ABSTRACT
Testing object-oriented software is even for human beings a challenging task. Automating it is therefore very helpful for developers but by no means trivial. In this paper we present Alana, an approach for automatically generating complex objects used as test input data that satisfy a given precondition in terms of Design by Contract specification. Alana transforms the existing Design by Contract specification of the parameter type and the precondition of the method under test to PDDL (plan domain description language). Based on it, existing AI planners can be used to create a plan, i.e., a method sequence that transforms the object to the goal state. The goal state is given by the precondition of the method under test. Alana is evaluated on two case studies: a student-developed stack-based calculator, and a real-world event-based application developed by our industry partner. Alana outperforms a random approach significantly in terms of methods tested and line coverage on both case studies.

1. INTRODUCTION
Automating test generation consists of two subproblems: test sequence generation, and test data generation. The former deals with generating meaningful method call sequences for the object under test, whereas the latter focuses on generating test input data for those method calls. This work focuses on test data generation, more precisely on instantiating objects that satisfy the precondition of the method under test.

Consider, for example a method under test that requires a parameter of type Stack containing at least two elements. This very simple specification, turns out to be already very hard to solve in case of random-based approaches. The Java Stack implementation, for example, consists of 60 methods. Six methods add elements to the Stack, whereas ten methods remove elements from the Stack. Therefore, the probability to increase the size of the stack is lower than decreasing it. Calling two times a method that adds an element to the stack object on a random basis works out in only 1% (6/60 + 6/60 = 0.1 * 0.1 = 0.01 = 1%). But real-world methods do impose far stronger constraints on their input parameters.

To overcome this we introduce Alana: An automatic test data generation approach based on AI planning. The basic idea is similar to Howe [18] and Leitner [11]. The existing specification of the software system is used as planning domain description. An AI planner can use it to calculate an instantiation sequence for all objects involved, such that they satisfy the method under test precondition. Figure 1 gives an overview of the involved steps. But in contrast to previous work, Alana fully automates the approach and can deal with real Java implementations.

The paper continues as follows: Section 2 shortly introduces concepts and notations that are required to understand the Alana approach presented in Section 3. Alana is evaluated on two case studies, from which one is a software component from an international telecommunication company. The results are presented in Section 4. Finally, related work is presented in Section 5 and the paper concludes in Section 6.

2. PRELIMINARIES
First, we introduce a running example, which is used throughout the paper to explain the Alana approach. We removed all parts from the original specification that are not necessary for explaining Alana. Second, we introduce Design by Contract and AI planning, shortly. Third, we present a common notation in terms of formal definitions that we use to explain Alana in Section 3. The listed specification is explained in the following subsection.

2.1 Running Example
Figure 2 shows the running example we use throughout the paper. The task we try to solve is to generate a unit test...
24 @Post("stack.size()==1")
23 @Pre("stack.size()>=2")
22 class SummationOperator {
21 @Post("@Return>=0")
20 @Pure
19 public double peek () { ... }
18 @Post("size()==@Old(size())")
17 @Pre("size()>0")
16 public void pop () { ... }
15 @Post("@Return>=0")
14 @Pure
13 public void push (double element) { ... }
12 @Post("size()==@Old(size())+1")
11 private int size_ ;
10 public class Stack {
 9 @Invariant("size_>=0")
 8 public class Stack {
 7 @Post("size()>=0")
 6 public void push (double element) { ... }
 5 public void pop() { ... }
 4 private int size_ ;
 3 public class Stack {
 2 @Post("size()==0")
 1 public class Stack {

Figure 2: The running example. The method under test, i.e., sumUp(), requires an instance of a Stack object.

2.2 Design by Contract

Design by Contract™ allows to specify the intended semantic behavior of a class or method declaration. It became popular since Bertrand Meyer created Eiffel [12], a programming language that has built-in support for Design by Contract™ [13]. For Java 3rd party libraries such as JML [10] and Modern Jass [14] exist that add Design by Contract™ functionality.

The three main concepts of Design by Contract™ are [13]:

1. precondition: The precondition P is evaluated before the method is called. It specifies the expected value range of the method’s parameters. The method is only executed when the caller satisfies the precondition.

2. postcondition: The postcondition Q is evaluated directly after the method body is executed and before returning to the caller. It specifies the guaranteed behavior of the method. Whenever the client satisfies the precondition the method has to satisfy the postcondition, i.e. $P \implies Q$. In addition, the @Old keyword can be used in a postcondition to reference to a value before method execution. This is also called pre-state access.

