RI14008 2 June 2014 Computer Science

IBM Research Report

Test Generation from Business Rules

Simon Holm Jensen
Samsung Research America
USA

Suresh Thummalapenta
Microsoft Corporation
USA

Saurabh Sinha
IBM Research – India

Satish Chandra
Samsung Research America
USA

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

ABSTRACT

Enterprise applications are difficult to test because their intended functionality is either not described precisely enough or described in cumbersome business rules. It takes a lot of effort on the part of a test architect to understand all the business rules and design tests that "cover" them, *i.e.,* exercise all their constituent scenarios. Part of the problem is that it takes a complicated set up sequence to drive an application to a state in which a business rule can even fire. In this paper, we present a business rule modeling language that can be used to capture functional specification of an enterprise system. The language makes it possible to build tool support for rule authoring, so that obvious deficiencies in rules can be detected mechanically. Most importantly, we show how to mechanically generate test sequences—*i.e.,* test steps and test data—needed to exercise these business rules. To this end, we translate the rules into logical formulae and use constraint solving to generate test sequences. One of our contributions is to overcome scalability issues in this process, and we do this by using a novel algorithm for organizing search through the space of candidate sequences to discover covering sequences. Our results on three case studies show the promise of our approach.

1. INTRODUCTION

A business rule articulates some aspect of the expected functional behavior (or a *requirement*) of an enterprise application. Here is a simple business rule that determines how an invoice total is determined in a billing application:

> The *Balance Type* of a customer affects how invoice total is computed; it can be one of the following:
>
> *None*: The customer's account will not hold a balance; instead all charges accrued in an order will be included in the next invoice;
>
> *Credit*: The customer's account may accrue charges up to the set credit limit. Charges will automatically be paid from the users credit pool until the set limit is reached. Users are responsible for paying their credit debt as well as any overages.

We will examine this rule closely later; for now, suffice it to say that the requirements of an enterprise system are typically captured by a large number (often, hundreds) of business rules such as the one above.

It is reasonable to expect functional testing of an enterprise system to *cover* its business rules, which is to say, testing would exercise every distinct scenario described in each of its business rules. For example, in the preceding rule, one of the scenario to be exercised is that a customer's balance type is credit, and that the order amount exceeds the customer's credit limit. A test that exercises this scenario would set up a customer with the balance type as *credit* as well as a certain credit limit, say, *100*, create an order and add items to the order to bring a total to, say, *120*, to exceed the credit limit for that customer, and then finally create an invoice for that order,and verify the invoiced amount. Figure 1 illustrates this flow. Although the values *credit*, *100*, and *120* can be identified just from this rule (by constraint solving), identifying a test sequence is also important to apply those values at the right fields on the appropriate screens.[1]

It requires a carefully thought-out test scenario, *i.e.,* a sequence of test steps as well as associated test data, *i.e.,* values to be entered in the relevant test steps to exercise a business rule, or a scenario thereof. Without systematic test creation, testers may end up creating multiple tests that exercise the same business rule, or a scenario thereof, over and over again without any additional benefit (especially if they are incentivised by the number of tests created rather than quality of the created tests), and more problematically, may neglect to create tests for some other business rules.

In practice, due to time pressures, testers are more often ad-hoc than methodical in creating test scenarios and test data. One part of the problem is that a realistic system might have hundreds of business rules written out in plain text, and it is difficult to grasp a global view of how the rules together describe the application behavior. A related part of the problem is that it may need complex reasoning to piece together test sequences that would cover each scenario. The net result is that, despite a lot of resources spent in testing, bugs still escape into the field.

Our vision is to make testing of enterprise software more tool-based, by adapting technology developed for automated and systematic test generation for programs. In this vision, business rules would be written in a structured notation that allows mechanized analysis. Special editors could be created to enable non-programmers to capture business rules in a structured notation; this is an independent challenge in *end-user* programming. A tool would validate business rules and point out any ambiguities or omissions that it can detect. After the business rules pass validation, another tool would generate test sequences and test data to exercise the application thoroughly as well as without redundancies.

We have built a system to partially fulfil this vision. In the rest of this introductory section, we give an overview of our system, describe some of the challenges in automating test generation for covering business rules, and briefly summarize our results.

1.1 An Overview

Enterprise systems of interest to us are transaction-oriented, which means that they consist of a set of transactions or operations (*e.g.,* create a customer, add an item to an order, and so on) that operate on databases. A business rule applies to a particular operation supported by the system. Formally, a business rule describes the relation between the database state before and after the operation. Figure 2 shows formalization of the business rule quoted informally at the beginning of the introduction. It says that the operation refers to an invoice record, `inv`, and modifies specific attributes of `inv`. There are three scenarios that occur in this rule. The first scenario applies when the customer to which the invoice refers has balance type `None`; the *precondition* is shown on the left of the first arrow. Note that the invoice has references to the customer to which this invoice pertains and an order created in the system; in a relational database, these would be foreign keys in the customer and order tables. The second scenario applies when the customer has balance type `Credit` and the credit limit is sufficient to cover the order total. The third scenario applies when the customer has balance type `Credit`, but the order total exceeds the credit limit. In each scenario, the effect of the operation is to compute invoice total and update the customer's residual credit limit; this effect, or *postcondition* is shown to the right of the arrow in each case. Note that business rules refer to the state observable at transaction boundaries; intermediate program states encountered in the implementation while a transaction is in process, are not important to business rules.

Covering a business rule means to exercise each of its constituent scenarios, referred henceforth as *rule parts*. To cover these three rule parts in the business rule of Figure 2, it is necessary to create (separately) each of the three preconditions and, in each case, verify that the postcondition holds after the operation has completed.

[1]Not all fields on the screens in Figure 1 are constrained from the point of view of exercising a particular scenario, but the application might still demand sensible values for them. The tester is expected to make these up.

Figure 1: Test sequence for exercising a business rule from the `jBilling` application.

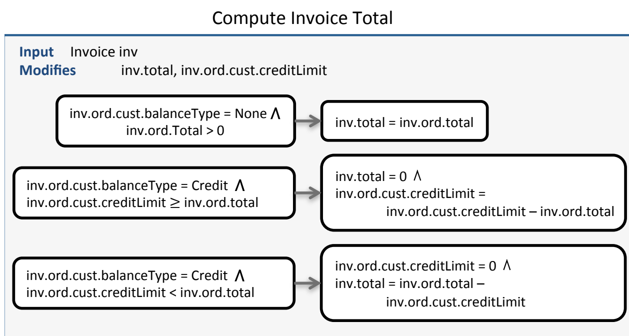

Figure 2: Business rule for computing invoice amount.

This brings us to the main difficulty in creating tests: appropriate database state, such as a customer with a certain balance type and an order with a certain amount, needs to be established before any of these scenarios can be exercised. We showed in Figure 1 the steps that would be required to create these preconditions. How can we identify these steps, and the data to be entered in each of these steps automatically to cover a rule part of a business rule?

Our observation is that the steps that are required to drive the database state to a desired precondition are carried out by operations, and those operations too would have rules that specify their functionality. We could then use business rules as "state transformers" and piece together a sequence of operations to arrive at a desired state. The advantage of looking at business rules as state transformers is that we can adapt the technology developed for test generation on *programs* for the problem at hand.[2] The disadvantage of relying on business rules to act as state transformers is that they need to be specified to a certain level of detail for them to work out as state transformers; this is generally not a big problem—practioners tend to write business rules with an intention to be complete—but their intended use as input to test generation process does increase expectations from the rules and, therefore, from the analysts who write them.

