

# Object-Oriented Attribute Grammar based Grammatical Approach to Problem Specification

Maria João Varanda  
Instituto Politecnico Bragança, Portugal  
Marjan Mernik, Tomaž Kosar, Viljem Žumer  
Univerza v Mariboru

Pedro Henriques  
Universidade do Minho

August 17, 2004



## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Related Work</b>	<b>6</b>
<b>3</b>	<b>Basic Concepts</b>	<b>8</b>
3.1	Attribute Grammars . . . . .	8
3.2	UML Diagrams . . . . .	11
<b>4</b>	<b>A grammatical approach to problem specification</b>	<b>11</b>
4.1	The Principle . . . . .	12
4.2	Deriving a context-free grammar from a conceptual class diagram . . . . .	15
4.3	Deriving an AG from an operational diagram . . . . .	16
4.4	Generating the rapid prototype of a system . . . . .	17
<b>5</b>	<b>Case Studies: Specification</b>	<b>18</b>
5.1	CS1: Vending Machine . . . . .	19
5.2	CS2: Automatic Teller Machine . . . . .	29
5.3	CS3: Video Store . . . . .	37
5.4	CS4: Cleaning Robot . . . . .	48
<b>6</b>	<b>An OO approach to the implementation of AG-based Processors</b>	<b>56</b>
6.1	LISA system . . . . .	58
6.2	Case Studies Implementation . . . . .	64
6.2.1	CS1: Vending Machine . . . . .	64
6.2.2	CS2: Automatic Teller Machine . . . . .	69
6.2.3	CS3: Video Store . . . . .	74
6.2.4	CS4: Cleaning Robot . . . . .	84
<b>7</b>	<b>Conclusion</b>	<b>91</b>
<b>A</b>	<b>Meta-Grammar for OOAG Descriptions</b>	<b>95</b>



# 1 Introduction

This tutorial aims at introducing a method to specify problems based on attribute grammars (AGs). The *grammatical approach to problem solving* lies upon the success reached by attribute grammars in the context of the specification of language semantics and of the systematic implementation of language processing tools. It is well-known that language processors can be generated automatically from AG; the advantages that we obtain from that approach are obvious: much easier and faster to write and maintain a specification (a grammar) than a C or Java program; more efficient generated processors, as we reuse code developed by compiler experts. Besides that important benefit of creating programs from formal specifications, another relevant lesson comes out from this domain: the syntactic/semantic approach to problem solving—first build the syntax tree, decorate it (evaluating attribute values associated with tree nodes), and then traverse the tree to transform it into the desired output.

In the technical report we propose a grammatical approach to problem specification supported by an attribute grammar developed and written in an object-oriented style (OOAG). We also suggest the use of UML diagrams to model the analysis phase, and we introduce a method to easily obtain from the diagrams the grammar-based specification. Then a rapid prototyping technique will be used to validate the specification in a pragmatic way. The idea is to translate the OOAG obtained in the first step (specification phase) into the concrete syntax of a compiler generator in order to create a simulator for that problem. We can then write scenarios (in the language defined by that OOAG) describing different uses of the system under discussion, and use the generated simulator to process those scenarios computing the desired results.

The main contribution of this document—the proposal of our **method** to approach the complex task of *computer problem solving*—is presented in section 4; we start by discussing the principle that guides such approach (subsection 4.1), and then we present a template that exhibits the notation used to write the object-oriented modular attribute grammar that specifies the problem under analysis (section 5)—the formal definition of the syntax of this meta-language is given in the appendix A via a context free grammar (CFG).

Before that explanation and in order to give the necessary background to understand the rest of the tutorial, section 3 is devoted to the basic concepts and principles supporting UML and Attribute Grammars. About UML we just

explain the diagrams that will be used on the analysis process: Use-Cases will be used to identify the actions and actors involved in the project; Conceptual Class Diagrams describe the entities (classes) and the relationships among them; and Operational Diagrams, derived from the Use-Cases, detail each action. Concerning Attribute Grammars, the notions of attribute, evaluation rule and contextual condition are introduced, as well as the notation used along the report. In that section we also discuss, briefly, the monolithic and modular approaches to AG development and the attribute evaluation process.

Section 5 is also very important because it supports the proposal: to illustrate the strategy introduced in 4.1, we develop the OOAG to specify four case studies (subsections 5.1 to 5.4)—after stating the problem, we present the UML diagrams that results from its analysis, and we describe the structure of the problem domain using a CFG (obtained with the help of the Conceptual Class Diagram); then a partial modular AG is written to describe the computation of the attribute that denotes each goal of the problem (those AGs are derived taking into Account the Operational Diagrams).

The specification obtained for these case studies will be used to build a rapid prototype to enable its pragmatic validation, as it will be shown in section 6. In subsections 6.2, the OOAG written for each problem will be automatically transformed in a processor—using an AG-based compiler generator—that reads a *scenario* (a sentence with data values that describe a possible system input) and computes the respective solution, the values of the desired goals (the system output). The *language processor generator* used to perform that transformation—the LISA system—will be previously introduced in subsec. 6.1.

The tutorial ends with a synthesis of the document, and hints for future work in section 7.

## 2 Related Work

The grammatical approach to problem solving (software development) can be seen as an extension of object-oriented design methods [RBP<sup>+</sup>91] [Fow97] [GHJV95] where a problem domain model is developed from use-cases and class diagram. However, their main goal is to develop good software models. Our goal is to develop rapid prototypes and early validation of user's requirements.

Our work is closely related to the Grammar-Oriented Object Design (GOOD) [Ars01] [LA02], where all valid object interaction sequences of the cluster of

objects are identified. Then a meta-model is constructed and represented as a context-free grammar. Therefore, a context-free grammar represents the set of all possible interactions (collaborations) of objects in a particular cluster in order to fulfill the domain goals. When a grammar is interpreted at a run-time a cluster will dynamically bind the collaborators to the collaborations. Hence, GOOD facilitates the creation of dynamically configurable components, which encapsulates volatile business rules. The rationale behind this is that creating and representing a model of solutions is more extensible, simpler and more scalable than just creating the single solution. Possible solutions are modelled with a meta-model and represented as a context-free grammar. If this grammar is available to the "users" at run-time, then they are able to customize the system behavior. Since the interaction of objects is obtained from use-case diagrams that describe the functionality of a system, the author called such a grammar a use-case grammar. In other words, use-cases are described with a domain-specific language. In the domain analysis the key abstractions are identified and classified as interactions among subsystems that may be realized as software components. The author in his work distinguishes two types of meta-models: the static (class diagram) and the dynamic (valid object interaction sequences) meta-model. The latter is described with a context-free grammar. Our approach differs from [Ars01] [LA02] since they are using a context-free grammar to describe behavior of the objects (methods), while in our case the structure of a class (attributes) is described. An example of a production rule in [Ars01] [LA02] using the EBNF is:

```
ShoppingCartOperation ::= {AddItem | DeleteItem |
                          SaveShoppingCart} Checkout
```

Our approach has different goals and advantages. However, it can be seen as complementary to the GOOD approach. Combining both approaches to describe the behavior and the structure with a domain-specific language, is under investigation.

The grammatical approach to software development is also related to the rapid prototyping research (e.g. [BL02] [LBNE00]). In [BL02] Two-Level Grammars (TLG) were proposed as an object-oriented requirement specification language. Successive refinement steps starting with natural language lead to more detailed specifications that can be translated to VDM++, which in turn is translated to Java, yielding a rapid prototype of a system. With this approach it is possible to obtain the rapid prototype of a system from natural language specifications. Their Specification Development En-

vironment (SDE) has natural language parsing capabilities and can classify words into nouns (objects/class) and verbs (operations) and their relationships. This initial analysis of requirement documents provides the basis for further refinement with an attempt to classify the domains (classes) to which functions (operations) belong. In more complex cases a rapid prototype is not completely automatically derived since a sufficient degree of interaction with a user is required to ensure a correct interpretation.