3. class invariant: A class invariant specifies a property that applies for all instances of the class throughout their lifetime. All class invariants are checked before and after a method execution. The invariant may be violated while executing a method’s body.

Example 1: Consider the pop() method from our running example. The precondition "size()>0" specifies that pop() may only be called when at least one element is on the Stack. The postcondition "size()==@Old(size())-1" declares the behavior of pop(), namely that the size of the Stack object will decrease by one after successful execution of pop(). The "@Old(size())" specification references to the value of size() before pop() is executed. An invariant for the Stack object is stated in Line 1, which specifies that a Stack may not have negative size.

Furthermore, we have to note that in Design by Contract™ methods can be annotated as pure, which basically states that those methods do not change the objects’ state. Pure methods may be called within pre- and postcondition specifications.

2.3 AI planning

Planning can be described as finding a sequence of actions that will bring a system from its initial state IS to a desired goal state GS. Such a system must provide descriptions of what an action may change in the world and when the execution of the action is valid. The description of which conditions must hold before execution of an action a is valid, can be referred as precondition P of the action, whereas the list of changes is called the effect E. In terms of Hoare logic [9] such an action can be written as:

\[
\{P\} \text{A}\{E\}
\]

Therefore, a plan can be described as sequence of actions $a_1 \ldots a_n$ that transform the initial state IS such that in the end the goal state GS is satisfied. The plan is valid if the following sequence can be proven:

\[
\{IS\} \text{plan}\{GS\} \iff \{IS\land P_1\} a_1 \{E_1\} \land \{P_2\} a_2 \{E_2\} \land \ldots \land \{P_i\} a_i \{E_i\land GS\}
\]

Definition 1 (planning problem): A planning problem is the triple $PP = (D, IS, GS)$ where $D$ is the planning domain consisting of all available actions: $D = \{a_1, \ldots, a_n\}$, $IS$ is the initial state and $GS$ is the desired goal state. Initial state and goal state are defined in Definitions 11 and 13, respectively.

Example 2 (planning problem): For the running example the planning problem is given by $PP = (D, IS, GS)$ with $D = \{push, pop, stack.size()\}$, $IS = \{size == 0\}$ and $GS = \{size >= 2\}$.

2.4 Common Notation

In this section we try to define the notation of a class, the Design by Contract™ specification of a class, the external state and the state manipulators of a class formally. This is necessary to be able to explain the Alana approach in detail in Section 3.
Definition 2 (class): Galler et al. [5] define a class as a triple $C = (c, m, f)$. Where $c$ is the set of public available Constructors, $f$ is the union of public, protected and private fields: $f = f_{\text{pub}} \cup f_{\text{prot}} \cup f_{\text{priv}}$, and $m$ the set of all methods.

$$m = \bigcup_{\text{vis} \in \{\text{pub}, \text{prot}, \text{priv}\}} \bigcup_{\text{ret} \in \{\text{void}, \text{abs} \}$ \}$ \{\text{aba}, v_{\text{ret}}\}$$

Here \(\text{vis}\) denotes the visibility of the methods, which is either public, protected or private, \(\text{ret}\) states whether the method returns a value (\(\text{nvoid}\)) or not (\(\text{void}\)) and \(\text{abs}\) denotes whether the method is concrete (\(\text{con}\)) or abstract (\(\text{abs}\)).

Example 3 (class): For the Stack class from Figure 2 the set of constructors is given by $c = \{\text{Stack}\}$. There are no public or protected fields in the example. Therefore, $f_{\text{pub}} = \emptyset$ and $f_{\text{prot}} = \emptyset$. There is one private field: $f_{\text{priv}} = \{\text{size}\}$.

Hence $f$ is $f = f_{\text{pub}} \cup f_{\text{prot}} \cup f_{\text{priv}} = \emptyset \cup \emptyset \cup \{\text{size}\}$. There are two concrete methods that do not return a value, namely $\text{push}(\text{double})$ and $\text{pop}()$. Thus $\text{con}m_{\text{pub}} = \{\text{push}(\text{double}), \text{pop}()\}$. The two other methods of the Stack do return a value and are concrete, leading to $\text{con}m_{\text{void}} = \{\text{peek}(), \text{size}()\}$. As there are no more methods in this example, all other sets of methods are empty. The method set $m$ is hence given by

$$m = \text{con}m_{\text{void}} \cup \text{con}m_{\text{pub}} \cup \text{abs}m_{\text{void}} \cup \ldots \cup \text{abs}m_{\text{priv}}$$

$$m = \{\text{push}(\text{double}), \text{pop}()\} \cup \{\text{peek}(), \text{size}()\} \cup \emptyset \cup \ldots \cup \emptyset$$

The Stack class is therefore given as

$$C(\text{Stack}) = \{(\text{Stack}()), \{\text{push}(\text{double}), \text{pop}(), \text{peek}(), \text{size}()\} \}$$

The external state of a class are those class members that can be observed from outside the class. Imposing constraints on a parameter to the method under test is similar to specify some constraints on the external state of the object.

