Directed Generation of Test Data for Static Semantics Checker

M.V. Arkhipova and S.V. Zelenov

Institute for System Programming of the Russian Academy of Sciences
{maryn, zelenov} @ ispras.ru
http://www.unitesk.com

Abstract. We present an automatic method, named SemaTESK[1], for generation of test sets for a translator front end. We focus on the validation and verification of static semantics checker. Most the know methods for semantics test generation produce test suites by filtering a pregenerated set of random texts in the target language. In contrast, SemaTESK allows to generate tests for context conditions directly. It significantly reduces generation time and allows reaching completeness criteria defined in the paper. The presented method to specify static semantics allows to formalize informal requirements described in normative documents (e.g. standard). The method includes SRL notation for compact formal specification of context conditions and STG tool for efficient generation of test suite from SRL specification.
The SemaTESK method has been used in a number of projects, including testing static semantics checkers of C and Java.

Keywords: automated test data generation, context condition, grammar, specification based testing, static semantics.

1 Introduction

Formal languages are widely used in many areas of IT: Programming languages are the main instruments in software development; query languages are used to manage databases; markup languages are used in various document processing systems (e.g. browsers, text processors), etc. Translator is a program that converts text written in a formal language into some appropriate form. For example, a compiler translates a program into an executable form, a DBMS translates a query written in a high level query language (e.g. SQL) into sequence of low level operations on DB, a browser translates an information page into drawing commands, etc. Defects in a translator break entities resulting from translation: their properties differs from what is specified in the language specification. For example, defects in executable entities induced by erroneous compiler are hard to detect and find a workaround, thus correctness of executables obtained from an incorrect compiler is always a doubt. Validation and verification of a translator is an important activity for dissemination of the translator in industry.

[1] *SemaTESK* stands for “Semantics Testing Kit”.

Validation and verification of translators is always complicated. The main source of difficulties is complexity of input and output: the input is a document with a furcated syntax structure and rich set of context constraints imposed by the language specification, the output is written in machine or intermediate language and possesses similar or even higher degree of complexity. The usual way to cope with complications of translator validation and verification is decomposition the validation and verification task into several subtasks that in total cover the whole functionality of the translator. Typical translator includes the following set of functions:

1. analysis of syntax correctness and parsing of input text;
2. checking static semantics[2] of the input;
3. generation of the output.

In this paper, we focus on the task of validation and verification of static semantics checker. We treat *static semantics* as a synonym to *context conditions*. Within this paper a static semantics checker is a boolean function of the form $f : S(L) \rightarrow B$, where $S(L)$ is a set of syntactically correct strings in the given formal language L, and $B = \{true, false\}$. $f(s)$ takes a value of $true$ if string s satisfies all context conditions of language L and takes a value of $false$ if string s violates at least one of the context conditions of the language L.

We use testing [1] based on formal specifications and models [2] as the primary tool for validation and verification. In the course of testing, one checks a software under test for quality on some specially created test input data. There is the following problem: The set of test data should be representative, so that results of the testing do reflect real quality of a software under test. Another problem is: Usually, there are too many situations that should be tested. So, it is practically impossible to create tests by hand. Usage of formal description of a software under test allows both formulating appropriate test completeness criteria and generating tests automatically.

1.1 Related Approaches

The most traditional way to specify language properties uses grammar of some appropriate kind.

In order to generate semantically correct tests, the approaches presented in works [3–5] use grammars in the form of extended BNF supplied with special code fragments that incorporate semantics-related information: *actions* contain some calculations, *guards* are used to check conditions that allow generating corresponding part of test. However, such form of a grammar[3] is not a specification of a language. It rather seems like a program for generating tests.

[2] Given a formal language, a *static semantics* describes properties of the language that may be checked at compile-time (such as scoping and static typing constraints), whereas a *dynamic semantics* describes run-time properties of the language ("meaning" of language constructs).

[3] Sirer and Bershad in [5] call it a *production grammar*.