In [LBNE00] Computer-Aided Prototyping Systems (CAPS) were used. Both approaches support rapid prototyping, where user's requirements can be easily tested. Unlike throw away prototypes, the supplemental goal of both approaches is to use prototype design in a construction of real life systems. Like our approach, the CAPS supports the incremental prototyping process. All of system functionalities don't have to be implemented in the beginning. The CAPS system has been used as a research tool in the prototyping of a large war-fighter control system. The CAPS is a software tool which generates source programs directly from high level requirements specifications. The generated code can be either in their own domain-specific language (PSDL) or in the target programming language (Ada). Their approach differs from ours in the presentation of domain concepts. The domain concepts in our approach are shown as a context free grammar. In the CAPS they are represented as data-flow diagrams which can be drawn in a PSDL editor. In the CAPS, the stress is put on writing the structure of the prototype, modules, data streams and control constrains. The difference is also in the aim. The CAPS is a stand alone approach in system development, while our approach can be seen as supplemental approach to OOA/D.

## 3 Basic Concepts

### 3.1 Attribute Grammars

Any compilation can be broken down into two major phases:

- Analysis: to discover the meaning of the source program, i.e. its structure, and the precise contents
- Synthesis: to create a target program related with the source program

The source program is converted into an abstract representation which embodies the essential properties of the source language. This abstract representation may be implemented in many ways, but it is usually conceptualized



as a tree. The structure of the tree (nodes labelled by grammar symbols and productions, hierarchically linked to other nodes) represents the control and data flow aspects of the program; additional information (attributes) is attached to the nodes to describe other important values for the compilation process.

The analysis phase of a language processor can be divided into three tasks: lexical, syntactical and semantical. The lexical analysis scans the source text and identifies terminal symbols; the syntactical analysis matches the input stream of terminal symbols against the grammar to identify the productions used and rebuild the derivation (syntactic) tree; the semantic analysis decorates tree nodes, computing the value of the attributes describing properties of the node, and additionally checks their consistency. Notice that the information collected in the attributes of a node is derived from the environment of that node; this allows to deal with *context sensitive information* based on a *context free grammar*.

Attribute Grammars (AG) have proven to be an useful aid in specifying the construction and decoration processes of the structure tree, because they constitute a formal definition of all syntactic (context-free) and semantics (context-dependent) language properties. Moreover, also decorated tree transformation can also be specified through AG

An attribute grammar is based on a context-free grammar (CFG),

$$G = (N, T, S, P)$$

It associates: a set,  $A(X)$ , of attributes with each symbol  $X$  in the vocabulary ( $V$ ) of  $G$  (terminal ( $T$ ) and non-terminal ( $N$ ) symbols); a set,  $R(p)$ , of evaluation rules with each production  $p \in P$ ; and a set,  $C(p)$ , of contextual conditions with each production  $p \in P$ .

So an attribute grammar is formally defined as the following tuple:

$$AG = (G, A, R, C)$$

where  $A = \bigcup_{X \in (N \cup T)} A(X)$  is the set of all the attributes;  $R = \bigcup_{p \in P} R(p)$  is the set of evaluation rules for all the productions; and  $C = \bigcup_{p \in P} C(p)$  is the set of contextual conditions for all the productions.

Each attribute has a type, and represents a specific property of symbol  $X$ ; we write  $X.a$  to indicate that attribute  $a$  is an element of  $A(X)$ . For each  $X \in (N \cup T)$ , the set of attributes of  $X$  is splitted into two disjointed sets:  $A(X) = Inh(X) \cup Syn(X)$ , respectively the *inherited* and the *synthesized* attributes.

Each  $R(p)$  is a set of formulas

$$X.a = func(\dots, Y.b, \dots)$$

that define how to compute, in the precise context of production  $p$ , the value of each attribute  $a$  as a function of the value of other attributes  $b$ , where each defined attribute  $a$  should be a synthesized attribute associated with the nonterminal in the lefthand side or an inherited attribute associated with a nonterminal in the righthand side

$$a \in (Syn(X_0) \cup Inh(X_i)), i \geq 1$$

and each used attribute  $b$  should be an inherited attribute associated with the nonterminal in the lefthand side or a synthesized attribute associated with a symbol in the righthand side

$$b \in (Inh(X_0) \cup Syn(X_i)), i \geq 1$$

Each  $C(p)$  is a set of predicates

$$pred(\dots, X.a, \dots)$$

describing the requirements that must be satisfied in the precise context of production  $p$ . Each predicate, checked for the actual value of the argument attributes (any synthesized or inherited attribute that occurs in that context can be an argument), must hold a **true** value, so that the production is meaningful (is valid from a semantic point of view).

Although based on a small number of simple concepts, formally well-defined, a real attribute grammar (to specify an actual programming language) is a long text with many similar equations, hard to develop and maintain. This fact, the size of a monolithic specification, led the scientific community to search for a modular approach aiming at the down-sizing of the specifications, and the reuse of AG components; in another words, leading to a effective improvement of the development cycle.

There are mainly two possible approaches: attribute-oriented—one module to define one attribute, or a set of related attributes (the full CFG will be repeated in all modules); or symbol-oriented—one module for each grammar symbol (the CFG is split by the different modules).

Independent of the structural approach (monolithic, or modular), the meta-language chosen to write the AG is an important concern. The notation used to write the AG can be more abstract, or more concrete, depending of the

purpose: if we just are interested in the AG as a specification document, we can use a notation close to the mathematical formalism, as simple, abstract and generic as possible; if we intend to build a language processor based on the attribute grammar, we need a more concrete notation, not so far from the implementation language and strategies (the evaluation algorithm, etc.). To build a semantic-directed language processor, all the analyzers and the translator should be developed according to the attribute grammar. We can systematically derive from the AG the semantic module, as it is possible for the lexical analyzer, and the parser. This enable the construction of tools that generate the semantic-directed language processor automatically from the AG. The main difficulty is concerned with the detection of cycles in the attribute definitions and the efficiency of the evaluation process. However this is a well-studied problem that is out of the scope of the present report<sup>1</sup>.

### 3.2 UML Diagrams

**Use-Cases** are used to describe the requirements of a system. They are used in the requirement analysis phase of a project, and contribute to test plans and user guides. We must start identifying the actors involved. Actors are anything that interacts with the system. Each actor defines a particular role. So, a physical person may be represented by several actors because that person has different roles with regard to the system. Or several physical people can be represented by one actor if they have the same role. The next step is to go through all the actors and identify **Use-Cases** for each one. **Use-Cases** describe the things actors want the system to do.

When we pick one particular path through the use case, that is called a **scenario**. Each scenario represents one instance of the use case. They are written from the actor's point of view and they represent a sequence of events that describe the functionality of the use case.

## 4 A grammatical approach to problem specification

This section is devoted to the explanation of our proposal, this is, our method to develop a formal specification for a given problem using a complementary syntax/semantics approach. To support that strategy we will use an attribute grammar described in an object-oriented style.

---

<sup>1</sup>Because it is not relevant for the problem under discussion.

## 4.1 The Principle

The grammatical approach to problem solving hereafter defended is very simple. We associate a grammar symbol with each problem concept, or object class, identified in the analysis phase<sup>2</sup>. If it is a complex concept (defined in terms of other concepts), it will be a nonterminal symbol; and if it is a primitive concept, it will be denoted by a terminal symbol. The main concept, or the top class, corresponds to the grammar *axiom*, or *start symbol*.

Each goal, or desired result, will be represented by a synthesized attribute associated with the *start symbol*. For each result we will write an AG to specify the evaluation of the corresponding attribute.

The final specification will be the composition, or merge, of those partial AGs. The development of each partial AG and their composition to obtain the global AG constitutes the third (and last) step of the proposed approach.

The modularity implicit in any grammar (based on the locality associated with symbols and productions), suggests that we can think about attribute grammars (at the specification and implementation levels) in an object-oriented (OO) manner. AG development, and even the implementation of AG-based language processors, can be clearly improved if we take profit of the advantages of the OO-approach to problem solving, such as:

- natural modelling (due to the proximity with real life situations);
- easy evolvability (addition of new properties or new behavior) due to the extension by inheritance;
- and easy reusability by instantiation of abstract or concrete classes.

Due to these arguments, it is a good choice to use object-orientation in the design, specification and implementation of AG. So our grammatical approach to problem specification will be supported by an OOAG.

Syntax and semantics of each symbol will be specified in a module, similar to a class in an OO-approach. The module name is the symbol name. The first part<sup>3</sup> is the declaration of its attributes, divided in two subsets, the inherited (context dependent) and the synthesized (computed locally). For each attribute declared, we identify its type.

---

<sup>2</sup>Notice that here we suggest an OO-approach to problem analysis, however our principle can be also used with other analysis methods, like the traditional Entity-Relationship and Data-flow diagrams.

<sup>3</sup>This part is similar to the static description of a class in an OO-approach.