Definition 3 (external state): For a class $C = (c, f_{\text{pub}} \cup f_{\text{prot}} \cup f_{\text{priv}}, \text{con}m_{\text{void}} \cup \text{con}m_{\text{pub}} \cup \text{abs}m_{\text{void}} \cup \ldots \cup \text{abs}m_{\text{priv}})$ the external state is the set $ES(C)$ of all public accessible fields and non-void methods:

$$ES(C) = \text{con}m_{\text{void}} \cup f_{\text{pub}}$$

Example 4 (external state): Using the Stack from Figure 2 for Definition 3 we can write its external state as:

$$\text{con}m_{\text{void}} = \{\text{size}(), \text{peek}()\}$$

$$f_{\text{pub}} = \emptyset$$

$$ES(\text{Stack}) = \{\text{size}(), \text{peek}()\}$$

Whereas the external state allows to observe the state an object is currently in, state manipulators are necessary to change it.

Definition 4 (state manipulators): We define the set of state manipulators $SM(C)$ of a class $C = (c, f_{\text{pub}} \cup f_{\text{prot}} \cup f_{\text{priv}}, \text{con}m_{\text{void}} \cup \text{con}m_{\text{pub}} \cup \text{abs}m_{\text{void}} \cup \ldots \cup \text{abs}m_{\text{priv}})$ that might manipulate the object’s external state $ES(C)$ as the union of public available constructors and concrete methods:

$$SM(C) = c \cup \text{con}m_{\text{pub}} \cup \text{con}m_{\text{void}}$$

Example 5 (state manipulators): Given the Stack class from Figure 2, the set of constructors is $c = \{\text{Stack}\}$, all public concrete methods with non-void return type are $\text{con}m_{\text{pub}} = \{\text{size}(), \text{peek}()\}$ and the set of all public concrete methods with void return type is given by $\text{con}m_{\text{void}} = \{\text{push}(\text{double}), \text{pop}()\}$. Therefore, the set of state manipulators is

$$SM(\text{Stack}) = \{\text{Stack}(), \text{push}(\text{double}), \text{pop}(), \text{peek}(), \text{size}\}$$

Furthermore, we use the Design by Contract™ specification of a class or method. We therefore have to define the contract of a class.

Definition 5 (class contract): The Contract of a class $C$ is given by $\text{DbC}(C) = (m, f, \text{inv}, mc)$, where $m$ is the set of model fields in the classes contract and $\text{inv}$ is the invariant of the class $C$. Furthermore, $mc$ is the set of method specifications $mc = \{m_1, \ldots, m_k\}$ where one method specification belongs to one constructor $c_j \in c$ or one method $m_k \in m$ of class $C$.

One method specification $m$ is defined as the tuple of $ms = (\text{specs}, \text{pure})$ where $\text{pure}$ is a specification element that tells the system that no changes to the objects state happen when invoking the annotated method. $\text{specs}$ is a set of pre- and postcondition pairs $\text{spec} = (P, Q)$. Thus we can write $mc$ as:

$$mc = \{\{(P_1, Q_1^1), \ldots, (P_1, Q_n^1)\}, \text{pure}, \ldots, \{(P_m, Q_n^m), \ldots, (P_m, Q_n^m)\}, \text{pure}\}$$

If a pre- or postcondition is not given, we define it to be true.

3. APPROACH

Alana automatically solves the problem of generating test input data that fulfills the precondition of the method under test by means of AI planning. State-of-the-art planners are used to calculate a method sequence for each non-primitive parameter type of the method under test to instantiate the object and transform it to a state where the precondition of the method under test is fulfilled. Figure 3 gives an overview of the involved subsystems and required data for the approach.