1.2 Our Approach and Results

At a high level, the idea is to use backward analysis to piece together a sequence of operations to arrive at a desired state. We look for an operation whose business rule has a rule part whose postcondition would imply the desired precondition. Such an operation, if executed in a way that that specific rule part applies, would establish the desired state. The operation may require some user-provided values, but may partially rely on prior database state. The process is repeated until no prior database state is assumed—that is, all the database state is established by operations identified in the process.

Consider the second rule part of the rule shown in Figure 2 for the operation to generate an invoice. To satisfy the precondition of the rule part, a customer with balance type `Credit`, and an associated credit limit, needs to be created first. Then, an order whose total does not exceed the customer's credit limit needs to be generated, which involves adding items with suitable prices to the order. Only after this state has been set up, the operation for invoice generation can be invoked. An operation sequence and test data (we explain the notation in Section 4) that achieves this is:

```
State st; BalanceType bt = Credit; int crLimit = 100, price = 20;
Customer cust = CreateCustomer(st, bt);
Customer cust1 = AddCreditLimit(cust, crLimit);
Order ord = CreateOrder(cust1);
Item item = CreateItem(int price);
Order ord1 = AddItemToOrder(ord, item);
Invoice inv = GenerateInvoice(ord1);
```

The idea of backward traversal is definitely not novel; it is reminiscent of weakest preconditions. Our contribution is to make this idea work in the context of business rules. In general, the space of possible operation sequences can be large, in which only a few sequences cover the target rule part. Thus, the challenge is to search

[2] We clarify, though, that business rules are themselves not executable programs; rather they only are an abstract description of the functionality of a program.

this space soundly, but efficiently in a goal-driven manner. Specifically, our technique builds the sequence incrementally using constraint solving. If the logical formula for a sequence is not satisfiable, it extracts the unsatisfied core of the formula and constructs new candidate sequences by considering only those operations and rule parts whose postconditions are compatible with the unsatisfied core. In this way—and using additional optimizations—the technique can prune out large parts of the search space and efficiently narrow down to the covering sequences.

We also present a notation for capturing business rules formally, which enables mechanized analysis for test generation. Moreover, prior to test generation, formally specified rules can be checked for various consistency and completeness properties.

We have implemented a prototype system, which includes a business rule editor (an Eclipse plug-in) and automated analyses for rule checking and test generation. Our preliminary results illustrate the promise of the approach: for 77 rule parts, modeled from three applications, our technique generated covering sequences and test data for 99% of the rule parts and missed only one rule part (which could not be covered because of limitations of the underlying constraint solver). By comparison, a technique that performs exhaustive (unguided) search could cover 74% of the rule parts, although it explored substantially more candidate sequences than our technique.

Contributions. The contributions of this paper are as follows:

- We describe a notation for describing business rules formally, and articulate a set of well-formedness properties that can be checked mechanically. We have also developed an Eclipse plugin for our rule language.
- We describe an algorithm that can mechanically construct test sequences that exercise the business rules of a model. The algorithm uses a novel optimization to prune the search space. We have implemented the algorithm in a tool called BUSTER.
- To evaluate our approach, we have formalized the business rules of three enterprise systems and used BUSTER to generate test sequences. Using our approach we are able to generate tests for 99% of the rule parts.

In Section 2, we describe our notation for modeling business rules and present four well-formedness properties of models. In Section 3, we present the running example model, JBilling. In Section 4 we describe our approach including several optimizations. In Section 5 is the experimental evaluation using our implementation BUSTER and in Section 6 we discuss related research.

2. RULE MODELING AND CHECKING

In this section, we discuss the notation for modeling business rules and the static checking for completeness and consistency performed on formally captured rules.

2.1 Rule-Modeling Language

Overall, our approach models rules in the context of operations in the system under test. An operation is described by a set of input entities, a set of created entities, a set of modified entities, and a set of rules, where each rule consists of a set of precondition-postcondition pairs. For example, Figure 2 shows the operation for computing invoice totals in the `jBilling` application. The rules associated with this operation, which govern how the invoice total and the customer's credit limit are updated, are modeled with the operation in the form of precondition-postcondition pairs.

```
RuleSpec    ::= Entities Operations Triggers

Entities    ::= Entity Entities | ϵ
Entity      ::= Enum | Object
Enum        ::= enum ID { EnumVals }
EnumVals    ::= ID EnumVals | ϵ
Object      ::= object ID { VarDecl }

Operations  ::= Operation Operations | ϵ
Operation   ::= operation ID { Input Creates Modifies Rules Next }
Input       ::= input : VarDecl
Creates     ::= creates : VarDecl
Modifies    ::= modifies : VarDecl
Rules       ::= Rule Rules | ϵ
Rule        ::= group ID { RuleParts }
RuleParts   ::= RulePart RuleParts | ϵ
RulePart    ::= rule ID { pre : Expr ; post : Expr }
Next        ::= next : ID | ϵ

Triggers    ::= ID → ID Triggers | ϵ

VarDecl     ::= TypeName : ID VarDecl | ϵ
TypeName    ::= bool | int | float | string | set<TypeName> | ID
Expr        ::= ...
```

Figure 3: Partial rule-modeling syntax.

Figure 3 presents the formal syntax of the rule-modeling language (for clarity, we omit some of the details and present only the important parts of the language). A *rule specification* consists of entities and operations. An *entity* can be an object in the system (*e.g.,* invoice, order, customer) or an enumerated type (*e.g.,* a customer's balance type can be `None`, `Credit`, or `Prepaid`).

The key part of the syntax, which models rules, is based on operations. Formally, an *operation* O is the tuple (I, C, M, R, o_n), where I is the set of input entities read during the execution of O, C is the set of entities created by O, M is the set of entities whose attributes are modified by O, R is the set of rules that describe the behavior of O, and o_n is a succeeding operation that, if specified, is the only operation that can execute after O.

The notion of *succeeding operation* can simplify the modeling of "coarse grained" operations that have many rules associated with them. Such an operation can be broken down into a sequence of finer-grained operations—with the sequence specified via the `next` clause—that have simpler rules. The specification lets the test-generation algorithm ensure that the atomic nature of the coarse-grained operation is preserved in the finer-grained sequence: while chaining operations, the algorithm avoids interleaving other operations in the middle of a finer-grained operation sequence. Moreover, such decomposition is necessary when the rules of an operation have data dependences, which define an implicit ordering among the rules.

A *rule* $R = \{r_1, r_2, \ldots, r_k\}, k \geq 1$, consists of a set of rule parts. A *rule part* r is a precondition-postcondition pair, $p \implies q$, where p and q are boolean formulas such that if p holds in the state before the operation, q is true in the state resulting from the execution of the operation. If the precondition of a rule part is true, we say that the rule is *applicable*.

The rule shown in Figure 2 has three rule parts, each of which consists of a precondition and a postcondition. The first rule part pertains to the case where the customer's balance type is `None`; the second rule part is for the case where the balance type is `Credit` and the credit limit exceeds or equals the order total; the third rule part covers the case where the balance type is `Credit` and the order total exceeds the credit limit.

