Cadar et al. 2006; Chipounov et al. 2011; Fraser and Arcuri 2011; Godefroid et al. 2005; Pacheco and Ernst 2007; Sen et al. 2005; Tillmann and De Halleux
2008), and Grammar-Based Whitebox Fuzzing (Godefroid et al. 2008) in Section 3. Here we go into some more detail and discuss other related work.

**Solver-Aided Languages.** Solver-aided languages use constraint solvers for synthesis, angelic execution, verification, and debugging. Rosette (Tolak and Bodik 2013) is a framework for building these languages. We extend Rosette with strings and regexes and implement our language in Rosette.

Iorek is not the first solver-aided language for testing. PBNJ (Samimi et al. 2013) helps programmers write mocks for testing. Iorek generates inputs which can be used alongside mocks.

**Enumeration.** SciFe (Kuraj et al. 2015) is a language for enumerating structures. SciFe provides dependent enumerators that can be used structurally to rule out redundant inputs. Iorek’s bounded enumeration combinator is similar in expressiveness to dependent enumerators. Unlike SciFe, Iorek can express this style of enumeration to any SMT solver.

Kaplan (Köksal et al. 2012) is a general-purpose constraint programming extension of Scala. Kaplan also gives programmers control over the solution space. Kaplan can be used for exhaustive and ordered enumeration. In contrast with Kaplan, Iorek’s constructs for control over the solution space are declarative. Kaplan also does not contain the notion of a representative set or support constraints on strings.

**Languages for Test Input Generation.** Iorek is a DSL for test input generation, and is thus similar to the property-based testing DSL QuickCheck (Claessen and Hughes 2000). Iorek and QuickCheck both enable the programmer to control the generation of inputs. However, they do so in different ways: QuickCheck programmers write probabilistic constructive generators which assign weights to input possibilities, while Iorek programmers write declarative generators which abstract details of low-level datatypes and query SMT.

These approaches have tradeoffs: Iorek may be more appropriate for generating tests that use SMT-supported datatypes, since using Iorek lessens the developer burden of reasoning about those datatypes. QuickCheck may be more appropriate for generating tests that do not make heavy use of those datatypes or that require guarantees on the probability distribution of inputs. Incorporating QuickCheck and Iorek into the same tool would simplify writing generators for SMT-supported datatypes and give the developer fine-grained control over inputs both probabilistically and using SMT.

The DSL Luck (Lampropoulos et al. 2017) makes it easier to write probabilistic generators for QuickCheck-style testing. Like Iorek, Luck is a DSL for test input generation that interacts with constraint solvers. However, Luck uses constraint solvers for efficiency, not for controlling the distribution of generated inputs. In contrast, Iorek uses constraint solvers to control the space of generated inputs and allow programmers to write non-probabilistic generators. Luck also leaves compatibility with off-the-shelf solvers to future work.

**Testing and Fuzzing.** Equivalence partition testing is a technique that divides inputs into classes that are equivalent according to some property and does not generate multiple equivalent inputs (Ntafos 1998). Recent work (Anand et al. 2006) uses symbolic execution and model checking to rule out equivalent cases. Iorek can be thought of as a language for equivalence partition testing. Unlike existing approaches, Iorek allows the programmer to define the notion of input equivalence, combine this notion with predicates about the input, and pass the resulting constraints to any SMT solver.

Reggae (Li et al. 2009) introduces a technique that helps existing automated testing and symbolic execution tools produce better inputs for strings constrained by regexes. Iorek can
also be used for this purpose (we demonstrate the use of Iorek with Randoop in Section 7), but is not limited to strings and regular domains. Furthermore, Iorek provides a simple IR that can be used with multiple testing tools and that gives programmers granular control over how it generates these inputs.

The Iorek implementation at Amazon uses existing service specifications to generate inputs. In this way, it is similar to model-based testing tools, which use a specification of the code to generate tests (Dick and Faivre 1993). Recent work in model-based parametric testing (Calamé et al. 2007) combines this approach with a constraint-solving procedure. Unlike these approaches, Iorek is agnostic to whether specifications come from a model of the code or the CUT. Iorek can also be used with any SMT solver.

Field-exhaustive testing (Ponzio et al. 2016) uses incremental SAT solving to generate fewer, higher quality inputs than BET. Iorek expresses a similar style of coverage, except that it is not specialized to objects and uses an SMT solver to handle strings and other theories.

Fuzzers automatically produce interesting inputs to trigger invalid program states, often to test security vulnerabilities. SAGE (Godefroid et al. 2008) uses symbolic execution to find constraints to generate inputs that exercise different paths. AFL-Fuzz uses branch coverage to generate interesting inputs. Fuzzers are meant to be used without any guidance; in contrast, Iorek is meant to be used in a testing context to help the programmer write better tests, and gives the programmer control over what constitutes an interesting input.

Guiding Solvers. Many existing works guide solvers to optimize performance, including CEGIS (Solar Lezama 2008), the synthesis algorithm SKETCH uses to efficiently solve nested quantifiers. Recent work evaluating CEGIS (Jha and Seshia 2014) investigates the impact of different kinds of counterexamples on the performance of the algorithm. Metasketches (Bornholt et al. 2016) expose a way of controlling the space of solutions to solve the problem of optimal synthesis. Iorek’s constraints provide a general way to guide the solver over a sequence of queries to find an interesting set of inputs. This is similar to recent work using solvers for sampling and counting problems (Meel et al. 2015), except that it is more general.

10 CONCLUSIONS AND FUTURE WORK

We designed and implemented a solver-aided language and testing framework to tackle the tedious process of generating test inputs. Our tool Iorek generates many satisfying solutions (inputs) and empowers the programmer with the ability to express how these solutions differ from each other. We implemented the Iorek back-end in Rosette and exposed a rich language for constraining how inputs differ. In this language, we introduced a bounded enumeration combinator that makes it easy to define a flexible notion of a representative set of values for a structure and use an SMT solver to generate those values. We demonstrated the power and flexibility of this combinator to generate strings. We showed that Iorek is effective at finding bugs in an industrial setting, can increase code coverage of a random testing tool, provides useful flexibility, and scales to large and complex queries.

10.1 Future Work

Future work could use constraints on the solution space (such as the existence of a unique solution) to optimize the performance of a solver in finding a single solution. Future evaluations could include Iorek performance with solvers other than Z3, comparison with

9http://lcamtuf.coredump.cx/afl/
fuzzers, and a user study on writing difference constraints. Possible extensions to Iorek include synthesizing inputs from examples (Gulwani 2012), deriving white-box specifications, test input prioritization for unbounded model spaces, Reggae integration, QuickCheck integration, and applying Iorek to other domains.

11 ACKNOWLEDGEMENTS

We thank Emina Torlak for help with our contributions to Rosette. We thank Nikolaj Bjørner for prompt fixes to bugs in the development version of Z3. We thank Rohin Shah for help formalizing the enumerate* combinator. We thank Benjamin Keller for help integrating Iorek into Randoop. We thank Neha Rungta, Byron Cook, and the UW PLSE lab for feedback on early revisions. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256082. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES


Susmit Jha and Sanjit A. Seshia. 2014. Are there good mistakes? A theoretical analysis of CEGIS (3rd Workshop on Synthesis (SYNT)).


Phil McMinn, Muzammil Shabbaz, and Mark Stevenson. 2012. Search-Based Test Input Generation for String Data Types Using the Results of Web Queries (International Conference on Software Testing, Verification and Validation).


Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. 2013. Declarative Mocking (ISSTA).


Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette (Onward!).