Alana is the first fully automated AI planner based approach that can handle more than just boolean variables. Figure 4 shows the four main steps of Alana. For describing the planning domain, goal and the resulting plan, PDDL [6] is used. Therefore, the approach is independent of the used planner. Different planners may be used for different software components, and improved planners in future years will further improve the results and applicability of Alana.
The plan is retranslated to Java code and the instantiated objects are used as parameters for the method under test.

```java
Object AIana(DbC(C), Pmut) {
    D := generatePlanningDomain(DbC(C));
    G={IS, GS} := generateGoalFile(C, Pmut);
    P := runPlanner(D, G);
    object := convertPlanToJava(P);
    return object;
}
```

Figure 3: Overview of A1ana. Based on the specification of the method under test and the specification of the required type, a planning problem is generated. A state-of-the-art planner is used to generate a plan, representing a series of method invocations. The plan is retranslated to Java code and the instantiated objects are used as parameters for the method under test.

### 3.1 Generation of Planning Domain

Based on the Design by Contract specification of the type required as parameter for the method under test, the planning domain is generated.

**Definition 6 (planning domain):** The planning domain is the set \( D \) of actions \( a \); \( D = \{a_1, ..., a_n\} \). An action is the tuple of an action name, parameters, precondition and effect: \( a = (\text{name}, \text{parameters}, P, E) \).

**Example 6 (planning domain):** We use a data structure as an example domain. Assume that there are two actions within the planning domain. One is called `push` and increases the size of the structure by one. This action can be called at any time. Thus its definition is: \( a_{push} = (\text{push}, \emptyset, \text{true}, \text{size} + 1) \). Moreover, there is an action `pop` that reduces the size by one, and is only allowed to be called when the size is greater than zero: \( a_{pop} = (\text{pop}, \emptyset, \text{size} > 0, \text{size} - 1) \). The resulting planning domain now is:

\[
D = \{(\text{push}, \emptyset, \text{true}, \text{size} + 1), (\text{pop}, \emptyset, \text{size} > 0, \text{size} - 1)\}
\]

For each method of the parameters class that is able to transform the state of it, an action in the planning domain is generated. More precisely, in case a method is annotated with multiple pre- / postcondition pairs an action for each of the pairs is created.

Therefore, we have to define the set of action methods, which are those methods that are considered to change the class' external state \( ES(C) \).

**Definition 7 (action method):** We define the set of action methods of a class \( AM(C) = \{ sm \in SM(C) | Pure(sm) \} \) by the set of state manipulating methods \( SM(C) \) that are not annotated with pure \( Pure(sm) \). Where \( Pure(sm) \) returns the pure value from the method specification \( ms = \langle \text{spec}, \text{pure} \rangle \) from the class contract \( DbC(C) = \{mf, inv, mc\} \) with \( mc = \{ms_1, ..., ms_n\} \).

**Example 7 (action method):** The action methods for the Stack from Figure 2 is the set of those \( SM(Stack) \) that are not annotated with pure:

\[
AM(Stack) = \{\text{Stack()}, \text{push(double)}, \text{pop()}, \text{peek()}\}
\]

Furthermore, we have to define the interpretation function, which converts Design by Contract pre- and postconditions to plan precondition and plan effects, respectively.

**Definition 8 (interpretation functions):** We define \( \Phi_{pre}(P) \rightarrow \phi_{pre}^i \) and \( \Phi_{post}(Q) \rightarrow \phi_{post}^e \) to be two interpretation functions for transforming a pre- and postcondition to a plan precondition and plan effect, respectively.

**Example 8 (interpretation functions):** Using the Design by Contract specification of the pop() method from the running example in Figure 2 will result in \( \Phi_{pre}(size() > 0) = (> size 0) \) and \( \Phi_{post}(size()) == 0 \text{Old}(size()) - 1 = (== size (increase old\_size 1)) \). Where the variable \( old\_size \) denotes the value of size in the pre-state.

Since the planner does not know anything about Java objects, the interpretation function has to take care of required method sequences. For example, the constructor has to be called before any method is executed on the object. Furthermore, calling a constructor twice should be avoided. Therefore, the interpretation function adds a predicate `instantiated` to each methods precondition and effect transformation. For a constructor, the precondition and postcondition are extended by the predicates `not(instantiated)` and `instantiated`, respectively. This guarantees that the planner will first choose a constructor and then call methods on the instantiated object. Destructor’s or finalize methods are not converted to plan actions, since it does not make sense to first instantiate a Java object and then destruct it before using it as test input.

Since PDDL does not support the full range of characters of Java, and each combination of method and pre-/postcondition pair has to be uniquely identified, we introduce name mangling functions.