Often in enterprise systems, the execution of an operation or a transaction automatically triggers other transactions or operations. For example, in `jBilling`, customers with balance type `Prepaid`

have the option of getting their prepaid limit automatically recharged if it falls below a threshold amount. Our rule-modeling syntax accommodates this via the `Triggers` clause, using which the triggers relation between a pair of operations—where the first operation automatically triggers the second operation—can be specified.

The primitive types include boolean, integer, float, and string types. The model also accommodates sets of these types.[3]

2.2 Rule Checking

We define a few well-formedness properties on rule specifications to ensure consistency and completeness, and that also facilitate operation chaining for test generation. These properties are amenable to mechanical checking. Thus, we envision that the rules can be iteratively refined in a rule editor, based on automatic semantic checking for property violations (in addition to syntactic checking for conformance to the modeling syntax).

Property 1: Rule-part Disjointedness. In our notation, a rule part is intended to represent disjoint preconditions so that when a rule is applicable, the precondition of only one rule part is true; consequently, there is no ambiguity in identifying the relevant rule part for an applicable rule. Formally, we define this property as follows. Let $R = \{r_1, r_2, \ldots, r_k\}$ be a rule such that $k \geq 2$. Then, for all $r_i, r_j \in R$ where $r_i := (p_i \implies q_i)$ and $r_j := (p_j \implies q_j)$, $(p_i \wedge p_j)$ must not be satisfiable. A simple example of a rule specification that violates this property is $p_i = (a > 0)$ and $p_j = (a < 10)$; this specification represents ambiguous behavior when, for example, $a = 5$. The rule parts illustrated in Figure 2 have disjoint preconditions.

To check that a rule satisfies this property, first, we enumerate all pairs of rule parts. Then, for each pair of rule parts (with preconditions p_i and p_j), we determine whether the boolean formula $(p_i \wedge p_j)$ has a solution; if it does, we flag a violation of the property.

Property 2: Rule-part Completeness. This property is intended to ensure that a rule specifies the complete operation behavior for the variables mentioned in the rule. Let $R = \{r_1, r_2, \ldots, r_k\}$ be a rule such that $k \geq 2$ and $r_i := (p_i \implies q_i)$. Then, $\neg(p_1 \vee p_2 \vee \ldots \vee p_k)$ must not be satisfiable. For example, the rule consisting of two rule parts with preconditions $(a < 5)$ and $(a > 10)$, respectively, violates the completeness property because the operation behavior for $5 \leq a \leq 10$ is left unspecified. To verify this property, our technique checks, for each rule that has two or more parts, whether the formula $\neg(p_1 \vee p_2 \vee \ldots \vee p_k)$ has a solution.

Property 3: Rule Compatibility. The rule compatibility property requires that for any set of applicable rules of an operation, the postconditions of their relevant rule parts must not be conflicting. To illustrate, consider rules $R_1 = \{r_1\}$ and $R_2 = \{r_2\}$ for an operation, such that $r_1 := ((a > 0) \implies (total = 10))$, $r_2 := ((b > 0) \implies (total = 20))$, and the two preconditions are not disjoint (*i.e.,* $(a > 0) \wedge (b > 0)$ is satisfiable). This pair of rules violates the compatibility property because the postconditions are conflicting, whereas the corresponding preconditions can be true simultaneously—the first precondition does not constrain the value of `b`, and the second precondition does not constrain the value of `a`. Thus, when both preconditions are true, it is not clear what value `total` would have after the operation.

To state this property formally, let $r_1 := (p_1 \implies q_1)$ and $r_2 := (p_2 \implies q_2)$ be the relevant rule parts of two applicable rules of an operation. Then, if $(p_1 \wedge p_2)$ is satisfiable, $(q_1 \wedge q_2)$ must be satisfiable. To verify this, our technique enumerates all pairs of

[3]In the currently implemented system, set types are handled in a restricted manner: by limiting the set size to two and specifying the predicates over a set in terms of its (two) elements.

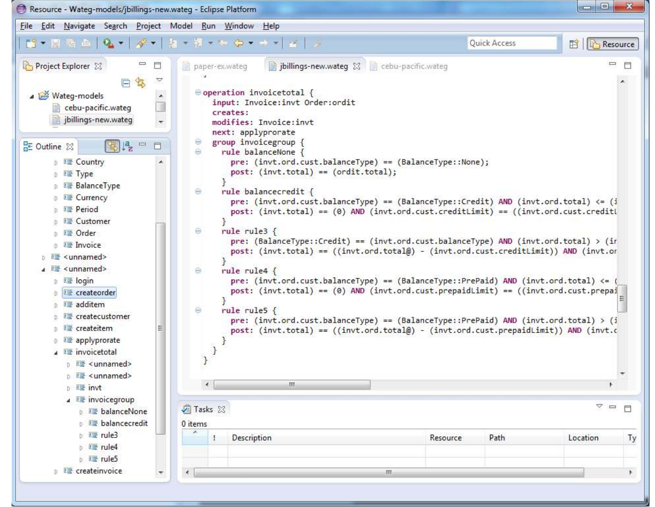

Figure 4: Screenshot of the Eclipse-based rule editor.

rules for an operation. Then, for each pair R_1 and R_2, the techniques lists each combination (p_1, p_2) of the preconditions of R_1 and R_2, and checks the satisfiability of $(p_1 \wedge p_2)$ and the conjunction of the corresponding postconditions.

If two rules violate the compatibility property, it might in fact indicate that their rule parts can be merged into one rule. In the preceding example, R_1 and R_2 could be merged into one rule with two rule parts $R_m = \{r_{m_1}, r_{m_2}\}$, where $r_{m_1} := (a > 0) \implies (total = 10)$ and $r_{m_2} := (a \leq 0 \wedge b > 0) \implies (total = 20)$. Note adding the conjunct $a \leq 0$ (the negation of the precondition of r_1) to the precondition of r_{m_2} makes the two preconditions disjoint, which ensures that R_m satisfies the rule-part disjointedness property. Alternatively, the negation of the precondition of r_2 could be added to the precondition of r_{m_1} to satisfy this property.

Property 4: Rule Independence. Finally, we enforce the restriction that there can be no data dependence between the postcondition of one rule and the precondition of another rule of the same operation. This property ensures that there is no implicit ordering among the rules of an operation. When such an ordering exists between two rules of an operation, the operation should be split into two operations in a sequence specified using the `next` clause.

We formalize this property as follows. Let $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ be the rules associated with an operation. For any $R_i, R_j \in \mathcal{R}$, let $V_{i,post}$ be the set of variables used in the postconditions of the rule parts of R_i and $V_{j,pre}$ be the set of variables used in the preconditions of the rule parts of R_j. Then, $V_{i,post} \cap V_{j,pre} = \emptyset$. The checking of this property is straight forward: for each pair of rules for an operation, the technique computes the sets of variables used in the preconditions of one rule and the postconditions of the other rule, and verifies that the two sets are non-intersecting.