A symbol can be constructed in different ways, that is, it can have different syntactic forms, as stated in the CFG that comes out of the second step. So the grammar rules, or productions, for the symbol under specification are declared, each one identified by a unique name.

The functions to be used to evaluate each attribute are then defined, in the context of each production. Also the contextual conditions, if any, that express the data constrains<sup>4</sup> are defined in the context of each production.

To achieve a good understanding of the user's world we need to understand the application domain. In other words, we need to identify concepts and their relationships in the problem domain. For this purpose object-oriented design (OOD) employs use-case diagrams (UCD) and conceptual class diagrams (CCD) [Fow97] which we will take as a starting point for our approach. The UCD [Coc01][Ado02] describes the functionality of the system and its interaction with an environment. The UCD form foundations for further modelling of developing system. They are also helpful for generating system test cases. While UCD are narrative descriptions of specific tasks, the conceptual class diagram captures concepts and relationships between them. Guidelines for developing the conceptual class diagram can be found in [RBP<sup>+</sup>91]. To develop the conceptual class diagram one can apply iteratively the following steps:

- identification of potential classes (look for nouns in the description of the problem),
- elimination of unnecessary (eg. redundant, irrelevant) classes,
- identification of potential associations (any dependency between two classes is an association),
- elimination of unnecessary associations,
- identification of attributes (attributes are properties of individual objects),
- elimination of unnecessary attributes,
- refining with inheritance.

From the use-case diagram and from the conceptual class diagram a skeleton design model is obtained which should be robust with respect to changes of the user's requirements. To identify concepts and their relationships in

---

<sup>4</sup>These constrains must be checked to guarantee the semantic consistence.

the problem domain our grammatical approach is not limited to object-oriented design. Also other approaches, such as entity-relation diagrams and data-flow diagrams, which show the flow of work and the relationship between activities and deliverables, can be applied. However, OOD [Boo94] [GHJV95] is now almost the-facto standard for software system design, and on account of that, it was also our choice. Therefore, our approach (described in Figure 1) is based on the following steps:

- describe the syntax of the problem (the structure of the classes that characterize problem domain), deriving the context-free grammar from the conceptual class diagram,
- describe the semantics of the problem (the meaning of the classes in problem domain), associating attributes to every concept derived from the use-cases and operational diagrams,
- generate a rapid prototype of the system, using a Compiler Generator and the attribute grammar obtained in the two previous steps.

The detailed description of the above steps will follow in the next subsections.

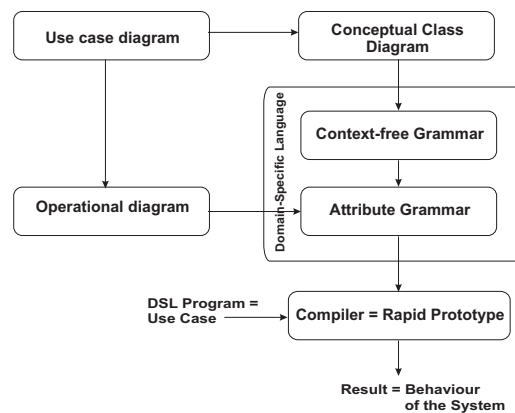


Figure 1: High-level view of the grammatical approach

## 4.2 Deriving a context-free grammar from a conceptual class diagram

The role of non-terminals in a context-free grammar is two fold. First, at higher abstraction level non-terminals are used to describe different concepts in the programming language (e.g. an expression or a declaration in a general-purpose programming language). On the other hand, at a more concrete level, non-terminals and terminals are used to describe the structure of a concept (e.g. an expression consists on two operands separated by an operator symbol, or a variable declaration consists of a variable type and a variable name). Therefore both the concepts and relationships between them, belonging to the specific problem domain, are captured in a context-free grammar. But, this is also true for the conceptual class diagram which describes concepts in a problem domain and their relationships. It is clear that both formalisms can be used for the same purpose and that some rough transformation from a conceptual class diagram to a context-free grammar and vice versa should exist. The transformation from a conceptual class diagram to a context-free grammar is depicted in Tables 1 and 2. In general, classes are mapped to non-terminal symbols and instance variables are mapped to terminal symbols.

Transformation table shows how to derive a context-free grammar from a conceptual class diagram. A class and a non-terminal are basic concepts in a conceptual class diagram and in a context-free grammar. The mapping here is self-evident. A conceptual class diagram contains instance variables, which define the state of a class instance. Instance variables are represented in a context-free grammar as terminal symbols. In general, a class diagram consists also of operations, which will be identified when the semantics of context-free grammar is going to be defined. Associations represent the interaction between classes and have to be included in a context-free grammar. The navigability association can be shown with the production  $A \rightarrow B$ , where the non-terminal  $A$  gets information about attributes of the non-terminal  $B$ . Association has multiplicity. Describing multiplicity with grammar productions is straightforward as shown in Tables 1 and 2. For generalization we propose the production  $A \rightarrow B \mid C$ . The non-terminal  $A$  can be implemented either with the non-terminal  $B$  or non-terminal  $C$ . The composition and aggregation are shown as the navigability association. In the composition the non-terminal  $B$  can appear in other productions. On the other hand, in the aggregation the non-terminal  $B$  is reachable only from the non-terminal  $A$ .

In CCD classes can collaborate with more than just one class. For example,




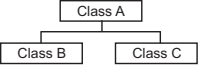
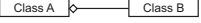

Association	Class diagram element	Grammar
Class		Class (non-terminal) attribute (terminal)
Association		$A \rightarrow B$
Navigability		$A \rightarrow B$
Generalization		$A \rightarrow B \mid C$
Aggregation		$A \rightarrow B$ $(\neg \exists X \in N, X \Rightarrow B)$ $\wedge X \neq A$
Composition		$A \rightarrow B$

Table 1: From a conceptual class diagram to a context-free grammar

a class A associates with classes B, C and D. In our approach, this collaboration is described with CFG production  $A \rightarrow B C D$ . The sequence of non-terminals on right side of the production should be in natural order and depends on collaboration of entities in a given problem domain.

### 4.3 Deriving an AG from an operational diagram

To describe the semantics or the meaning of a concept an attribute grammar is used. Attribute grammars [Knu68] [ME00] are natural extensions of context-free grammars and as such very well support our approach which is based on context-free grammars. The syntax and semantics of each symbol is specified in a module; modularity is, on one hand, inherent to the class concept in OOD, and, on the other hand, it is implicit to grammars (based on the locality associated with symbols and productions). The first part of a module is the declaration of its attributes, divided in two subsets, the inherited (context dependent) and the synthesized (computed locally).







Cardinality	Class diagram element	Grammar
Multiplicity exactly one		$A \rightarrow B$
Optional multiplicity		$A \rightarrow B \mid \epsilon$
Multiplicity [0..m]		$A \rightarrow \text{MoreB}$ $\text{MoreB} \rightarrow \text{MoreB } B \mid \epsilon$
Multiplicity many		$A \rightarrow \text{MoreB}$ $\text{MoreB} \rightarrow \text{MoreB } B \mid B$

Table 2: Association multiplicity

The functions to be used to evaluate each attribute are then defined, in the context of each production. Also the contextual conditions, if any, that express the data constraints are defined in the context of each production. This step is intellectually most demanding; therefore some additional supporting techniques based on the use-cases (diagrams and scenarios) should be used; namely we suggest the use of the operational diagram that is inferred from the referred scenarios.

The result of this step is a complete attribute grammar specification for a given problem.

#### 4.4 Generating the rapid prototype of a system

To generate the rapid prototype of a system our compiler-generator LISA [MLAŽ02] has been used. The LISA system automatically generates a compiler or an interpreter and other language-based tools—such as language-knowledgeable editor, inspectors, and animators [HPM<sup>+</sup>02]—from an attribute grammar specification. One of LISA’s most important feature is that it supports incremental development of specifications, which is especially important in particular tasks of the software development described in this paper.

## 5 Case Studies: Specification

To illustrate our approach, we will present in this section four case studies. For each case, we introduce informally (in english) the problem to be solved; then we present the conceptual class diagram (CCD) and we design a CFG to introduced the non-terminal and terminal symbols that describe the complex and primitive concepts existing in the problem domain; and, at last, we give the semantic specification for each concept (non-terminal or terminal symbol), using the proposed modular, attribute based, approach.