**Definition 9 (name mangling function):** The name mangling function \( \mu_{inout}(m) \) uniquely translates the given method \( m \) from the input language to the output language such that \( \mu_{inout}(\mu_{out}(m)) = m \) holds.
Example 9 (name mangling function): Applying the name mangling function on the push(Double) method of the Stack object from Figure 2 will result in
\[
\text{push}(\text{Double}) = \text{push\_Double}\_\text{value}
\]

Putting all those definitions together let us finally define the transformation from a class contract to the planning domain. For every pre-/postcondition pair of every action method of the requested class an action specification is generated using the mangled name and the interpretations of the pre- and postcondition, respectively.

Definition 10 (class transformation): We define the transformation of a class \( C = \langle c, f_{\text{pub}} \cup f_{\text{prot}} \cup f_{\text{priv}}, \text{con}\_\text{void}, \text{con}\_\text{pub}, \text{con}\_\text{priv}\rangle \) and a contract \( DbC(C) = \langle mf, \text{inv}, mc \rangle \) to the planning domain \( D = \{a_1, \ldots, a_n\} \) as:
\[
D = \forall_{\text{am} \in \text{AM}(C)} \forall_{\text{spec} \in \text{am}} \forall_{a_{\text{am},\text{spec}}} = (\mu_{\text{java}}(am), \emptyset, \Phi_{\text{pre}}(\text{spec}_{\text{pre}}), \Phi_{\text{post}}(\text{spec}_{\text{post}}))
\]

3.2 Generation of Goal File

The goal file \( G \) provides information about the initial state \( IS \) and the to achieving goal state \( GS \) of the type. The initial state is the set of all variables used within the planning problem. The values are set to the default values for their type. The postcondition of the chosen constructor is used to update those values specified.

Definition 11 (initial state): We define the initial state \( IS \) to be the set of all predicates that describe the initial state of the world: \( IS = \{p_1, \ldots p_k\} \)

Example 10 (initial state): Using the domain from Example 6 the initial state according to Definition 11 is: \( IS = \{\text{size} == 0\} \)

The planning goal is defined by the precondition of the method under test. \( Alana \) aims to generate test input data that satisfies the method under test precondition, therefore we use it as planning goal. But only parts of the precondition are used that imply a constraint on the object, for which \( Alana \) is requested to generate an instantiation plan. Therefore, \( Alana \) uses the approach introduced by Galler et al. [4] to determine those parts of the precondition that have to be part of the goal state. Definition 12 shows the extraction rules that remove all those parts \( o \) from the precondition that do not impose a constraint on the object \( m \) for which currently an instantiation sequence is created.

Definition 12 (relevant specification): We define the relevant specification \( RS \) of a specification \( P \) with respect to a concrete instance of a class \( I \) as \( RS(P,I) \). \( RS(P,I) \) is calculated by iteratively applying the following extraction rules as long as no rule is applicable anymore. \( m_1 \) to \( m_n \) are expressions that reference a state variable of \( I \), i.e., \( m_i \in ES(I) \).
\[
\begin{align*}
(m_1 \& \& o) \& \& m_2 & \rightarrow m_1 \& \& m_2 \\
(m_1 || o) \& \& m_2 & \rightarrow m_1 \& \& m_2 \\
(m_1 \& \& o) || m_2 & \rightarrow m_1 \\
(m_1 || o) || m_2 & \rightarrow m_1 \\
m_1 || m_2 & \rightarrow m_1
\end{align*}
\]

3.3 Create Plan

The planning domain, the initial state and the goal state are transformed to the planning domain description language (PDDL) [6]. PDDL is used as default input language for all planners taking part at the international planning competition (IPC) [1]. Depending on the required functionality of the planner \( Alana \) can use different planners to solve the problem of instantiating complex objects that satisfy the precondition of the method under test. After execution the planner returns a plan.

Example 12 (goal state): Using the example from Figure 2, the method under test precondition \( \text{"size() >= 2"} \) and the to be constructed parameter \( I = \text{\textbf{stack}} \). The goal state is given by \( GS = \{\text{size}() >= 2\} \)

Example 13 (plan): Considering the running example from Figure 2, the method under test requires a Stack object with at least two elements, will result in a plan
\[
P = \{\text{Stack}, \text{push\_Double}, \text{push\_Double}\}
\]

The plan is returned in an IPC compliant format as well.