Boyapati et al. [6] presents the Korat test data generator. This generator uses a specification of desired test data in the form of Java-method that checks correctness of data structure, and parameters that restrict the set of all possible test data to some finite set. Khurshid and Marinov [7] presents the TestEra framework for specification-based testing of Java programs. Specifications are first-order logic formulae written in Alloy declarative language [8]. TestEra also requires a bound that limits the size of the test cases to be generated. Unfortunately, both Korat and TestEra can not be provided with a domain-specific completeness criteria. They generate *all* non-isomorphic structures that match given restrictions. Besides, in the case of semantics checker testing, it is practically impossible to have a (either reference or "under test") checker written in Java.

Daniel et al. [9] present a method for automated testing of refactoring engines. In order to generate tests for some refactoring, one should develop corresponding generator on the basis of specific ASTGen library. The main disadvantage of this approach is that test data generators are developed manually.

Harm and Lämmel [10, 11] present approaches to automated test data generation for static semantics checker based on usage specification of semantics in the form of attribute grammars (AG) [12]. Kalinov et al. [13] suggest another approach to this task that uses specification in the form of Gurevich's ASM [14]. The authors of these approaches consider various test coverage criteria for static semantics checker testing and suggest corresponding automated test generators that work as follows: First, the generator creates a set of syntactically correct sentences; Next, the generator checks semantical correctness of every generated sentence with the help of appropriate interpreter of AG or ASM specification of semantics; All semantically incorrect sentences are rejected.

Thus, in those approaches, specification of semantics is used only for the following purposes: To formulate coverage criteria and to check semantical correctness of generated sentences. Tests generation is syntax-directed. In other words, specification of semantics is not used to generate semantically correct sentences directly. In general case, the generation process is very time-consuming: Generator has to create millions of syntactically correct sentences in order to obtain several hundreds of semantically correct tests. In papers [11, 15], the authors of these approaches suggest some optimizations of generation process: They try to reveal semantic incorrectness of subsentences of generated sentences "as soon as possible".

Here we proceed with discussion of AG-related approaches. Similar arguments are also applicable to approaches based on ASM. AG is suitable for developing checkers, but it seems not suitable for directed generation of tests. One can treat an AG specification of language semantics as a predicate P over abstract syntax trees. In order to create semantically correct test, one should find a tree t such that $P(t) = true$. To do this, the AG-related approaches construct syntactically correct trees t and check value $P(t)$ for every t.

We believe that more effective way to generate semantically correct tests is to use an algorithm that directly creates some solutions of the equation $P(t) = true$.

The form of classical AG is the main obstacle to solve this equation. AG is a very powerful tool. However, it has several essential weaknesses that are shown in the following example.

Example 1. Let us consider some procedural programming language that requires separate statements for variable declarations and assignments. There are two context conditions:

1. All names of variables declared in one procedure must be different;
2. Name of an assigned variable must be declared in the same procedure.

In the classical AG approach, the corresponding part of attribute grammar for this language looks as follows:

```
1: Procedure ::= ( Stmt )*
2:  { attribute SymbolTable vars;
3:    attribute Boolean ok;  };
4: Stmt ::= VarDecl | Assignment | ... ;
5: VarDecl ::= ‘‘var’’ <name:ID>
6:  { Procedure.ok &= !Procedure.vars.has( name );
7:    Procedure.vars.add( name );  };
8: Assignment ::= <name:ID> ‘‘=’’ ...
9:  { Procedure.ok &= Procedure.vars.has( name );  };
```

In this grammar, there is the symbol table `vars` declared in the `Procedure` rule that collects information about names of variables declared in one procedure. The table is updated in the `VarDecl` rule and checked in the `VarDecl` and `Assignment` rules. The checking results are stored in the boolean attribute `ok` declared in the `Procedure` rule.

This example uncovers the following weaknesses of classical AG:

– The attributes declared in the `Procedure` rule are sort of *global variables*, which yields well known *problems in maintenance* of a grammar.
– In order to formalize *a context condition*, one has to write many lines of code in *several different parts* of a grammar: For example, the second context condition (see above) is formalized in four lines (2, 3, 7, and 9) that relate to three different rules. This yields very *weak traceability.*

We suppose that the weaknesses stated above do not allow creating semantically correct tests directly on the basis of a language semantics description in AG form. Indeed, a generator must be very intelligent to understand what it should do in order to resolve a context condition.