To specify the OOAG we will use a formalism independent of any concrete generator (AG-processor). The syntax for each module (describing an AG symbol) is defined by the grammar in appendix A. That grammar also specifies the way to build a complete OOAG, and the compositional language to assembly the partial OOAGs into the final (or global) OOAG.

**A Notation for OOAG's** To make next section readable, we show bellow the **template** used for a nonterminal symbol:

NonTerm X0:

```
Inh: { InheritedAttName: AttType }
Syn: { SynthesizedAttName: AttType }
```

ProdX0Name( X0 -> X1 X2 ... Xn ):

```
X0.SynAttName = FunctionName( X0.InhAttName, ...,
                               Xk.SynAttName )
```

.....

```
Xi.InhAttName = FunctionName'( X0.InhAttName, ...,
                               Xj.SynAttName )
```

.....

i,j,k >= 1

CC: ( PredicateName(Xi.InhAttName, ..., Xj.SynAttName) ) &&

.....

```
( PredicateName'(Xk.InhAttName, ..., Xl.SynAttName) )
```

i,j,k,l >= 0

For a terminal, the template is similar: the first keyword is replaced by `Term`, the `Inh` declaration disappears, and `ProdXOName()` is replaced by `RegExp()`.

## 5.1 CS1: Vending Machine

**The problem:** We want a program for the daily management of a Vending Machine (VM). Given the stock (name, price and quantity of each choco available) and the data for each sale (name of chosen choco, and amount of money introduced), the goal is to compute the income (sum of money introduced), and the final stock. No choco should be provided if:

- the name does not exist in the stock list or
- the quantity is 0 or
- the amount of money is different from the price.

**The problem specification:** After the analysis of the problem stated above, the discovering of the main functionalities is to be done and present them as use case diagram.

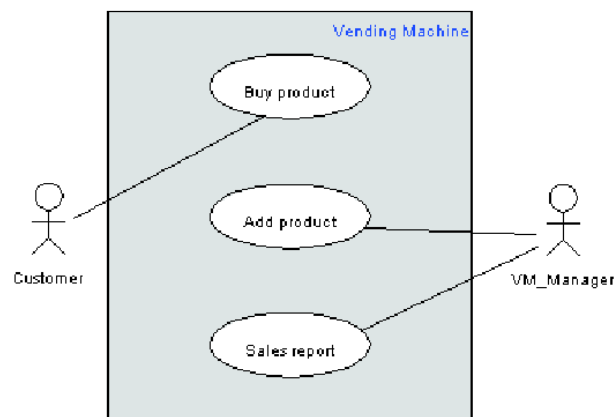


Figure 2: Use case diagram for case study of Vending Machine

For the case study of the Vending Machine we identify three main services represented with use cases *Buy Product*, *Add Product* and *SalesReport* (Fig. 2). To specify their functionalities, the sequence of actions has to be defined. Therefore, scenarios for use cases are written (description follows below).

Scenario for Buy product use case:

1. Request the name of the product.
2. Request for customer's money.
3. Check for correct amount of inserted Money.
4. Update the list of products in product database. Use case end.

ALT 2a: Quantity of product is zero.

Go to step 1.

ALT 3a: Not enough money. Go to step 2.

Scenario for Add product use case:

1. Request for a product name.
2. Request for a product quantity.
3. Request for a product price.
4. Update the list of products in product database. Use case end.

ALT 1a: Product name already contained in the product database. Inserting skipped.

Use case end.

Scenario for Sales report use case:

1. Request the state of product storehouse.
2. Calculate the income of vending machine services.
3. Print the services income and state of product database. Use case end.

### **The Class Diagram:**

After a detailed analysis (with the use case diagram and scenarios) of the problem stated above, we identified vending machine (VM) as the main concept.

Two main concepts that interfere in the management of a vending machine are: *Stocks*, and *Sales*. *Stocks* is a set of *Stock*, and each stock (a product) has a name (*ProdName*) with attribute *name*, a price (*ProdPrice*) with attribute *price*, and a quantity (*ProdQty*) with attribute *quantity*.

In a similar way, the *Sales* class is a non-empty set of sale operations; the data to be kept for each singular *sale* operation is the product name (*ProdName*) and the amount of Money given.

The structure of the problem domain can be defined in terms of classes and relationships as depicted in the conceptual class diagram of figure 3.

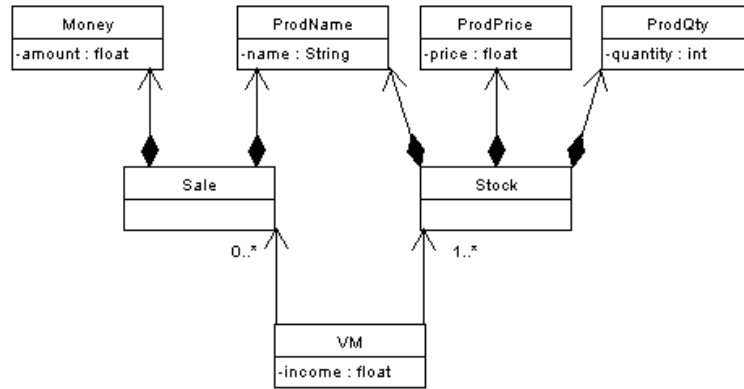


Figure 3: Conceptual class diagram for Vending Machine

### The Structure:

Remembering that, in our approach, a problem concept is denoted by a grammar symbol, the CFG below formalizes the problem syntax in the sense that it specifies the structure of the problem domain, relating the concepts (the object classes) among them. To write the CFG we just follow the conceptual class diagram drawn above and rules from transformation tables 1 and 2.

```

VM      -> Stocks Sales
Stocks  -> Stocks Stock
        | Stock
Stock   -> ProdName ProdPrice ProdQty
ProdName -> name
ProdPrice -> price
ProdQty  -> quantity
Sales    -> Sales Sale
        | &
Sale     -> ProdName Money
Money    -> amount
  
```

**The Semantics:** After the generic overview of the classes and the relationships given by the CCD and the CFG<sup>5</sup>, we describe their semantics, writing an attribute grammar module for each non-terminal symbol.

In the 1. phase of defining semantic information, the operational diagram and attribute mapping tables are defined to support writing specifications of attribute grammar (2. semantic phase).

**The semantics–1.st phase:** Capturing semantics of the domain is the most demanding part of the approach, therefore an auxiliary (supporting) diagram is proposed.

The semantic constructs in attribute grammar are determined in subsection 4.3. The starting point for finding them the use case diagram is used. Use case diagram is further described with scenarios, which define the interaction between an actor and evolving system. Parsing the scenarios can bring most of the semantic information needed for writing attribute grammar. To support the derivation of semantic information from scenarios, the operational diagram (Fig. 4 shows operational diagrams for case study of Vending Machine) has been used.

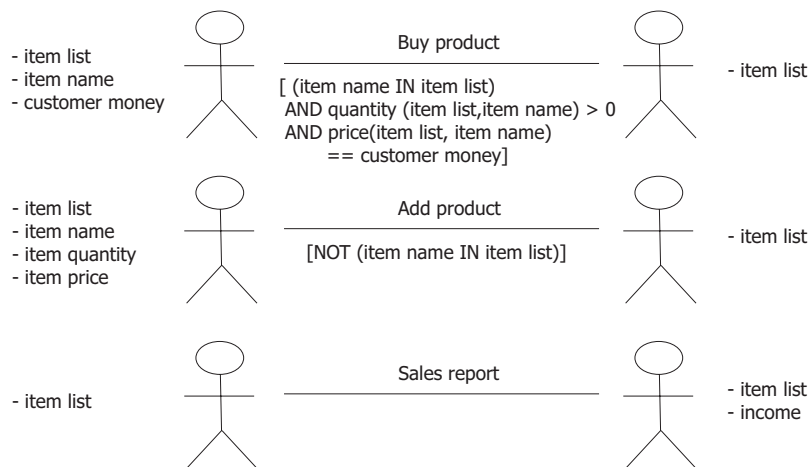


Figure 4: The operational diagrams for Vending Machine

Each collaboration of an actor and use case diagram is introduced with operational diagram. In the diagram actor shows up twice. First appearance

<sup>5</sup>That defines each complex class in terms of the other classes or atomic values.

on the left represents an actor before using the system and on the right represents an actor after collaboration with the system. In the middle, the name of influenced use case is noted.

Both actors are supported with semantical information, which we get with parsing scenarios of involved use case. Left actor possesses information that the actor needs to collaborate with the system. On the right we write information that actor synthesizes in collaboration with the use case.