3.4 Instantiate Parameter Object

The calculated plan has to be transformed to actual method calls in Java as last step of the \( Alana \) approach.

Each action \( a_i = (\text{name}, \emptyset, P_i, E_i) \) can be uniquely mapped to a concrete Java method and a precondition. The plan does not contain any information about actual values for the parameters required by the identified Java method. To instantiate the object with respect to the plan it is necessary to provide concrete parameters as well. Therefore, random values are used for primitive type parameters and \( Alana \) is called recursively for all non-primitive parameters. The new planning problem is defined by all actions provided by the parameters type and the precondition of the action \( P_i \) from the original plan.
Example 14: Consider the generated plan from Example 13. In more detail the plan is given by

\[ P = \{ \langle \text{push} \_1, 0, \text{"true"} \rangle, \]
\[ \ "size() == @Old(size()), int + 1" \}, \]
\[ \langle \text{push} \_1, 0, \text{"true"} \rangle, \]
\[ \ "size() == @Old(size()), int + 1" \} \]

To instantiate the \textit{push()} method call \textit{Alana} has to generate a value for each parameter, in this case a double value for the only parameter of \textit{push()}. Therefore, it recursively calls the test data generation algorithm and provides the precondition specification for the corresponding action, i.e. \textit{“true”} in this case.

Using this recursive approach, simplifies the planning domain and reduces plan generation time.

3.5 Limitations
The presented approach has two limitations. First, PDDL does not support quantifiers. Since \textit{Alana} uses PDDL to specify the planning domain, specifications that include quantifier statements cannot be translated. Second, PDDL interprets action effect declarations as calculation directives. Therefore, \textit{Alana} can only translate post-conditions to PDDL effects that do provide explicit calculation directives, such as \textit{size() == @Old(size())} + 1. In case the post-condition does look like \textit{@Return == size()}, \textit{Alana} cannot create a correct PDDL action effect description.

Both limitations may be removed by adapting \textit{Alana} to directly communicate with a planner, hence bypassing PDDL. In this case the planner can be implemented to unroll the \textit{@ForAll} and \textit{@Exists} statements at test execution time and interpreting the action effects as description of the world and not as calculation directives. But both limitations do not occur in any of the case studies.

4. EMPIRICAL EVALUATION
We evaluate \textit{Alana} using two case studies written in Java. One of it is a student project, a stack based calculator. The second one was provided by our industrial partner. \textit{Alana} significantly surpasses a random generation strategy used as benchmark problem, with respect to the number of successfully generated tests (function coverage) and line coverage.

4.1 Case Studies
\textit{StackCalc} is the student project implementation of a stack based calculator. There are a total of 19 different operator implementations, that operate on a stack. They require the stack to be in a specific state. Furthermore, this case study contains the factory design pattern and a textual input/output interface. The needed specifications where added by the authors.

\textit{BillingSoftware} is a real-world application, developed by our industry partner. It is an event processing framework, expecting these events to be configured specifically. The annotations where created before the actual implementation. The \textit{BillingSoftware} case study contains twice as many post-conditions than the \textit{StackCalc} case study (see in Table 1).

4.2 Experimental Setup
We compare \textit{Alana} with \textit{JET}, developed by Cheon et al. [2]. \textit{JET} was adapted to use \textit{Modern Jass} [14] Design

<table>
<thead>
<tr>
<th>attribute</th>
<th>StackCalc</th>
<th>BillingSoftware</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>51</td>
<td>41</td>
</tr>
<tr>
<td>methods</td>
<td>147</td>
<td>249</td>
</tr>
<tr>
<td>NCSS</td>
<td>868</td>
<td>1456</td>
</tr>
<tr>
<td>@Invariant</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>@Pre</td>
<td>52</td>
<td>65</td>
</tr>
<tr>
<td>@Post</td>
<td>69</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 1: Case studies in figures (NCSS = non comment source statements).

by Contract™ annotations, instead of JML [10]. It aims on generating test data using a random strategy.

For each test run both approaches were asked to generate a unit test for each method of both case studies. Each approach had one single shot for each method under test. After test generation all tests are classified as either succeeding, failing or meaningless. Succeeding tests both fulfill the precondition and the postcondition of the MUT. Invocations of the method under test that raise a postcondition error are classified as failing. These tests reveal an implementation error. Tests that raise a precondition error are labeled as meaningless. Thus, the parameters generated do not satisfy the precondition of the method under test, stating that the generation strategy failed to create a valid object as parameter.