In this paper, we present the SemaTESK method aimed at automated generation of tests for static semantics checkers. The SemaTESK method is based on UniTESK approach [16, 17] that belongs to the family of specification-based approaches to testing. The SemaTESK method includes an appropriate language called SRL[4]. The purpose of SRL is to write formal specifications of static semantics in a form that is suitable for directed generation of tests. The SemaTESK

[4] *SRL* stands for “Semantics Relation Language”

method is supported by a test case generator called STG[5] that allows automated generating of tests on the basis of static semantics specifications written in SRL.

The remainder of the paper is organized as follows. In Section 2 we present the SRL language. In Section 3 we formulate completeness criteria. In Section 4 we describe the STG generator. In Section 5 we show several application examples and discuss benefits of the SemaTESK method. Section 6 contains some discussion. In Section 7 the paper is concluded.

2 Semantics Relation Language

2.1 Peculiarities of SRL

We start with the following example:

Example 2. The context condition "Name of an assigned variable must be declared in the same procedure" from Example 1 written in SRL looks as follows:

```
one-to-many relation DeclareAssignedVarName {
  ordered  equal
  target Assignment {name}
  source VarDecl {name}
  context: same Procedure  }
```

One can read this *context condition descriptor* as follows: If there is an occurrence of `Assignment` in a sentence, then there must exist an occurrence of `VarDecl` in the `same Procedure` such that the attribute `name` of the `VarDecl` is `equal` to the attribute `name` of the `Assignment`, the `Assignment` must be in succession to the `VarDecl` (i.e. the `Assignment` and `VarDecl` must be `ordered`), and the `name` declared in *one* `VarDecl` may be used in *many* `Assignment`s (cf. `one-to-many` keyword).

This example shows the following peculiarities of SRL:

– The static semantics of a language are formalized as context condition descriptors over an attributed context-free grammar. We do not relate a context condition descriptor to a particular grammar production rule, as in AG, since in many cases it is difficult to choose pertinent production rule (e.g. in Example 2, do the context condition relates to `Assignment`? or to `VarDecl`? or to `Procedure`? As Example 1 shows, in the AG-related approach, different parts of code of this context condition relate to all those three rules!).
– One context condition from a language specification expressed in a natural language corresponds to one block of code (i.e. context condition descriptor) written in SRL. This yields efficient traceability.
– Context condition descriptors have a form of "item declaration — item usage" relations[6]. The core of a context condition descriptor is a pair of constructions started with keywords `target` and `source`. Generally speaking,

[5] *STG* stands for "Semantic Tests Generator"

[6] The practice shows, that in the majority of cases, context conditions are managed to be specified in such manner but, when it is necessary, context condition specification can be appended by Java code

for any occurrence of `target` (i.e. an item usage), there must exist an appropriate occurrence of `source` (i.e. an item declaration) that meets the context condition.

At present, both the SRL language and the STG generator are still evolving. Every new application of the STG generator uncovers new ways the generator could be improved "if only SRL had this new feature". Nevertheless, the underlying principle of SRL remains firm, and this is the subject of the rest of this section.

2.2 The Form of Underlying Grammar

In order to formalize a context-free grammar, we use the TreeDL[7] language [18]. The purpose of TreeDL is to describe structure of abstract syntax trees. TreeDL allows describing structure of tree nodes that includes specification of children nodes and additional attributes.

Example 3. The context-free part of the grammar presented in Example 1 may be described in TreeDL as follows:

```
node Procedure { child Stmt* statements; }
abstract node Stmt {}
node VarDecl : Stmt { child ID name; }
node Assignment : Stmt { child ID name; ... }
node ID { attribute string value; }
```

The TreeDL form of a grammar has the following advantages over BNF:

– All children and attributes in one node are named; So, each child or attribute can be unambiguously addressed.
– One can add some additional (e.g. semantics related) attributes into nodes.