2.3 Guided Model Creation

The modeling process must be realistic in that users should be able to create and refine models in a tool-assisted manner, with immediate checking of, and feedback on, errors in the models. We have implemented such a rule editor as an Eclipse plugin; Figure 4 shows a screenshot of the editor. The editor is built using Eclipse Xtext,[4] which is a general framework for developing domain-specific and programming languages. Much like any modern syntax-directed editor for a programming language, the rule

[4]`https://www.eclipse.org/Xtext/`

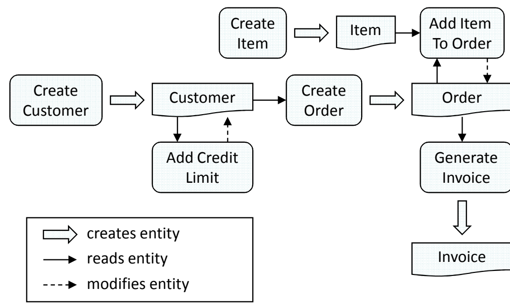

Figure 5: Sample operations and their interactions in `jBilling`.

editor provides syntax highlighting, navigation features, auto completion, and on-the-fly detection of syntax errors. The user can also run a semantic checker to detect violations of the well-formedness properties discussed previously. Thus, the editor provides a convenient environment for writing rules, in which the user is guided by automated checking and feedback.

3. EXAMPLE APPLICATION MODEL

Before presenting our technique, we elaborate upon the billing application example (`jBilling`) mentioned in the previous sections: we explain various operations and rules for the application, which we use subsequently to illustrate our technique. To facilitate the illustration, we use a simplified version inspired by the actual application.[5] (In the empirical evaluation, we modeled a larger part of the application.) The model for `jBilling` includes four entities: `Customer`, `Item`, `Order`, and `Invoice`. Figure 5 shows the flow among some of the operations, which create, read, or modify these entities. For example, operation `CreateCustomer` creates an instance of `Customer`, whereas the operation `AddItemToOrder` modifies an `Order` instance by adding a new item to the order.

Table 1 presents the formal specification of the `jBilling` operations and rules. Columns 2–4 list, respectively, the entities read, created, and modified by an operation. Columns 5 and 6 provide informal descriptions and formal representations of the rules associated with an operation. `GenerateInvoice` has two rules associated with it, whereas all the other operations have only one rule associated with them. The first rule for `GenerateInvoice` has three rule parts, which determine how the invoice total and the customer's credit limit are updated by the operation, based on the customer's balance type. The second rule for `GenerateInvoice` has two rule parts—which also determine the computation of invoice total, this time based on the customer's state of residence.

The boolean expressions in the rule parts refer to the input/created/modified entities by names: *e.g.,* the rule part for `CreateOrder` refers to the input `Customer` instance as `cust` and the created `Order` instance as `ord`. The expressions also refer to attributes of the entities. For example, some of the relevant attributes of `Customer` are:

```
Customer {
    State: state
    BalanceType: balType
    int: crLimit
}
```

where `State` and `BalanceType` are enumerated types (*e.g.,* attribute `BalanceType` can take two values—`None` or `Credit`) and the attribute `crLimit` is an integer.

If an expression needs to refer to the old and new values of an attribute (*i.e.,* the values prior and subsequent to an operation invocation), the old value is distinguished by appending '@' to the attribute name. For instance, consider the postcondition in the rule for `AddCreditLimit`: `cust.crLimit = cust.crLimit@ + amount`; this states that the input credit amount is added to the customer's old credit limit to obtain the new credit limit.

[5] `http://www.jbilling.com/`

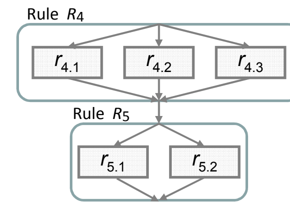

Figure 6: Operation flow graph for `GenerateInvoice`.

4. TEST GENERATION

Given an operation O and a target rule part r in O, our technique attempts to generate a *test sequence*, consisting of an operation sequence and test data, that *covers* r—that is, the sequence produces all input entities of O in appropriate object states to ensure that the precondition of r is satisfied. The challenge is to search for a covering sequence efficiently from a large set of candidate sequences. For example, one of our experimental subjects, `Cebu-pacific`, has eight operations that accept an entity `Ticket` as input and modify it. Therefore, any sequence composed using one or more of these operations can be a candidate sequence; however, different sequences and orderings produce different object states of `Ticket`. Our technique addresses this challenge by exploring the large search space efficiently, in a goal-directed manner, by ignoring sequences that are not likely to cover the target rule.

4.1 Terminology

Before presenting the test-generation algorithm, we introduce some terminology and definitions: we discuss two types of graphs (the dependence graph and the operation flow graph) and define abstract and concrete operation sequences.

4.1.1 Graphs

Dependence Graph. A *dependence graph* is a directed graph that captures interactions of operations with entities: nodes in the graph represent operations and entities, whereas edge represent creates, reads, and modifies relations. A *creates edge* or a *modifies edge* from an operation to an entity indicates that the operation creates or modifies that entity; a *reads edge* from an entity to an operation indicates that the operation reads the entity. Figure 5 shows the dependence graph for our example application.

Operation Flow Graph. To identify combinations of rule parts (within an operation) that create different object states of entities, we model an operation as a flow graph. The *operation flow graph* for an operation is a directed graph consisting of rule subgraphs—one per rule of the operation—and edges representing flow among the rule subgraphs. A *rule subgraph* consists of a unique entry point, a unique exit point, and nodes that represent rule parts; the subgraph contains an edge from the entry point to each node and an edge from each node to the exit point. Figure 6 illustrates the operation flow graph for `GenerateInvoice` with two rules, R_4 and R_5, which have three and two rule parts, respectively.

Because preconditions of all rule parts in a rule are disjoint (Property 1, Section 2), control flow through a rule covers exactly one rule part. The rule subgraph captures this aspect by representing each rule as a choice among its rule parts. Moreover, because there are no data dependences between rules of an operation (Property 4, Section 2), the rule subgraphs can be composed in any order in the operation flow graph (*i.e.,* all orderings are equivalent).

Table 1: Formal specification of sample business rules in `jBilling`.

Operation	Inputs (I)	Creates (C)	Modifies (M)	Rules (R)	
				Description	Formal Representation
Create Customer	{State, BalanceType}	{Customer}	$\emptyset$	R_1: The *credit limit (crLimit)* of newly created customers should be *zero*	$r_{1.1}$: $(\texttt{true}) \Longrightarrow (\texttt{cust.state} = \texttt{state} \wedge \texttt{cust.balType} = \texttt{balType} \wedge \texttt{cust.crLimit} = 0)$
Create Item	{Int}	{Item}	$\emptyset$	R_2: The price of an item must be greater than zero	$r_{2.1}$: $(\texttt{price} > 0) \Longrightarrow (\texttt{item.price} = \texttt{price})$
Create Order	{Customer}	{Order}	$\emptyset$	R_3: The *total* of a newly created order should be zero	$r_{3.1}$: $(\texttt{true}) \Longrightarrow (\texttt{ord.total} = 0 \wedge \texttt{ord.cust} = \texttt{cust})$
Generate Invoice	{Order}	{Invoice}	$\emptyset$	R_4: A customer's balance type determines how the invoice total is computed (see complete rule in the Introduction)	$r_{4.1}$: $(\texttt{ord.total} > 0 \wedge \texttt{ord.cust.balType} = \texttt{None}) \Longrightarrow (\texttt{inv.total} = \texttt{ord.total})$ $r_{4.2}$: $(\texttt{ord.total} > 0 \wedge \texttt{ord.cust.balType} = \texttt{Credit} \wedge \texttt{ord.cust.crLimit} \geq \texttt{ord.total}) \Longrightarrow (\texttt{inv.total} = 0 \wedge \texttt{ord.cust.crLimit} = \texttt{ord.cust.crLimit@} - \texttt{ord.total})$ $r_{4.3}$: $(\texttt{ord.total} > 0 \wedge \texttt{ord.cust.balType} = \texttt{Credit} \wedge \texttt{ord.cust.creditLimit} < \texttt{ord.total}) \Longrightarrow (\texttt{inv.total} = \texttt{ord.total} - \texttt{ord.cust.crLimit} \wedge \texttt{ord.cust.crLimit} = 0)$
				R_5: If the customer's residence is in NY *state*, an additional 2% discount is given while generating invoices	$r_{5.1}$: $(\texttt{ord.total} > 0 \wedge \texttt{ord.cust.state} = \texttt{NY}) \Longrightarrow (\texttt{inv.total} = \texttt{ord.total} * (98/100))$ $r_{5.2}$: $(\texttt{ord.total} > 0 \wedge \texttt{ord.cust.state} = \texttt{Other}) \Longrightarrow (\texttt{inv.total} = \texttt{ord.total})$
Add Credit Limit	{Customer, Int}	$\emptyset$	{Customer}	R_6: The credit limit can be incremented for customers with balance type *Credit*	$r_{6.1}$: $(\texttt{cust.balType} = \texttt{Credit} \wedge \texttt{amount} > 0) \Longrightarrow (\texttt{cust.crLimit} = \texttt{cust.crLimit@} + \texttt{amount})$
Add Item to Order	{Order, Item}	$\emptyset$	{Order}	R_7: Adding an item to an order increases the order's total by the item's price	$r_{7.1}$: $(\texttt{true}) \Longrightarrow (\texttt{ord.total} = \texttt{ord.total@} + \texttt{item.price})$