Only those tests that are classified as succeeding or failing are exported to unit tests.

4.3 Results
\textit{Alana} is able to generate complex objects for both case studies. For the \textit{StackCalc} case study \textit{Alana} is able to construct test input data that satisfies the precondition of the method under test for about 50% more methods than random (see Table 2). Using tests generated by \textit{Alana} increases line coverage by about 25% as pointed out in Table 3; branch coverage increased by 49%.

For the \textit{BillingSoftware} case study \textit{Alana} is able to test 45% more methods than random, as given in Table 4. For this case study line coverage doubled as shown in Table 5. Branch coverage only slightly increased due to input parameter checks within the method body. Those checks are not necessary due to the usage of Design by Contract™, but our industry partner had to ensure that in the productive environment meaningful exceptions are thrown in case of wrong input. Since valid input data is provided those branches are never executed, resulting in an only slightly increased branch coverage.

The deviation for the \textit{Alana} approach is due to its still random generation of primitive parameter values.

Using a planner increases test generation time. Table 6 shows the average generation time for both case studies (overall) in seconds. In addition, row “per succ.” present the average time required to generate one successful test. For the \textit{StackCalc} case study the planning domain and the resulting plan are small and trivial. Therefore, the overhead is relatively small compared to the \textit{BillingSoftware} case study. For the \textit{BillingSoftware} case study \textit{Alana} needs three times as much time as the random strategy. This is due to the complex object construction problem for the \textit{BillingSoftware} case study. The longest object construction sequence.
Table 2: Number of generated tests within the StackCalc case study. The amount of meaningless tests drops significantly when using Alana.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Alana</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. (stdev.) succeeding</td>
<td>83.40 (5.24)</td>
<td>56.20 (0.98)</td>
</tr>
<tr>
<td>avg. (stdev.) failing</td>
<td>7.40 (5.24)</td>
<td>4.00 (1.26)</td>
</tr>
<tr>
<td>avg. (stdev.) meaningless</td>
<td>17.80 (2.40)</td>
<td>49.60 (1.02)</td>
</tr>
</tbody>
</table>

Table 3: Line and branch coverage of the generated tests of the StackCalc case study.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Alana</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. (stdev.) line coverage</td>
<td>0.71 (0.02)</td>
<td>0.57 (0.01)</td>
</tr>
<tr>
<td>avg. (stdev.) branch coverage</td>
<td>0.34 (0.06)</td>
<td>0.24 (0.03)</td>
</tr>
</tbody>
</table>

Table 4: Number of generated tests of the BillingSoftware case study. Using Alana, the amount of meaningless test drops significantly. 6 test generation attempts raised an exception during generation of the test.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Alana</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. (stdev.) succeeding</td>
<td>185.53 (2.97)</td>
<td>126.97 (2.21)</td>
</tr>
<tr>
<td>avg. (stdev.) failing</td>
<td>6.13 (2.97)</td>
<td>5.62 (0.66)</td>
</tr>
<tr>
<td>avg. (stdev.) meaningless</td>
<td>51.34 (2.98)</td>
<td>110.41 (1.98)</td>
</tr>
<tr>
<td>exceptional</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5: Line and branch coverage for the BillingSoftware case study.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Alana</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. (stdev.) line coverage</td>
<td>0.40 (0.02)</td>
<td>0.20 (0.01)</td>
</tr>
<tr>
<td>avg. (stdev.) branch coverage</td>
<td>0.18 (0.03)</td>
<td>0.16 (0.01)</td>
</tr>
</tbody>
</table>

Table 6: Average test generation time in seconds over all 100 runs.

<table>
<thead>
<tr>
<th>Software</th>
<th>Alana</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackCalc</td>
<td>overall per succ.</td>
<td>162.07</td>
</tr>
<tr>
<td>BillingSoftware</td>
<td>overall per succ.</td>
<td>495.96</td>
</tr>
</tbody>
</table>

necessary to fulfill the method under tests precondition included 30 method invocations. Each of them requires again objects as parameters in a given state.

But with respect to successfully generated tests the overhead decreases for both case studies. Important to notice is that Alana was able to test 67 methods from the BillingSoftware case study, which random never did in any of the 100 experiment iterations. Manually inspecting those methods leads us to the conclusion that all those cases are methods where very complex objects, have to be constructed. It is just very unlikely that a random approach chooses 30 times in a row the correct method. In turn the random strategy was able to generate a test for one method Alana could not test. This is due to an unsupported Design by Contract specification, i.e., the Alana implementation does not support the “instanceof” operator in a precondition specification.