2.3 Context Condition Descriptor

The main atomic object of a language semantics description written in SRL is a *context condition descriptor* (CCD). Every CCD specifies a relation between two nodes called *source* and *target* in the following sense: A structure of a target node depends on a structure of a source node. In most cases, one can treat target node as a usage of an item and a source node as a declaration of that item. In Example 2, source and target nodes are described by their types:

```
  target Assignment {...}
  source VarDecl {...}
```

This description means that for the CCD under consideration, any node of type `Assignment` may be a target and any node of type `VarDecl` may be a source. In general case, source and target nodes may be described more accurate (see Subsection 2.4). In fact, a CCD specifies a relation between some subtrees of source and target nodes. Those subtrees are described in braces that follows node descriptions. In Example 2, the CCD specifies a relation between the field `name` of a source node and the field `name` of a target node:

[7] *TreeDL* stands for "Tree Description Language"

```
target Assignment {name}
source VarDecl {name}
```

One can treat such subtrees as *arguments* of a CCD. Dependency between the arguments is specified by means of describing an appropriate dependency kind. SRL provides several keywords to specify dependency between source and terget. Those keywords cover almost all context conditions of typical programming languages (see Subsection 2.5). In Example 2, the keyword `equal` is used in the CCD in order to specify that the arguments must be equal.

In some cases, a context condition is restricted to have source and target in some specific context. For instance, in Example 2, both assignment and variable declaration must be in the same procedure. A context is specified in CCD by means of `context` keyword. There are two variants of a context specification.

– If both source and target should be located in the same subtree with the root node of type `<RootNodeType>`, then the following construction is used:

```
context: same <RootNodeType>
```

– If source and target must be in different subtrees with roots of (possibly) different types, then the following construction is used:

```
context: differ source_context <SourceRootNodeType>
                target_context <TargetRootNodeType>
```

If a source node of a CCD must precede the target node, then the CCD should be marked by `ordered` keyword (cf. Example 2). Otherwise, it should be marked by `unordered` keyword.

In order to specify, how many nodes may relate to each other by a dependency imposed by a CCD, one should describe a relation type of the CCD (see Subsection 2.6). In Example 2, the relation type is specified by the keyword `one-to-many` that means that a name declared in *one* `VarDecl` may be used in *many* `Assignment`s.

Here we proceed with detailed description of some SRL constructions used in CCDs.

2.4 Node Description

Source and target nodes are specified by paths over an abstract syntax tree. Such a path has a form of chain $e_1.\cdots.e_n$ of path elements e_i separated by dots. Each path matches a corresponding set of nodes. Such a set is defined inductively as follows. Let S_0 be an empty set, and S_k $(k = 1,\dots,n)$ be a set of nodes that match a path $e_1.\cdots.e_k$. The set S_k $(k = 1,\dots,n)$ depends on S_{k-1} and the kind of the element e_k.

There are the following kinds of path elements:

– `<NodeType>` — if $k = 1$, then such an element matches all nodes N such that type of N is `<NodeType>`; If $k > 1$, then it matches all nodes N such that N is a child of type `<NodeType>` some node from S_{k-1}.

- `<fieldName>` — matches all nodes N such that N is a value of a child or an attribute named `<fieldName>` in some node from S_{k-1}.
- `parent` — matches all nodes N such that N is the parent of some node from S_{k-1}.
- `^<ParentNodeType>` — matches all nodes N such that N is the nearest parent node of type `<ParentNodeType>` for some node from S_{k-1}.
- `target` — matches the target of the current CCD (valid only in path that specifies a source node).
- `context` — matches the context of the current CCD (valid only if context has the "`same`" form).
- `source_context` and `target_context` — match context of the source and the target nodes of the current CCD correspondingly (valid only if context has the "`differ`" form).

Example 4. For trees described in Example 3, the node description `Assignment.name` specifies all names of all assignments, the node description `Assignment.^Procedure.VarDecl` specifies all variable declarations contained in procedures that contain an assignment.