4.1.2 Operation Sequences

Abstract Sequence. An *abstract sequence* is a sequence of operations describing flow of objects among the operations such that all variables that represent instances of entities are defined and other variables (of enumerated and primitive types) need not be defined. An example abstract sequence for `GenerateInvoice` is:

```
State st; BalanceType bt; int price;
Customer cust = CreateCustomer(st, bt);
Order ord = CreateOrder(cust);
Item item = CreateItem(price);
Order ord1 = AddItemToOrder(ord, item);
Invoice inv = GenerateInvoice(ord1);
```

All instances of entities are initialized within the sequence. In case an operation creates or modifies multiple entities, we represent those entities as arguments prefixed with the keyword `out`.

Concrete Sequence. A *concrete sequence* constrains an abstract sequence with selected rule parts for each operation. For an operation, there can be multiple rule parts selected from different rules of the operation. Our technique uses concrete sequences to generate test data by leveraging a constraint solver: it builds a logical formula from concrete sequences using preconditions and postconditions of all rule parts, and uses constraint solver to check whether the formula is satisfiable. An example concrete sequence for the preceding abstract sequence is:

```
State st; BalanceType bt; int price;
Customer cust = CreateCustomer(st, bt) [r1.1];
Order ord = CreateOrder(cust) [r3.1];
Item item = CreateItem(price) [r2.1];
Order ord1 = AddItemToOrder(ord, item) [r7.1];
Invoice inv = GenerateInvoice(ord1) [r4.1];
```

4.2 The Algorithm

The test-generation algorithm (shown as Algorithm 1) takes as inputs an operation O and a rule part r, and generates a concrete sequence seq that cover r. To illustrate the algorithm, we consider the operation `GenerateInvoice` and rule part $r_{4.2}$. For brevity, we omit details such as exiting the main loop (lines 6–13) when the user-defined threshold of maximum number of sequences to be explored is reached. We first explain the core algorithm and then present our optimization based on using unsatisfied cores for efficiently exploring the search space.

Algorithm 1: The algorithm for generating a concrete sequence that covers a given rule part.

Input: Operation O, Rule part r
Output: Concrete sequence *seq* or *null*

1 Let q represents a queue of concrete sequences;
2 Generate all initialization sequences *iseqs* for O;
3 **foreach** *sequence iseq* $\in$ *iseqs* **do**
4   Generate all concrete sequences *cseqs* for *iseq*;
5   Add *cseqs* to queue q;
6 **while** *q not empty* **do**
7   Dequeue sequence *cseq* from q;
8   Check whether *cseq* is satisfiable;
9   **if** *satisfiable* **then**
        return *cseq*;
10   Identify candidate operations *ops*;
11   **foreach** *operation op* $\in$ *ops* **do**
12   Generate new sequences *nseqs* by adding *op* to *cseq*;
13   Add all *nseqs* to queue q;
return *null*;

Generate Initialization Sequences (Lines 2–5). The algorithm first generates, using the dependence graph, all possible initialization sequences that produce input entities of O. An initialization sequence is an abstract sequence, with the restriction that it includes only those operations that create entities. To create the initialization sequences, the algorithm identifies input entities $I = \{i_1, i_2, \ldots, i_m\}$ of O through *reads* edges in the dependence graph. For each i_k, it identifies the operations $OP_k = \{O_1^k, O_2^k, \ldots, O_n^k\}$ that create i_k through *creates* edges in the graph. Then, it computes combinations of all operations across each set corresponding to i_k to generate initialization sequences. Therefore, for m input entities and n operations that create each entity, the algorithm generates n^m initialization sequences. Note that the order of operations among the sequences does not matter because the operations create different entities. In theory, the number of initialization sequences could be high; however, in our empirical evaluation, we found that the number of operations that create entities is often quite low, resulting in a few initialization sequences only.

The algorithm checks whether any operation in OP_i further requires additional entities, and repeats this process until no new

input entities are required in all initialization sequences. For our illustrative example, the initialization sequence for the operation `GenerateInvoice` is:

```
Sequence S1:
State st; BalanceType bt;
Customer cust = CreateCustomer(st, bt);
Order ord = CreateOrder(cust);
Invoice inv = GenerateInvoice(ord);
```

Next, for each initialization sequence, the algorithm generates concrete sequences by computing all possible combinations of rule parts among operations in the sequence. To do this, it joins the operation flow graphs for the operations and enumerates all paths in the composed flow graph. The concrete sequences generate different object states for input entities of O. For our running example, there is only one concrete sequence because each operation in the initialization sequence has only one rule part:

```
Sequence S2:
State st; BalanceType bt;
Customer cust = CreateCustomer(st, bt) [r1.1];
Order ord = CreateOrder(cust) [r3.1];
Invoice inv = GenerateInvoice(ord) [r4.2];
```

Check Satisfiability (Line 8). Next, the algorithm checks whether the concrete sequences in the queue are satisfiable. A concrete sequence is satisfiable if it generates desired object states for all input entities of O that cover the precondition of the target rule part. To achieve this, the algorithm constructs a logical formula composed of constraints in preconditions and postconditions in each rule part of the sequence and leverages a constraint solver to check whether the composed formula is satisfiable.

For illustration, consider the sequence of operations with selected rule parts as $(O_1[r_{1.1}], O_2[r_{2.1}], \ldots, O_n[r_{n.1}], O[r])$. The algorithm starts with the precondition of r (referred to as *target*) in operation O. It generates binding constraints that substitute the entities consumed by O with the entities created or modified by the predecessor operation O_n. The binding constraints bind the identifiers in the postcondition of $r_{n.1}$ to the identifiers of the same type in the precondition of r. Because the solver has no notion of objects, the binding constraints ensure that referenced object attributes are appropriately bound as well.