Information in operational diagrams represent semantics of the system and will be further represented as inherited and synthesized attributes in attribute grammar. Still, the open question is to which non-terminals attributes are associated.

The operational diagram brought some important information about attributes and contextual conditions. The next task is to associate attributes from operational diagram to non-terminals in context-free grammar. The table 3 shows the partial attribute mapping to non-terminals.

Operational diagram	Non-terminal	Side	Terminal	I(x)	S(x)
item list	Stock	left,right	no	StkTab	FStkTab
item name	ProdName	left	yes		name
item quantity	ProdQty	left	yes		quantity
item price	ProdPrice	left	yes		price
customer money	Money	left	yes		amount
income	Sale	right	no		sum

Table 3: Attributes mapping to nonterminals

In the first column the attribute names that appeared in operational diagram are written. The next column represent the name of the non-terminal to which attribute should be associated. The column *Side* and column *Terminal* are crucial to determinate, whether attribute should be inherited or synthesized. The *Side* column represents the side where attribute in operational diagram appears. If attribute appears on both sides, attribute should be inherited, as well as synthesized. If it appears on left side of operational diagram and is represented as terminal in context-free grammar, the attribute should be defined as synthesized. If attributes appears on the left side and no terminal can be found in context-free grammar, the attribute should be inherited. The last case is, when an attribute appears only on the right side of the operational diagram. This attribute is synthesized.

So far, operational diagram defines attributes important for performing de-

defined functionalities of the system. The right side attributes give the information of returned data after actor's collaboration with the system. Therefore, the right side attributes should be mapped to starting non-terminal. The starting non-terminals are shown in table 4.

Attribute	Starting non-terminal
FStkTab	VM
income	VM

Table 4: Attributes in starting non-terminal VM

The table 5 shows attribute carrying between non-terminals in attribute grammar. In the table attributes that must be carried to other non-terminals are showed. To construct this mapping table the domain must be understood well. Each attribute, synthesized or inherited must be considered separately. The main point is to define where each attribute should be carried and with what purpose. One way to achieve that is to write syntax tree or write semantic modules for each non-terminal symbol. Detailed description is written in next subsection with attribute grammar specifications.

Attribute	Other nonterminals
FStkTab	Stocks, Sales, Sale
StkTab	Stocks, Sales, Sale
sum	Sales

Table 5: Attributes in other nonterminals

The alternatives in use case scenarios are basics to find the contextual conditions. The contextual conditions are inserted between square brackets (see Fig. 4) where basic boolean operator can be used. Contextual conditions noted on operational diagrams must be also associated to productions of attribute grammar. Their appearance in productions is closely connected to the attributes which define the contextual conditions. Contextual conditions are further implemented with functions which evaluates attributes. The identification of functions are further described in the next semantical phase.



**The Semantics–2.nd phase:** After detailed semantic description of the problem domain, we can write specifications in attribute grammar. The only semantic part left, is to define functions for attribute evaluation. The specifications are broken into separate non-terminal descriptions.

According to the semantic description (4), *vending machine* should have two attributes: **FStkTab** (the final stock table); and **income** (the final income). Notice that the values of stock table is not computed locally to the VM symbol definition; it depends on the sales processing (Table 5). To perform such computation, it is necessary to know the stock table before start selling. This value is defined during the processing of stock item descriptions.

From these statements it is clear that symbols **Stock** and **Sales** will be characterized by a stock table that holds two distinct values along the processing time: the initial one (depends on the environment); and the final one (computed during the sales processing). So both will be associated with two attributes—**StkTab**, **FStkTab**—the first one inherited from the context, and the second one synthesized from the previous and the attributes associated to each item. The type, **TAB**, of those two attributes is a finite function (a mapping) that associates a name with a pair (price, quantity).

This second attribute **income** is used to describe the computation of the vending machine service income. The final result depends on the amount accumulated on each sale, as stated in the following module for *vending machine*.

```
TAB = FF( string, ( string, real, int ) )
```

```
NonTerm VM:
```

```
  Inh: {}
```

```
  Syn: { FStkTab: TAB, income:real }
```

```
  vmManager(VM -> Stocks Sales):
```

```
    VM.FStkTab = Sales.FStkTab
```

```
    VM.income = Sales.sum
```

```
    Stocks.StkTab = {}
```

```
    Sales.StkTab = Stock.FStkTab
```

The **Stocks** is defined in terms of an stock list (non-empty list), and its two attributes —**StkTab**, **FStkTab** (Table 5) — are evaluated in a obvious way:

otherwise, we just take as final value the table obtained after storing in the initial one the data concerned with the new item added.

NonTerm **Stocks**:

Inh: { StkTab: TAB }

Syn: { FStkTab: TAB }

oneMoreStock(**Stocks** -> **Stocks** **Stock**):

**Stocks**/0.FStkTab = **Stock**.FStkTab

**Stocks**/1.StkTab = **Stocks**/0.StkTab

**Stock**.StkTab = **Stock**/1.FStkTab

firststock(**Stocks** -> **Stock**):

**Stocks**.FStkTab=**Stock**.FStkTab

**Stock**.StkTab=**Stocks**.StkTab

Notice that in the first production of module **Stocks** it was necessary to distinguish between the initial **Stocks** (the first symbol on the RHS of the production) and the final list of stock items, **Stocks** (the symbol on the LHS of the production). To do that, we indexed the occurrences of that ambiguous symbol with a position number (using 0 for the LHS, and starting in 1 for the RHS).

The semantic description of the stock table attribute synthesized by **Sales** follows the pattern above, as this symbol is also a list (now it may be empty list, but that makes no changes).

NonTerm **Sales**:

Inh: { StkTab: TAB }

Syn: { FStkTab: TAB, sum: real }

oneMoreSale(**Sales** -> **Sales** **Sale**):

**Sales**/0.FStkTab = **Sale**.FStkTab

**Sales**/0.sum = **Sales**/1.sum + **Sale**.sum

**Sales**/1.StkTab = **Sales**/0.StkTab

```

    Sale.StkTab = Sales/1.FStkTab

emptySale(Sales -> &):
    Sales.FStkTab = Sales.StkTab
    Sales.sum = 0.0

```

According to the definition and analysis presented in the beginning, the final income is the sum of the money introduced by the customers in each sale. If the sales list is an empty list, the sum is set to zero; otherwise, we add that amount to the sum carried by the previous list, as specified in the previous module.

Those two symbols that represent collections, `Stock`, `Sales`, were used to pass information over the tree; the new values are in fact constructed at the level of the list items. Therefore, collections should be present in the non-terminal `Stock` (Table 3). So we shall say now how the data contained in a stock item (`ProdName`), price (`ProdPrice`) and quantity (`ProdQty`)—is included in the initial stock table to produce the final stock table. This is specified in the `Stock` module bellow.

```
insert: TAB * string * real * int --> TAB
```

NonTerm Stock:

```

Inh: { StkTab: TAB }
Syn: { FStkTab: TAB }

```

```

mkStock(Stock -> ProdName ProdPrice ProdQty):
    Stock.FStkTab = insert(Stock.StkTab,
                           ProdName.name,
                           ProdPrice.price,
                           ProdQty.quantity)

```

```
CC: ( NOT(ProdName.name IN Stock.StkTab) )
```

The function `insert()` always adds a new triple to the initial stock table, even if the key element (the name) already exists. So it is necessary to include a *contextual condition (CC)* to state that the construction of the final stock table is valid (i.e. *makes sense*) if and only if the product name does not exist in the initial stock table. By other words, *re-declarations are not allowed*.

The update of the stock table after each sale is defined by the module `Sale` bellow. Once again that specification follows a pattern very similar to the previous one; now instead of the `insert()` function, we use the `update()` function that decreases the stock quantity for the product whose name is given. This function do not take care of exceptions: if that product name exists in the stock table, it subtract 1 to its quantity.

```
update: TAB * string * real --> TAB
```

```
NonTerm Sale:
```

```
Inh: { StkTab: TAB }
```

```
Syn: { FStkTab: TAB, sum: real }
```

```
mkSale(Sale -> ProdName Money):
```

```
    Sale.FStkTab = update(Sale.StkTab, ProdName.name,
                          Money.amount)
```

```
    Sale.sum = Money.amount
```

```
CC: ( ProdName.name IN Sale.StkTab ) &&
```

```
    ( qty(Sale.StkTab, ProdName.name) > 0 &&
```

```
    ( Price(Sale.StkTab, ProdName.name) ==
```

```
      Money.amount ) )
```

Three contextual conditions have been included on `Sale` specification: we must verify if the chosen product name belongs to the machine product list (the domain of stock table), the quantity in stock is greater than 0 and money inserted equivalent to product price.