2.5 Dependency Kind

SRL provides the following keywords for specifying a kind of dependency between arguments of a CCD:

- `equal` — means that values of the arguments of the CCD must be equal (more precisely, subtrees that have the arguments of the CCD as roots must be isomorphic).
- `unequal` — means that values of the arguments of the CCD must be different (more precisely, subtrees that have the arguments of the CCD as roots must not be isomorphic).
- `present` — means that if the `source`-related argument of the CCD exists, then the `target`-related argument of the CCD must exist as well.
- `absent` — means that if the `source`-related argument of the CCD exists, then the `target`-related argument of the CCD must not exist.
- `compatible` — means that the `source`-related argument of the CCD must be compatible (in a sense of type compatibility in programming languages, see Subsection 2.7) with the `target`-related argument of the CCD (in this case, the arguments of the CCD must describe types; see Example 6 below).

We understand that in some cases, a dependency imposed by some context condition may have a kind that differs from the kinds listed above. In such a case, one should use the keyword `custom` and provide some additional program code that implements processing the CCD in the test generator (this subject is beyond the scope of this paper, see [19] for details). Practice shows that in a such complex language as Java, there are only two context conditions (of about 300) that requires usage of the `custom` keyword[8].

[8] Those context conditions relates to semantics of method signature.

2.6 Relation Type

Relation type of a CCD specifies, how many nodes may relate to each other by a dependency imposed by the CCD. SRL provides the following keywords for specifying a relation type:

– The keyword `one-to-many` means that *one* source node may correspond to *many* target nodes, such that for any occurrence of a target node, there must exist an appropriate occurrence of a source node that meets the context condition (see Example 2).
– The keyword `many-to-many` means that *many* source nodes may correspond to *many* target nodes, such that any occurrence of a target node and any occurrence of a source node (that differs from the occurrence of the target node) must meet the context condition.
– The keyword `one-node` means that a context condition is imposed on *one node* (more precisely, it is imposed on a subtree that has this node as the root). Such a node is specified as a target node, and location of a source node is addressed relatively to the target node (i.e. `source` construction of a CCD starts with `target`).

Example 5. The context condition "All names of variables declared in one procedure must be different" from Example 1 written in SRL looks as follows:

```
many-to-many relation DifferentVarNames {
  unordered  unequal
  target VarDecl {name}
  source VarDecl {name}
  context: same Procedure  }
```

Example 6. Let the `Assignment` statement from Example 1 has an `Expression` in its RHS. Let us consider the following corresponding TreeDL description:

```
node Assignment : Stmt { child ID name; child Expression rhs; attribute Type lhs_type; }
abstract node Expression { attribute Type type; }
```

In order to describe semantics of types, we add the attribute `lhs_type` to the `Assignment` node and the attribute `type` to the `Expression` node. Thus, the context condition "A variable of one type can store only a value of a compatible type" written in SRL looks as follows:

```
one-node relation AsgnTypes {
  unordered  compatible
  target Assignment {rhs.type}
  source target {lhs_type}  }
```

2.7 Type Compatibility

Given a programming language, the semantics of the language usually has a significant part that relates to semantics of types. Semantics of types is generally reduced to conditions of type compatibility in different contexts.

In SRL, compatible types are specified as follows. Given a language under consideration, one should specify a partially ordered set of types of the language imposed by the compatibility relation. Such a set should be specified by means of chains of linearly ordered subsets. In order to specify a chain of types that are compatible from left to right, one should enumerate the types of the chain in an SRL `typeset` construction.

Example 7. Let us consider a language that contains the following types: `short`, `int`, and `long`. For this language, type compatibility may be specified by the following chain of types:

```
typeset PrimitiveTypes { PrimitiveType.SHORT, PrimitiveType.INT, PrimitiveType.LONG }
```

In this example, constructions `PrimitiveType.SHORT` and the others are the possible values of node attributes that describe types of expressions (cf. the attributes `lhs_type` and `type` in Example 6).

The SRL language also allows describing compatibility of user-defined types. This requires usage of some features of SRL that are beyond the scope of this paper (see [19] for details).