The binding constraints are generated as follows. Let v be an identifier of type τ occurring in the creates or modifies clause of a predecessor (*e.g.,* O_n) and let w be an identifier of the same type occurring in the input clause of the successor (*e.g.,* O). If τ is an enumerated or primitive type, the only binding constraint needed is $w = v$. However, if τ is an object type, we must bind all attributes as well, yielding the following constraint:

b_n: $w = v \wedge w.f_1 = v.f_1 \wedge \ldots \wedge w.f_n = v.f_n$

Here, $f_1, \ldots, f_n$ are attributes of τ. To generate the final binding constraint, this process is applied recursively on each attribute of object type. Using binding constraints, the algorithm generates the formula as $p_{n.1} \wedge q_{n.1} \wedge b_n \wedge p$, where $p_{n.1}$ and $q_{n.1}$ are the precondition and postcondition of $r_{n.1}$. The algorithm checks whether the composed formula is satisfiable. If it is, this formula becomes the next *target* and O_{n-1} the predecessor operation, and the algorithm repeats the same process with other operations in the sequence. Once it has processed all operations in the sequence and the formula is satisfiable, it extracts values for variables of primitive and enumerated types from the constraint solution to generate test data.

For our running example, the solver finds the composed formula unsatisfiable. The reason is that rule part $r_{3.1}$ of `CreateOrder` assigns zero to attribute `total` of `Order`, whereas the precondition of $r_{4.2}$ of `GenerateInvoice` requires `total` to be greater than zero.

Identify Candidate Operations (Line 10). Our algorithm next identifies candidate operations *ops* (along with relevant rule parts in those operations) that modify any of the entities that were produced in the sequence. The reason is that sometimes object states corresponding to intermediate objects need to be modified to produce desired object states for input entities of O to cover the rule part r. For instance, in our running example, the customer whose balance type is `Credit` should have sufficient credit limit to cover rule target $r_{4.2}$.

Generate Alternate Sequences (Lines 11–13). After a candidate operation *op* (along with relevant rule parts) is identified, the algorithm checks whether any additional input entities that are not yet available in the current sequence *cseq* are required by the operation *op*. If so, it identifies the additional operations that create those entities. Then, using the dependence graph, it identifies all positions in *cseq* where the candidate operation (along with the additional operations) can be inserted. Note that there can be multiple positions that satisfy dependencies for inserting the candidate operation, and the resulting sequences can produce different object states for the input entities of O. Therefore, the algorithm can generate multiple sequences while inserting candidate operation into *cseq*. Finally, the algorithm adds all newly generated sequences to the queue to further analyze those sequences. Our algorithm also prunes duplicate sequences that were already explored in the previous iterations.

For our running example, it identifies candidate operations as `AddItemToOrder` and `AddCreditLimit`. Since `AddItemToOrder` requires an additional entity `Item` that is not available in the sequence, the algorithm adds operation `CreateItem` as well to the newly generated sequence. In the next iteration, for the sequence including `AddCreditLimit`, our algorithm identifies `AddItemToOrder` as a candidate operation, and finally generates the concrete sequence that covers $r_{4.2}$ as follows:

```
Sequence S3:
State st; BalanceType bt; int price;
Customer cust = CreateCustomer(st, bt) [r1.1];
Customer cust1 = AddCreditLimit(cust, crLimit) [r6.1];
Order ord = CreateOrder(cust1) [r3.1];
Item item = CreateItem(price) [r2.1];
Order ord1 = AddItemToOrder(ord, item) [r7.1];
Invoice inv = GenerateInvoice(ord1) [r4.2];
```

4.3 Optimizations

The preceding algorithm can handle only small models where there exist only a few operations that modify entities. Instead, if there exist many operations that modify each entity, the search space can easily become exponential. To address this issue, we use the following optimizations based on unsatisfied core that helps prune the search space. These optimizations replace Line 10 of Algorithm 1 for identifying candidate operations.

Extract Unsatisfiable Core. In particular, if the composed formula in Line 8 is unsatisfiable, we extract the unsatisfiable core, `ucore`, of the formula. The unsatisfiable core is a subset of the formula that preserves the unsatisfiability but is simplified compared to the original formula. `ucore` guides our algorithm towards operations that modify attributes of entities that help produce the desired object states. (Section 5.1 presents the implementation details of how we extract `ucore` from a formula.) In our example, for sequence S_2, we extract `ucore` as `ord.total = 0 ∧ ord.total > 0`, composed of constraints from rule parts $r_{3.1}$ and $r_{4.2}$.

Before searching for other candidate operations, we discard constraints from `ucore` that are contributed by the last analyzed operation (as these constraints cause the unsatisfiability of *target*). We

use the notation ucore- to represent the remaining constraints in the extracted unsatisfied core. The intuition behind computing ucore- is to identify candidate operations that are compatible with ucore- so that the composed formula can be satisfiable. We first extract entities and their attributes involved in ucore-. Next, we identify candidate operations that modify those entities. For each such candidate operation, we identify rule parts that modify the desired attributes and also whose postconditions are compatible with ucore-, *i.e.,* $q \wedge b \wedge$ ucore- is satisfiable, where q is the postcondition of the selected rule part in the candidate operation and b represents binding constraints. This additional satisfiability check helps discard candidate operations that do modify the desired attributes but still cannot help in identifying a covering sequence for r; this can significantly reduce the search space of candidate sequences.

For our illustrative example, we compute ucore- as `ord.total > 0`, and identify the entity as `Order` and the desired attribute to be modified as `total`. We then analyze all operations that create or modify `Order`, and identify the candidate operation as `AddItemToOrder` and the rule part $r_{7.1}$, since $q_{7.1} \wedge$ ucore- is satisfiable. Even if there exist many operations that modify the `Order` entity, our optimization helps discard many of those operations, thereby increases the efficiency by aggressively pruning the search space. After creating the new concrete sequence, we check satisfiability of the new sequence, where we further identify ucore as `cust.crLimit = 0` $\wedge$ `cust.crLimit > 0` and find `AddCreditLimit` as a candidate operation.

Check Progress. While identifying candidate operations, our optimization discards those operations that do not help cover the target rule part. However, in a few cases, even the identified operations may not help make progress and need to be discarded to make the exploration efficient. To identify such cases, we check the following two aspects for each sequence *cseq*.

First, if the extracted unsatisfiable core ucore was already seen in previous iterations related to *cseq*, we discard *cseq*. The reason is that the candidate operation that was added to *cseq* in the previous iteration did not help satisfy the previous ucore, resulting in the same ucore in the next iteration as well.

Second, when ucore includes integer variables, we use a fitness function to measure whether *cseq* helps get closer to cover the rule part r or not. The first check is sufficient for boolean variables or variables of enumerated types but is insufficient to deal with the integer variables.

To illustrate the issue, consider that our model includes another operation, `RemoveItemFromOrder`, that removes a selected item and decreases the value of attribute `total` of `Order`. While exploring candidate operations, our optimization identifies `AddItemToOrder` and `RemoveItemFromOrder` because both operations modify `total`. However, `RemoveItemFromOrder` does not help cover $r_{4.2}$ as it actually decreases the value of `total`. To handle such cases, our optimization uses a fitness function, originally proposed in Fitnex [25], to measure whether *cseq* gets closer to covering the target rule part. The idea is to compute a fitness value from ucore and if this value is better than the fitness value computed during the previous iteration, *cseq* is processed further; otherwise, it is discarded.