The semantics for product name, price and quantity is specified using the values returned by the scanner (represented by the intrinsic attribute named `lexval` and associated to each terminal symbol).

```
NonTerm ProdName:
```

```
Inh: {}
```

```
Syn: { name: string }
```

```
getName(ProdName -> name):
```

```
    ProdName.name = name.lexval
```

```
NonTerm ProdPrice:
```

```
Inh: {}
```

```

Syn: { price: real }

getPrice(ProdPrice -> price):
    ProdPrice.price = atof( price.lexval )

```

```

NonTerm ProdQty:
    Inh: {}
    Syn: { quantity: int }

getQuant(ProdQty -> int):
    ProdQty.quantity = atoi( int.lexval )

```

To finish the specification, we need to define the semantics for **Money**. Its value is computed from the value returned by the scanner (as done above for product name, price and quantity).

```

NonTerm Money:
    Inh: {}
    Syn: { amount: real }

getMoney(Money -> int):
    Money.amount = atoi( int.lexval )

```

The union of all the modules presented, joining productions, evaluation rules and contextual conditions, produces an AG that is the formal specification of the problem stated. From that description we can generate a prototype to check every decision, as it will be shown in the section 6.2.1.

## 5.2 CS2: Automatic Teller Machine

**The problem** The next case study is taken from Two-Level Grammar (TLG) approach [BL02]. The *Grammatical approach to problem solving* is also related to this work, where TLG were proposed as an object-oriented requirements specification.

We want a program for Automatic Teller Machine (ATM). Each ATM belongs to a bank. In bank we keep the list of accounts. The account has three integer data fields: id, pin and balance.

The ATM machine provides a withdraw service. First, service verifies id and pin than withdraws an amount of money in the following sequence: first it

gets the balance of the account of id and pin from the bank, if the amount is less or equal to the balance, updates the balance of the account and delivers the amount. The withdraw service shouldn't be provided if:

- id doesn't exist,
- pin is not valid for the account with specific id,
- withdraw amount is bigger than balance.

**The problem specification:** After the analysis of the problem stated above, the discovering of the main functionalities is to be done and present them as use case diagram.

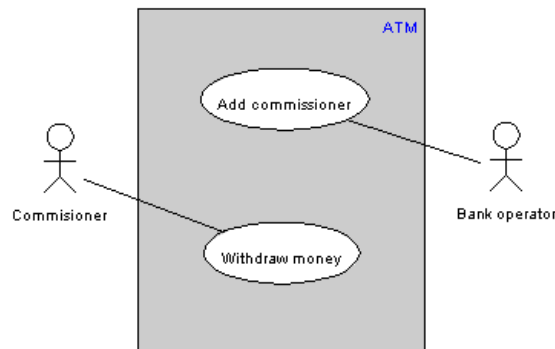


Figure 5: Use case diagram for the case study of Automatic Teller Machine

For the case study of the Automatic Teller Machine (Fig. 5) we identify two main services represented with use cases *Add commissioner* and *Withdraw Money*. Description of the scenarios follows:

Scenario for Add commissioner use case:

1. Generate the id number of the commissioner.
2. Request commissioner to select the pin for new bank account.
3. Request commissioner to deposit money.
4. Update the list of commissioners with new entry. Use case end.

ALT 1a: Id number already contained in the commissioner database. Go to step 1.

Use case end.

Scenario for Withdraw Money use case:

1. Request for account id.
2. Request for commissioner's pin.
3. Request for withdraw amount.
4. Update the account in commissioners database. Use case end.

ALT 1a: Account id doesn't exist. Use case end.

ALT 2a: Account pin not valid. Use case end.

ALT 3a: Withdraw amount greater than withdraw balance. Use case end.

**The Conceptual Class Diagram:** The basic step is creating a conceptual class diagram. For that purpose, we define basic entities. First entity which appear in our case study is **Bank**. The **Bank** keeps **Accounts** for customers. Next step is to implement Withdraw service. With service defining, new entity appears, entity **ATM**. The **ATM** interacts with both entities, **Bank** and **Account**. Conceptual class diagram is shown on figure 6.

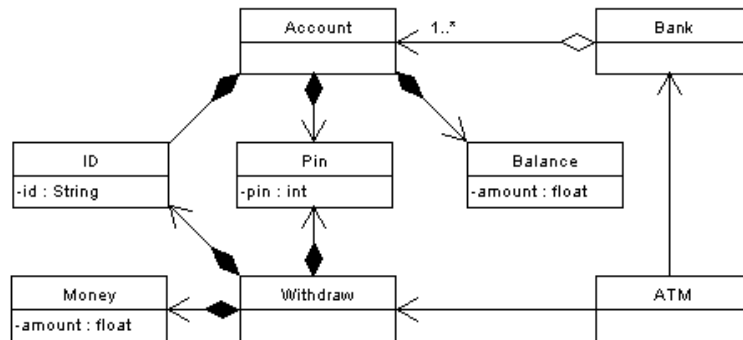


Figure 6: Conceptual Class Diagram for ATM

**The Structure:** The context-free grammar bellow formalizes the problem syntax defined above. In our approach, the entities are presented as grammar symbols, the relationships between main concepts are described with

transformation table in section 4.

ATM	-> Bank Withdraw
Bank	-> Accounts
Accounts	-> Accounts Account   Account
Account	-> Id Pin Balance
Withdraw	-> Withdraw Id Pin Money
Id	-> id
Pin	-> pin
Balance	-> balance
Money	-> amount

**Semantics:** After defining the structure, the stress is in deriving semantic specifications of our problem domain. In our approach, structure was represented with context-free grammar. For evaluation of CFG an attribute grammar for each non-terminal should be written. Foundations for attribute grammar can be found in the conceptual class diagram (with associations and class attributes), use-case diagram and operational diagrams (description follows in the next subsection).

According to the problem description, operational diagrams and belonging attribute mapping tables are shown in the first phase and semantic description in the second phase.

**The semantics-1.st phase:** Closer look at the use case diagram helps defining semantic constructs for problem domain. From use case diagram, two operational diagrams are created, for use cases *Add commissioner* and *Withdraw money* (Fig. 7). Parsing scenarios brings information of operational diagram attributes. Collaboration between system and bank operator carries attributes possessed and synthesized in the use case of *Add Customer*. To perform that service, the actor needs to possess following operational diagram attributes: commissioner list and data of new commissioner (id, pin and balance). The role of first attribute is two fold - for the execution of operation and as a result of collaboration. Therefore, attribute is presented on both sides of operational diagram. Other operational diagrams follow the same pattern. Reading alternatives of scenarios is the main source to specify the operational diagram conditions as depicted on figure 7.

To describe semantics of the problem domain, the operational diagram are crucial for defining basic properties of AG attributes (mapping non-



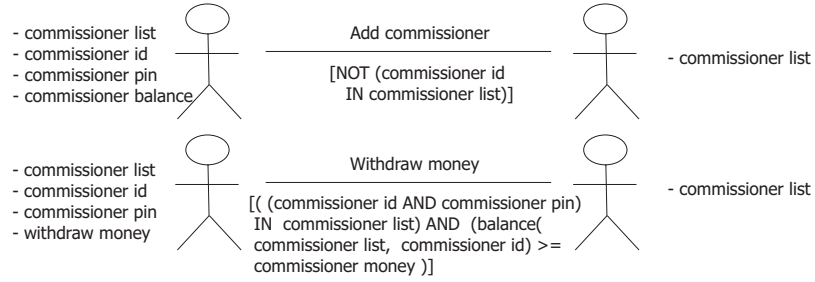


Figure 7: The operational diagram for Automatic Teller Machine

terminals, name, inherited or synthesized value), depicted in table 6. The first attribute from operation *Add commissioner* is *commissioner list*. To start with, the source non-terminal of the attribute has to be defined (**Account**). Next, name of the attribute in AG has to be chosen properly (**AS**). Finally, the type of attribute in attribute grammar has to be defined. Attribute *commissioner list* exists on both sides of operational diagram, therefore attribute is inherited (infix "in") and synthesized (infix "out") as well.