5. RELATED WORK

Related work can be categorized into approaches that use planners as we do, and approaches that use different generation strategies. We start with the former category:

Howe et al. [18] work on using an AI planner to find a sequence of method invocations, that fulfill a given precondition. In contrast to Alana, the approach of Howe et al. needs human interaction when specifying the planning domain and goal. Both steps are automated within Alana.

Eiffel [12], an object oriented programming language natively supports Design by Contract, Leitner et al. [11] use these Design by Contract specifications and translate them into a planning domain. They support boolean statements and integers with a precision of 2 bit only. Furthermore, they are not able to generate parameters for method invocations created by the planner. Alana can create parameters of any type by recursively calling itself for non-primitive data types and using a random approach for all primitive data types.

Dingels et al. [3] compared a series of planners with respect to their capability to generate series of method invocations that form up a JUnit test. The planning domain is extracted from a proprietary annotation format, which has to be generated for the purpose of test generation. Moreover, they only support boolean predicates. The Design by Contract annotations Alana uses, does not have to be created for test data generation only, and is not limited to boolean data types.

The following paragraphs describe related approaches that rely on different strategies:

Mark Harman [8] works on search-based algorithm for test data generation, based on evolutionary algorithm. They use the concepts of mutation and cross-over, inspired by biology, to modify already generated test data. Selection of the data to modify is done using a fitness function related to branch coverage.

Tonella et al. [16] also work on evolutionary methods for test generation for Java classes. In their approach, a chromosome encodes a whole test case, containing the methods to be called and all parameter’s values. The fitness of such a chromosome is determined by amount of test targets covered, whereas a test target can be a line of code or a whole branch. Thus, determination of test case fitness relies on examining the source code of the software under test.

DART was introduced by Godefroid et al. [7]. Its generation process uses both symbolic execution and real execution of the software under test. After generating a set of input data randomly, the software under test is executed and branching conditions are extracted. For the next data set, these branching criterion are altered. A SAT-solver is used to generate a new sequence fulfilling these altered branching conditions. Thus a new data set is generated, which tests an alternative branch.

Visser et al. introduced JPF, the Java Path Finder [17]. It uses model checking and symbolic execution to generate input data. Using symbolic execution of branching conditions enables JPF to generate input values, that lead to the execution of desired code paths. PEX [15] is developed by Microsoft. As JPF, it uses symbolic execution of branch conditions to generate input data for .NET programs that leads to the execution of specific branches.

All the above mentioned test generation techniques as
JPF, PEX, DART and the evolutionary approaches rely on the availability of the *software under test*’s source code. Our approach is able to generate test data for methods where no code is available, as long as the Design by Contract specification is retrievable. This enables AIana to generate test data for methods of external libraries that are used within a given *software under test*.

6. CONCLUSION

We presented AIana, a test data generation approach for object oriented software systems. It is able to instantiate an object such that it satisfies the precondition of the method under test. Therefore, it transforms Design by Contract annotations into a planning problem. This planning problem is passed to an existing state-of-the-art planner that returns a plan, i.e., an ordered list of actions. Each action corresponds to a method of the Java class. Therefore, AIana remaps the plan to method sequence calls. Parameters required by those methods are generated recursively by either calling AIana again for non-primitive data types or a random value generator for primitive data types.

Evaluating AIana on two case studies shows that it is very useful for generating complex objects as test input data. It clearly outperforms the random strategy with respect to methods tested (function coverage) and line coverage. Very promising is the result for the BillingSoftware case study, which is a real-world application developed and annotated with Design by Contract specification by our industry partner. 67 methods could never be tested by random. 47% more methods were tested on average by AIana. Line coverage could be doubled. The industrial case study shows that AIana perfectly handles non-complete specifications as well. Only those method behaviors that are necessary to achieve the goal state have to be present. Nevertheless, AIana is an approach for generating objects and it is therefore not useful for testing methods that perform heavy mathematical calculations.

Our future work will focus on integrating a planner directly into AIana, which will let us get rid of the limitations imposed by PDDL. Such as no support for quantifiers and requiring action effects to be written as calculation directives, instead of formal world descriptions.

7. ACKNOWLEDGMENTS

The research herein is partially conducted within the competence network Softnet Austria ([www.soft-net.at](http://www.soft-net.at)) and funded by the Austrian Federal Ministry of Economics (bm.wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

8. REFERENCES