5. EMPIRICAL EVALUATION

We implemented our technique in a prototype tool called BUSTER (BUSiness TEsting Rules), and conducted two empirical studies using two open-source applications and one proprietary enterprise application. In the first study, we compared the effectiveness of BUSTER in covering business rules with a related technique that systematically explores the search space without any guidance. In the second study, we investigated the efficiency of both the techniques. After describing the experimental setup, we present results of the two studies.

Table 2: Subjects used in our empirical studies.

Subject	URL	Entities	Operations	Rules	Rule parts
Cebu-pacific	www.cebupacificair.com	8	10	15	31
jBilling	www.jbilling.com	10	10	14	26
App	—	12	13	13	20
Total			33	42	77

5.1 Experimental Setup

Implementation. Our implementation BUSTER uses CHOCO constraint solver [1] for checking the satisfiability of sequences. Since CHOCO does not provide functionality for extracting unsatisfied core, we use a heuristic-based implementation to extract the core. While checking whether a concrete sequence is satisfiable, BUSTER starts with the last operation of the sequence and performs backward analysis by handling each operation and their rule parts. Therefore, when a composed formula is unsatisfiable, it can be caused only by a constraint in the most recently analyzed rule part. Next, it uses the variables involved in that constraint to extract the complete unsatisfied core. Note that BUSTER may not extract the minimal unsatisfied core; however, the extracted core is sufficient to suggest candidate operations. We leave improvements in the implementation to future work, for example, by exploring advanced algorithms for extracting unsatisfied core [7] and also leverage other constraint solvers [3]. Another optimization could include caching and reusing constraints (*e.g.,* as discussed in the Green approach [22]) to improve further the efficiency of the search.

To compare the efficiency of BUSTER with an unguided search over the space of candidate sequences, we implemented another tool, called EXHAUST. EXHAUST is primarily the same as BUSTER without optimizations presented in Section 4.3. Being an unguided technique, EXHAUST lets us evaluate the effectiveness of leveraging unsatisfied core to guide the search for covering sequences.

Subjects and Rules. We used two open-source applications and a proprietary enterprise application, listed in Table 2, as the experimental subjects. Due to confidentiality reasons, we refer to the proprietary application as `App`. `Cebu-pacific` is an airlines application, `jBilling` is an enterprise billing application, and `App` is a telecom application. Column 2 shows the URLs of the first two subjects. Columns 3–6 show additional details, such as the number of entities, operations, and rules. For each subject, we identified a module that is likely to have large number of interesting rules with respect to the sequences and test data required for coverage, and modeled those modules using our tool. In particular, we used the *ticket cancellation* and *generate invoice* modules for `Cebu-pacific` and `jBilling`, respectively. In total, we modeled 33 operations with 42 rules and 77 rule parts.[6] For each subject, we spent approximately 10 hours to model the operations and rules. We also found that our rule editor and property checking helped create correct rule set.

Method. We applied BUSTER and EXHAUST on the models created for each subject and generated test sequences. For each rule part, we let each tool explore up to a maximum of 100 concrete sequences. Next, we inspected the generated sequences to ensure that they cover the targeted rule parts. To evaluate the efficiency of

[6]The complete dataset including original English descriptions, models and the generated test sequences is available at `http://tinyurl.com/k4b4j8x` (also available on request from the authors).

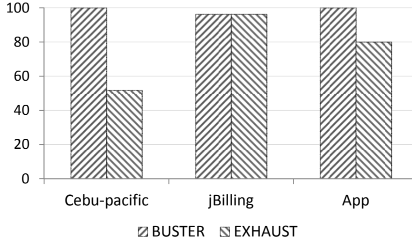

Figure 7: Effectiveness of the two techniques in covering business rules.

the tools, we measured the number of sequences explored and the lengths of sequences produced by each tool. All experiments were conducted on an Intel Core 2 Duo CPU machine with 2.53 GHz and 8GB RAM. Next, we present the results of the studies.

5.2 Coverage of Business Rules

Figure 7 presents the results for all three subjects: it shows the percentages of rule parts covered by each tool. For example, for `Cebu-pacific`, BUSTER generated covering sequences for all 31 rule parts, whereas EXHAUST was able to generate sequences only for 16 rule parts. Overall, the results show that BUSTER covered 99% of the rule parts, whereas EXHAUST could cover only 74% of the rule parts.

We further analyzed the cases where the tools could not cover the targeted rule parts. We found that, in general, EXHAUST performs poorly if there are many operations that create or modify the required input entities. This is expected because it results in a large search space of candidate operation sequences, which BUSTER is able to navigate effectively in a goal-directed manner (guided by the unsatisfied core), whereas EXHAUST has to try the candidate sequences in a blind manner. Thus, EXHAUST performed well on `jBilling`, in which each entity is created or modified by only a few operations. But, for `Cebu-pacific`, where some entities are modified by many operations, EXHAUST was much less effective. In such cases, a directed search guided by unsatisfied core, is necessary—and can be highly effective—for attaining high rule coverage.

For `jBilling`, both BUSTER and EXHAUST could not cover one rule part because of an issue with CHOCO solver: CHOCO terminated with an out-of-memory error while solving the composed formula for that rule part.

Next, we illustrate a complex sequence generated by BUSTER for a rule part in `Cebu-pacific`. This rule part pertains to fare refunds because of flight cancellation due to a delay of more than two hours. `Cebu-pacific` allows a ticket to be booked as multiple sectors, where each sector represents part of the journey from one city to another city. The airlines has a refund policy that if the flight corresponding to any sector gets canceled due to a delay of more than two hours, passengers can get a refund so as to make alternative travel arrangements. To cover this rule part that belongs to operation `Refund`, a specific instance of `Ticket` entity is required. First, the ticket should include at least one sector and the passenger should have sufficient funds to add a sector to the ticket. Next, the flight corresponding to that sector should be delayed by more than two hours and, consequently, canceled. BUSTER successfully generated the following covering sequence and test data, whereas EXHAUST failed to generate a covering sequence.

```
int fund = 200, passenger = 1, delay = 3, sectorid = 1;
Fund fund = CreateFund(fund, passenger);
Ticket ticket = CreateTicket(passenger);
int flight = 901, price = 100, departure = 10;
Sector sector = MakeSector(flight, price, departure);
AddSector(ticket, sector, fund, out Ticket ticket1, out Fund f1);
Ticket ticket2 = DelayFlight(ticket1, sectorid, delay);
Ticket ticket3 = PartialCancellation(ticket2, sectorid);
Refund(ticket3, sectorid, f1, out Ticket ticket4, out Fund f2);
```

Overall, our results illustrate the promise of our technique in effectively generating complex sequences for covering business rules.

Table 3: Efficiency of BUSTER and EXHAUST.

Subject	Sequence Length				# of Sequences Explored			
	BUSTER		EXHAUST		BUSTER		EXHAUST	
	Max	Avg	Max	Avg	Max	Avg	Max	Avg
Cebu-pacific	7	5	9	5	27	4	100	73
jBilling	6	3	6	3	48	2	39	2
App	9	6	10	6	51	5	100	46

5.3 Efficiency

Table 3 presents data about the lengths of sequences and the number of sequences generated by BUSTER and EXHAUST. Columns 2–5 show the maximum and average lengths of sequences generated for all rule parts, whereas Columns 6–9 show the number of sequences explored by each tool for all rule parts.