Operational diagram	Non-terminal	Side	Terminal	I(x)	S(x)
commissioner list	Account	left,right	no	inAS	outAS
commissioner id	Id	left	yes		id
commissioner pin	Pin	left	yes		pin
commissioner balance	Balance	left	yes		balance
withdraw money	Money	left	yes		amount

Table 6: Attributes mapping to non-terminals

As described at the first case study (Subsection 5.1), the right side attributes of operational diagrams define basic system information of system operations. Therefore, attribute **outAS** is mapped to starting non-terminal as shown in table 7.

Attribute	Starting non-terminal
outAS	ATM

Table 7: Attributes in starting non-terminal VM

To support evaluation of all system services, some attributes are propagated through non-terminals (Table 8). In the case of operational diagram *Withdraw money*, the attribute `inAS` needs to be propagated from the non-terminal `Account` to non-terminal `Withdraw` (non-terminals `Accounts`, `Withdraw`, but not `ATM`—already present in table 7).

Attribute	Other non-terminals
<code>outAS</code>	<code>Accounts</code> , <code>Bank</code> , <code>Withdraw</code>
<code>inAS</code>	<code>Accounts</code> , <code>Withdraw</code>

Table 8: Attributes in other nonterminals

**The semantics-2.nd phase:** As we can see, the main concept is the non-terminal `ATM`. Before performing withdraw service, `ATM` gets the list of all bank accounts. The second task of non-terminal `ATM` is to pass account list to non-terminal `Withdraw` to perform withdraw task. The final information of the accounts collection will be stored in attribute `outAS` (see Table 7). Attribute is of type `TAB`, which is finite function (a mapping) that associates a name with an account data (string, int, double).

`TAB = FF(string, (string, int, double))`

NonTerm `ATM`:

`Inh: {}`

`Syn: {outAs: TAB}`

`atmWithdraw(ATM -> Bank Withdraw):`

`ATM.outAS = Withdraw.outAS`

`Withdraw.inAS = Bank.outAS`

For each commissioner bank (non-terminal `Bank`) keeps the information about his account. For collection of all accounts information we have two attributes, `inAS` and `outAS`. From the names it is clear, that the attributes will keep distinct values along processing time: `inAS` will be used for initial value of environment (inherited attribute), an attribute `outAS` will be computed during processing and will contain the final information of all commissioners (synthesized attribute).

NonTerm Bank:

Inh: {}

Syn: {outAs: TAB}

bankManager(Bank -> Accounts):

Bank.outAS = Accounts.outAS

Accounts.inAS = {}

The semantic of non-terminal `Accounts` follows the principle of non-terminal `Bank`.

NonTerm Accounts:

Inh: {inAS: TAB}

Syn: {outAS: TAB}

oneMoreAccount(Accounts -> Accounts Account):

Accounts/0.outAS = Account.outAS

Accounts/1.inAS = Accounts/0.inAS

Account.inAS = Accounts/1.out

oneAccounts(Accounts -> Account):

Accounts.outAS = Account.outAS

Account.inAS = Accounts.inAS

Non-terminals `Bank` and `Accounts` are used to pass information that are read at the level of a leaf `Account`. Hence, the construction is at the same level. Semantic constructs of non-terminal `Account` is shown below. For each commissioner's account we get information from non-terminals `Id`, `Pin` and `Balance`.

insert: TAB \* (string \* int \* real) --> TAB

NonTerm Account:

Inh: {inAS: TAB}

```
Syn: {outAS: TAB}
```

```
getCommissioner (Account -> Id Pin Balance)  
Account/O.outAS = insert(Account.inAS,  
    new Account(Id.id, Pin.pin, Balance.balance))
```

```
CC: (NOT(Account.id IN Accounts/1.inAS))
```

The function `insert()` adds an element of triple: id, pin and balance to account table. If the element (with same id) is already contained in the list table, element is not added. This is the reason contextual conditional (CC) is added. With the condition we ensure, that the construction of the final account list is valid.

In the module `Withdraw`, the function `get()` gets from the table of accounts the account of commissioner which wants to withdraw the amount of money from account. If the element (with the same id and pin) is contained in the list table, withdraw service can be provided. To accomplish this request, the contextual conditional (CC) is added. Next condition also ensures, that withdraw service is cancelled if the balance is less than withdraw amount. The function `setAccount()`, updates the state of withdrawing account balance and implements conditional defined above. Finally, `update()` function inserts updated commissioner's account back to account list (attribute `outAs`).

The non-terminals `Id`, `Pin` and `Money` holds the `Withdraw` information: commissioners's id, pin and withdraw amount.

```
update: TAB * acc --> TAB  
get: TAB * string * int --> (string, int, real)  
setAccount: (string, int, real) * real --> (string, int, real)
```

NonTerm Withdraw:

```
Inh: {inAS: TAB}  
Syn: {outAS: TAB}
```

```
getCommissioner (Withdraw -> Id Pin Money)  
ATM.outAS = update (Withdraw.inAS,  
    setAccount(get(Withdraw.inAS, Id.id, Pin.pin),  
        Money.amount))
```

```
CC: (((Id.id AND Pin.pin) IN Withdraw.inAS) AND
      (Balance(Withdraw.inAS, Id.id) >= Money.amount))
```

The semantics for product `Id`, `Pin`, `Balance` and `Money` is specified using the values returned by the scanner (represented by the intrinsic attribute named `lexval` and associated to each terminal symbol).

NonTerm `Id`:

```
Inh: {}
Syn: { id: string }

getId(Id -> id):
    Id.id = id.lexval
```

NonTerm `Pin`:

```
Inh: {}
Syn: { pin: int }

getPin(Pin -> pin):
    Pin.pin = atoi( pin.lexval )
```

NonTerm `Balance`:

```
Inh: {}
Syn: { balance: real }

getBalance(Balance -> balance):
    Balance.balance = atof( balance.lexval )
```

NonTerm `Money`:

```
Inh: {}
Syn: { amount: real }

getQuant(Money -> amount):
    Money.amount = atof( amount.lexval )
```

### 5.3 CS3: Video Store

**The problem specification:** The Video Store (VS) case study is one of the basic examples of the refactoring [Fow97][vDM02]. The case study

represents a prototype program for customer charges at a video store. The program is told, which movies are rented by customer and for how long each movie is rented. It calculates the charges, which depend on how long the movie is rented and on the type of the movie. There are three kinds of movies: regular, children and new releases.

**The problem specification:** After the analysis of the problem stated above, the discovering of the main functionalities is to be done and present them as use case diagram (Figure 8).

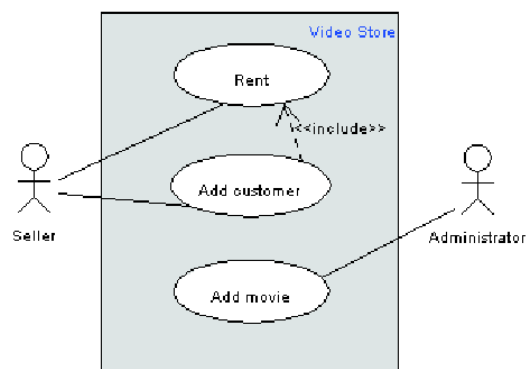


Figure 8: Use case diagram

For the case study of the Video Store we identify three main services represented with use cases *Rent*, *Add Customer* and *Add Movie*. To specify their functionalities, the sequence of actions has to be defined. Therefore, scenarios for use cases are written (description follows below).

Scenario for Rent use case:

1. Request the name of the customer.
2. Request the titles of rented movies.
3. Insert the list of rented movies in customer's database.
4. Calculate the charge for rental service. Use case end.

ALT 1a: Name not in the customer database. Insert new customer. Use Add customer.

ALT 2a: Movie title unknown.

Go to step 2.

Scenario for Add customer use case:

1. Request for a customer name.
2. Insert the customer in customer database. Use case end.

ALT 2a: Customer already contained in the customer database. Inserting skipped. Use case end.

Scenario for Add movie use case:

1. Request for a movie title.
2. Request for a movie type.
3. Insert the movie in movie database. Use case end.

ALT 3a: Movie title already contained in movie database. Inserting skipped. Use case end.

**The Conceptual Class Diagram:** Again, from problem specifications the conceptual class diagram has to be obtained. For that purpose, finding basic entities has to be done. Conceptual class diagram is shown on figure 9.