3 Completeness Criteria

3.1 Semantically Correct Tests

Given an SRL specifications, one can formulate the following naive completeness criterion for semantically correct tests: *All CCDs from the specification must be covered* with respect to the following definition:

Definition 1. *A CCD is considered covered iff a test set contains a sentence, such that the corresponding abstract syntax tree contains two nodes that match a pair of the source and the target of the CCD.*

Example 8. Suppose that the language from Example 1 allows using nested blocks in a procedure. A compiler developer will say that the following situations are different for a static semantics checker, whereas they cover the same context condition:

```
                                {
{                                 var A;
  var A;                          {
  A = 1;                            A = 1;
}                                 }
                                }
```

Thus, the criterion “All CCDs” is not sufficient for good testing, and one can improve it by the following way: *All CCDs from the specification must be covered with all possible environments of source and target nodes* with respect to the following definition:

Definition 2. *An environment of a node in an abstract syntax tree is a chain of nodes on the path from the node under consideration to the root of the tree.*

3.2 Semantically Incorrect Tests

Another important task in testing static semantics checkers is to test that a checker rejects incorrect sentences. Given a test that violates some context conditions, one may expect that a static semantics checker rejects this test and provides some appropriate diagnostics. If the test violates several different context conditions, then in general case it is difficult to predict the corresponding diagnostics since a checker under test may exit immediately after detecting only one violated context condition. Thus, we suggest generating such tests that each test violates only one context condition. Suppose that we can generate tests that do meet static semantics. In order to generate tests that violates some context condition, we suggest to generate tests that meet negation of this context condition.

Definition 3. *Given a CCD R, a negation $\tilde{R}$ of R is a CCD such that the following condition holds: if a sentence meets $\tilde{R}$, then the sentence violates R.*

Example 9. An example of negation for the CCD "Name of an assigned variable must be declared in the same procedure" from Example 2 looks as follows:

```
many-to-many relation DeclareAssignedVarName_neg {
  ordered  unequal
  target Assignment {name}
  source VarDecl {name}
  context: same Procedure  }
```

Let S be a specification of a language semantics written in SRL, and $R \in S$. Let us consider the following *R-negation* of the specification: $\tilde{S}^{(R)} = \{\tilde{R}\} \cup S \backslash \{R\}$. One can summarize the above discussion by the following completeness criterion for semantically incorrect tests: *For each CCD R, all negations $\tilde{R}$ must be covered in $\tilde{S}^{(R)}$ with all possible environments of source and target nodes.*

4 Semantic Tests Generator

The purpose of SRL is to describe a language semantics in a form that is suitable for automated generation of tests for static semantics checker. In the method SemaTESK we present in this paper, the test case generator STG generates test sets that meet the completeness criteria formulated above (see Section 3). STG takes a corresponding language grammar in TreeDL and a context conditions specification in SRL. The core of STG is the engine that builds syntax trees according to the grammar and the context conditions. Text builder maps generated trees to concrete documents in the given language. Text builder traverses syntax tree and creates textual elements that correspond to generated syntax nodes. Before we briefly formulate the algorithm of test generation used in STG engine (see [19] for details), let us give the following definitions.

Definition 4. *A subtree of an abstract syntax tree is called a syntactically complete tree if it corresponds to some syntactically correct sentence.*

Definition 5. *A context condition corresponding to some target node and described in some CCD over an abstract syntax tree is resolved if the tree contains all necessary elements (nodes and attributes) in required contexts such that the tree with this target node match the CCD.*

Definition 6. *An abstract syntax tree is called semantically complete if it corresponds to some semantically correct sentence.*

Definition 7. *Given a CCD, a subtree of an abstract syntax tree is called a prime tree if it contains nodes that match specifications of source and target nodes of the CCD.*

One can treat a prime tree for some CCD as a tree that contains only the source node and the target node with their environments. All context conditions in any semantically complete tree are resolved.

The STG generator applies the following algorithm to each CCD from the specification of a language semantics:

1. Given a CCD, the generator creates a set of all possible prime trees[9] (with respect to the given value of the recursion depth).
2. For each prime tree t_{prime}, the generator creates a minimal[10] syntactically complete tree t that contains t_{prime} as a subtree.
3. Given a syntactically complete tree t, the generator tries to create the corresponding semantically complete tree $\bar{t}$ (see below).
4. If the generator successfully creates the tree $\bar{t}$, then it prints its text in the formal language under consideration.