The results show that, in some cases, BUSTER generated relatively shorter sequences compared to EXHAUST. More significantly, BUSTER explored much fewer sequences than EXHAUST. For example, for `Cebu-pacific`, BUSTER explored on average only 4 sequences (maximum of 27), whereas EXHAUST explored on average 73 sequences for each rule part (and also hit the upper bound of 100 sequences). In none of the cases, BUSTER terminated by reaching the upper bound. Overall, these results indicate that guided search via unsatisfied core can make BUSTER highly efficient compared to EXHAUST.

5.4 Discussion

These results illustrate the promise of our technique. But, further experimentation with more varied subjects and business rules are required to confirm the generality of, and increase our confidence in, these observations.

This work partially fulfills our vision of making the testing of enterprise applications more tool-based. Our longer-term goal is to generate executable test cases that drive the application via its GUI—as illustrated by the flow depicted in Figure 1—to test the application's conformance to business rules. Toward that goal, we plan to leverage our previous work [17], in which we developed a tool, WATEG, that performs directed crawling of a web application to generate executable GUI test cases. WATEG also takes as input a specification of rules, but those rules are expressed in terms of the GUI elements (*e.g.,* links, buttons, and text boxes) of an application, and pertain to access-control properties, navigational properties, and so on. A natural integration between this work and WATEG-style crawling technology would be to extend our rule-modeling language to accommodate *flow specifications* for operations, which (along with the test data) could be provided as input to WATEG to generate executable rule-covering tests.

6. RELATED WORK

At first blush, this problem seems to be reminiscent of the problem of generating tests for programs written as control-flow graphs, with the goal of exercising each acyclic path in the program, if possible. There have been a number of techniques in the literature for test generation. Mostly notably, techniques such as *concolic* testing attempt to identify a series of test data that would force program execution thorough different paths [5, 16]. Other approaches are based on model checking, with the goal of creating test inputs to reach specific program states. However, the problem of test generation from business rules is different. Business rules do not describe the implementation of a system: rather they only describe a model

and many of the concerns that arise when dealing with control flow graph derived from real code are not pertinent.

Model Based Testing. The set of business rules can viewed as a model of an actual system that supports them and our approach then becomes a variant of model based test generation [21].

Typical modeling languages used by model based testing systems include UML sequence diagrams [10], modeling specific languages such as Systems Modeling Language [4] and finite state machine notations such as UML state charts [11]. While the business rules as presented in this paper could be expressed in any of these notations, it would be cumbersome and require some degree of non-trivial encoding on part of users, making it unattractive to a non-programmer such as a business analyst. Our business rule language is designed to be user friendly, where syntax is close to prose, thereby making it more accessible to non-programmers.

The work on combinatorial test optimization [15] also requires a model of an application as a starting point, but that model is typically not rich enough to allow generation of test scenarios; the model does not have an operation-based view of the application. However, test optimization can be carried out in a post pass over the tests generated by our technique.

AI Planning. Previous work has applied AI planning to software testing [14, 6]. The sequencing of operations done by our algorithm is similar to the AI planning problem [23] where actions with pre- and postconditions are sequenced by a planner algorithm using either forward or backward chaining. The algorithms used by AI planners are similar to EXHAUST (presented in Section 5) and proved to be insufficient for our benchmarks. Part of an AI planning problem is the initial state of the program, including what objects exist and what conditions can be assumed. Our technique does not need such information—the types of objects that exist in the system are defined in the model along with operations that create them.

Paradkar et al. [13] present a system for testing web services specified in the semantic markup language OWL-S [8]. The system is backed by an AI planner. OWL-S represents operations in a manner similar to our language with pre- and postcondtions. However each operation has only one associated pre- and postcondition pair making it more akin to a rule in our language. If one were to translate a model in our business rule language into OWL-S, each rule would be translated into a standalone operation. Such a translation could potentially increase the size of the search space and lead to spurious sequences being constructed. The focus of the technique presented in the paper is on conformance testing, that is, to test whether a given implementation conforms to the specification. In our technique, business rules are the specification and the aim is instead to generate sequences that cover all rules. Their paper does not mention the size of the models the technique was evaluated on nor the size of the generated test sequences.

Testing Object-Oriented Programs. Our work is also related to existing techniques that focus on test generation for object-oriented programs [12, 19, 18, 24, 2, 9, 20]. At a high level, these techniques also focus on inferring operation sequences with a goal of generating different object states for receiver or arguments of method under test so as to achieve high code coverage. Among these techniques, Symstra [24] uses bounded-exhaustive technique, Randoop [12] and JCrasher [2] select methods randomly to compose sequences, and Tonella [20] and McMinn et al. [9] use evolutionary approaches. Among all these techniques, our work is closely related to Seeker [18] that also attempts to compose sequences incrementally based on branches that are not yet covered. The major contribution of Seeker is to infer candidate operations especially when those branches include private fields, since private fields cannot be modified directly. In our current work, we do not face that issue, since our model does not have the notion of public and private attributes. Our current work extends Seeker by extracting unsatisfied core and using to guide the search process. We believe that Seeker can greatly benefit with our new algorithm of leveraging unsatisfied core as a guidance for suggesting candidate operations.

7. CONCLUSION

In this paper, we presented a new domain-specific language for modeling business rules that can capture functional specifications of enterprise systems. We also defined well-formedness properties on the model that can be verified mechanically. We implemented our tool as an Eclipse plug-in, where non-programmers also can create and refine models in a guided fashion We also presented a novel technique, based on unsatisfied cores, that generates test sequences by translating rules in to logical expressions. Our technique was evaluated using three models that were derived from business rules written in English. The results show that our technique is able to cover 99% of all business rules.

In future work, experimentation with more subjects would help confirm the generality of our results. Toward our longer-term vision of bringing end-to-end automation to the testing of enterprise applications, we will investigate the integration of BUSTER and WATEG (our previous work [17]) to generate executable test cases for validating an application's conformance to business rules.

8. REFERENCES

[1] Choco. http://www.emn.fr/z-info/choco-solver/.

[2] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[4] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Elsevier, 2011.

[5] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[6] A. E. Howe, A. V. Mayrhauser, R. T. Mraz, and D. Setliff. Test case generation as an ai planning problem. *Automated Software Engineering*, 4:77–106, 1997.

[7] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, 2008.

[8] D. Martin, M. Burstein, E. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-s: Semantic markup for web services, 2004.

[9] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Proc. GECCO*, pages 1013–1020, 2005.

[10] A. Nayak and D. Samanta. Model-based test cases synthesis using uml interaction diagrams. *SIGSOFT Software Engineering Notes*, 34(2), Feb. 2009.

[11] J. Offutt and A. Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, 1999.

[12] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[13] A. M. Paradkar., A. Sinha, C. Williams, R. D. Johnson, S. Outterson, C. Shriver, and C. Liang. Automated functional conformance test generation for semantic web services. In *ICWS*, pages 110–117. IEEE Computer Society, 2007.

[14] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an oo model with an ai planning system. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 0:250, 1999.

[15] I. Segall and R. Tzoref-Brill. Interactive refinement of combinatorial test plans. In *ICSE*, 2012.

[16] K. Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, 2007.

[17] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 162–171, 2013.

[18] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 189–206, 2011.

[19] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Tests And Proofs*, pages 134–153, 2008.

[20] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[21] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[22] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, page 58, 2012.

[23] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 1994.

[24] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, pages 365–381, 2005.

[25] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.