The STG generator uses both attribute dependency graph and syntax tree for stepwise directed creation of tests that meet context conditions. Given a syntax tree obtained at the previous step, the generator searches the tree for unresolved context conditions and, in order to resolve them, creates additional subtrees in the tree. Given a syntactically complete tree t, the STG generator tries to create the corresponding semantically complete tree by the following algorithm:

1. The generator searches the tree t for unresolved CCDs; If all CCDs are resolved, then t is semantically complete.
2. In order to resolve the CCDs found on the previous step, the generator tries to modify the tree by the following rules:
   a) the prime subtree t_{prime} is always invariant;
   b) if for some one-to-many CCD, there is no a source node in the tree, then the generator walks the tree and tries to add new subtree containing a node that match the specification of a source node in the CCD;

[9] This set consists of prime trees that contain source nodes and target nodes (w.r.t. the CCD) in all various possible combinations of environments.

[10] All lists are instantiated with minimal possible size; alternatives are instantiated to the simples variant, e.g. empty or terminal, etc.

   c) if a dependency kind of some CCD requires that the tree must contain some specific node that currently does not exists in the tree, then the generator tries to add new such node (like in rule 2.b).
3. If the generator could not resolve the previously found unresolved CCDs, then the tree is rejected; Otherwise, go to the step 1 of this algorithm, since the tree has been changed in the step 2 and may contain new unresolved CCDs.

5 Case Studies

The SemaTESK method has been approved in the following projects:

– testing IPMP-21 message header processors [20];
– testing the C front-end of the GCC compiler;
– testing the CTESK translator [21] developed in ISP RAS.
– testing the JavaTESK translator [22] developed in ISP RAS.

Some properties of the languages under test are presented in Table 1.

Table 1. Properties of languages under test

Language	Number of CCDs	Size of specification	Tests generated
IPMP-21 XML	4	28 lines	54
C	85	1019 lines	about 10000
Java	278	3350 lines	about 32000

The main purpose of the pilot project on testing IPMP-21 was to demonstrate feasibility of SemaTESK approach to static semantics formalization for the generation of semantically correct XML documents. The purpose of the pilot project on testing GCC was to demonstrate feasibility of SemaTESK approach to static semantics formalization of a complex programming language.

The SemaTESK method has been successfully approved in specifying semantics of C for testing the CTESK translator [21] and in specifying semantics of Java for testing the JavaTESK translator [22]: several bugs have been found in the semantics checkers of translators that had been thoroughly tested before by means of manually developed tests.

We consider a static semantics checker as a boolean function. In SemaTESK, we use an automatic test oracle to run generated semantically correct tests. The oracle considers a test run successful if a semantics checker under test returns true for the given test input. In practice, the true value means that work of the semantics checker completes without any error messages about context conditions violations. To run generated semantically incorrect tests we also use the automatic test oracle that considers a test run successful if a semantics checker under test returns false for the given test input. In practice, the false value means that the semantics checker completes with error messages about some context conditions violations.

Let us estimate benefits from using the SemaTESK method. Suppose that one test for a static semantics checker contains about 10–30 lines of code. Here are the approximate numbers of lines that should be manually written in order to create 10 tests by means of different methods (the estimations are based on the Table 1 and the assumption stated above):

– Manual development – about 100–300 of manually written lines per 10 tests.
– The SemaTESK method – about 1–2 of manually written lines per 10 tests.

Thus, the effort for development of tests by means of the SemaTESK method is about hundred times less than the effort for manual test development.

6 Discussion

We have developed the SemaTESK method just for directed generation of test data for static semantics checkers of formal languages. We doubt whether the presented ideas can be used for generation of efficient static semantics checkers.

We have applied SemaTESK to testing checkers of programming languages most of all. At present, the method is still evolving. Every new application of it may require to improve both SRL and STG. We believe that the method is applicable to testing checkers of formal documents content, telecommunications messages, DB queries, etc as well.

It is of interest to note that the most promising way of SemaTESK use is a generation of tests sets for language dialects under development. Because it is rather simple to change specifications of language dialects and thus the amount of handwork required to make language specification matching current state is reduced. On the other hand, if some context conditions change, then it is much easier to modify several CCDs in a corresponding SRL specifications than to revise all manually written tests that concern the changed context conditions.

7 Conclusions

This paper presents the SemaTESK method that implements specification-based testing approach for static semantics checker testing. The SemaTESK method provides the SRL language for writing formal specifications of static semantics in the form that yields efficient traceability and is suitable for semantics-directed automated generation of tests. The SemaTESK method is supplied by the corresponding test case generator STG that allows generating test sets that meet appropriate completeness criteria formulated on the basis of SRL specifications of a language under test. The STG generator takes SRL specifications as an input and automatically produces both semantically correct and semantically incorrect (with unambiguously stated kind of an incorrectness) tests.

The SemaTESK method has been used in several case studies including testing static semantics checker of such a complex programming languages as C and Java. Obtained practical results prove effectiveness of the SemaTESK method.

References

1. Beizer, B.: Software Testing Techniques. Second edn. van Nostrand Reinhold (1990)
2. Petrenko, A.: Specification based testing: Towards practice. LNCS **2244** (2001) 287–300
3. Duncan, A., Hutchison, J.: Using attributed grammars to test designs and implementation. In: Proceedings of the 5th international conference on Software engineering. (1981) 170–178
4. Guilmette, R.F.: TGGS: A flexible system for generating efficient test case generators (1995)
5. Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: Second Conference on Domain-Specific Languages. (1999) 1–13
6. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on java predicates. In: Proc. of International Symposium on Software Testing and Analysis (ISSTA). (2002)
7. Khurshid, S., Marinov, D.: Testera: Specification-based testing of java programs using sat. Automated Software Engineering Journal **11**(4) (2004) 403–434
8. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2) (2002) 256–290
9. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: ESEC/FSE. (2007) 185–194
10. Harm, J.: Automatic test program generation from formal language specifications. Rostocker Informatik-Berishte **20** (1997) 33–56
11. Harm, J., Lämmel, R.: Two-dimensional approximation coverage. Informatica **24**(3) (2000)
12. Paakki, J.: Attribute grammar paradigms – a high-level methodology in language implementation. ACM Computing Surveys **27**(2) (1995) 196–255
13. Kalinov, A., Kossatchev, A., Posypkin, M., Shishkov, V.: Using ASM specification for automatic test suite generation for mpC parallel programming language compiler. In: Proceedings of Fourth International Workshop on Action Semantic, AS'2002, BRICS note series NS-02-8. (2002) 99–109
14. Gurevich, Y.: Abstract state machines: An overview of the project. LNCS **2942** (2004) 6–13
15. Kossatchev, A., Kutter, P., Posypkin, M.: Automated generation of strictly conforming tests based on formal specification of dynamic semantics of the programming language. Programming and Computing Software **30**(4) (2004) 218 – 229
16. Bourdonov, I., Kossatchev, A., Kuliamin, V., Petrenko, A.: Unitesk test suite architecture. LNCS **2391** (2002) 77–88
17. ISP RAS: UniTESK Technology Web-site. `http://www.unitesk.com/`.
18. Demakov, A.V.: TreeDL: Tree Description Language. Available on: `http://treedl.sourceforge.net/treedl/treedl_en.html`.
19. Arkhipova, M.V.: Automated Generation of Tests for Semantics Analysers in Translators. PhD thesis, Moscow, Russia (2006) (in Russian).
20. ISO/IEC JTC1/SC29/WG11: IPMP: Intellectual Property Management and Protection in MPEG Standards. Available on: `http://www.chiariglione.org/mpeg/standards/ipmp/`.
21. ISP RAS: CTESK: toolkit for testing applications developed in C. Available on: `http://www.unitesk.com/content/category/7/14/33/`.
22. ISP RAS: JavaTESK: Toolkit for testing applications developed in Java. Available on: `http://www.unitesk.com/content/category/7/28/74/`.