**The Structure:** After describing conceptual class diagram, we are able to write context free grammar. In our approach we use the transformation tables 1 and 2 which provides CFG from conceptual class diagram shown on figure 9.

```
VideoStore -> Movies Customers
Movies      -> Movies Movie
           | &
Movie       -> title Price
Customers   -> Customers Customer
           | &
Customer    -> name Rentals
Rentals     -> Rentals Rental
           | &
Rental      -> daysRented Movie
Price       -> new | child | reg
```

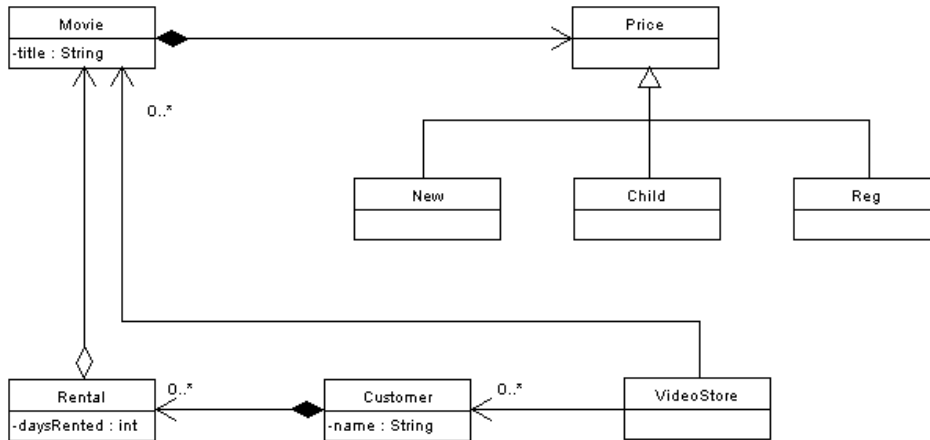


Figure 9: Conceptual class diagram for Video Store

**Semantics:** According to the problem description, the semantics of the case study will be described in two phases. First one, conceptually defines attributes for attribute grammar and the second, showing specifications of attribute grammar.

**The semantics-1.st phase:** Again, the starting point for finding semantic constructs, the use case diagram is used. From use case diagram three distinct operational diagrams are defined - *Add movie*, *Add customer* and *Rent* (Figure 10). Parsing use case scenarios brings valuable attribute information to write operational diagrams. For example, scenario of *Add movie* expects movie title, movie type and movie list before operation is performed. After execution, the movie database is refreshed (step three in scenario), therefore attribute "movie list" occurs in right side of operational diagram. Hence, alternatives in scenario bring the contextual condition as depicted on figure 10.

Information from operational diagram will be used for describing type of attributes in attribute grammar (Table 9). Operational diagram *Add movie* needs attribute *movie list* (Figure 10). To begin with, the attribute source non-terminal has to be defined. For the case of attribute *movie list* the mapping non-terminal is *Movies*. To perform the same operation, the attribute is placed at both sides of operational diagram. Therefore, attribute is inher-



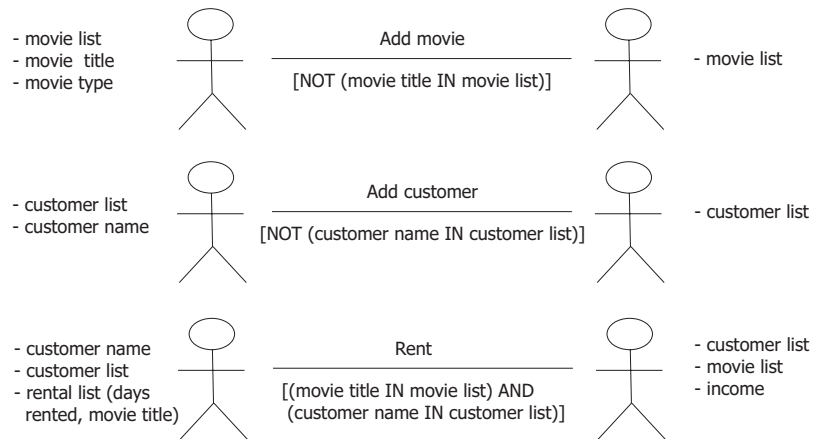


Figure 10: The operational diagram

ited (inMS) as well as synthesized (outMS). Further, attributes from operational diagram *Add movie* are *movie title* and *movie type*. To find mapping non-terminals, conceptual call diagram has to be studied well (non-terminal *Movie* for operational diagram attribute *movie title* and non-terminal *Price* for operational diagram attribute *movie type*). Both attributes have terminal source (attributes in conceptual class diagram). Therefore, attributes are defined as synthesized. Detailed explanation of mapping operational diagram attributes to non-terminals can be found in previous case studies. The right side attributes from operational diagrams are important to find

Operational diagram	Non-terminal	Side	Terminal	I(x)	S(x)
customer name	Customer	left	yes		name
customer list	Customers	left,right	no	inCS	outCS
rental list	Rentals	left	yes		outRS
days rented	Rental	left	yes		daysRented
income	Rental	right	no		income
movie list	Movies	left,right	no	inMS	outMS
movie title	Movie	left	yes		title
movie type	Price	left	yes		type

Table 9: Attributes mapping to nonterminals

Attribute	Starting non-terminal
outCS	Video_Store
outMS	Video_Store
income	Video_Store

Table 10: Attributes in starting non-terminal Video\_Store

information that should be present in starting non-terminal *VideoStore*. In the case study of Video Store, three distinct attributes are defined in operational diagram: *customer list*, *movie list* and *income*. Therefore all three attributes are synthesized in starting non-terminal (Table 10).

Attribute	Other nonterminals
inMS	Customers, Customer, Rentals
name	Rentals
inCS	Rentals, Customer
outCS	Customer, Rentals
type	Movie
title	Rental
income	Rentals, Customers, Customer

Table 11: Attributes in other nonterminals

To be able to compute services, attributes are propagated through non-terminals. Table 11 brings remaining mapping of operational diagrams attributes to non-terminals.

**The semantics-2.nd phase:** After detailed attribute description, we can write specifications in attribute grammar. The only semantic part left, is to define functions for attribute evaluation. The specifications are broken into separate non-terminal descriptions.

The first production is `VideoStore`  $\rightarrow$  `Movies Customers`. The non-terminal defines element of entity movies and customers. To keep video store information we define two attributes for each entity. Attributes are of type TABM (movie list) and TABC (customer list), which is a mapping function.

Further, the attribute `income` is used to evaluate renting service (see Table 10).

```
TABM = FF(string, (string, int))
TABC = FF(string, (string, TABR))
```

NonTerm VideoStore:

```
Inh: {}
Syn: {outMS: TABM, outCS: TABC,
      income: int}
```

```
mkVideoStore(VideoStore ->
              Movies Customers):
  VideoStore.outMS = Movies.outMS
  VideoStore.outCS = Customers.outCS
  VideoStore.income = Customers.income
```

```
Movies.inMS = {}
```

```
Customers.inMS = Movies.outMS
Customers.inCS = {}
```

For collecting the elements of entity `movie`, we use non-terminals `Movies` and `Movie` (see Section 4). The semantic of the non-terminal is described with attributes `inMS` and `outMS` (see Table 9), where first attribute `inMS` is inherited and `outMS` synthesized. The function `insert()` adds an element of pair (name, type) to movie table. If the movie is already in the collection, the element is not added in the collection of movies. This is represented with contextual condition (CC).

```
insert: TAB * (string * Price) --> TAB
```

NonTerm Movies:

```
Inh: {inMS: TABM}
Syn: {outMS: TABM}
```

```
oneMoreMovie(Movies -> Movies Movie):
  Movies/0.outMS =
    insert(Movies/1.outMS,
```

```

        new Movie(Movie.title,
                  Movie.type))

Movies/1.inMS = Movies/0.inMS

CC: (NOT(Movie.title IN
        Movies/1.outMS))

emptyMovies(Movies -> &):
    Movies.outMS = Movies.inMS

```

Semantic constructs of non-terminal `Movie` are shown below. The symbol `Movie` is semantically described with two attributes that represent basic data of the movie entity.

```

NonTerm Movie:
    Inh: {}
    Syn: {title: String, type: Price}

getMovie(Movie -> title Price):
    Movie.title = title.lexval
    Movie.type = Price.type

```

The entity `Customer` follows the same principle as shown at the non-terminal `Movies`. The multiplicity `0..m` brings the use of the non-terminals `Customers` and `Customer`. The function `insert()` adds an element (new customer) to customer list. The insertion is skipped, if the name is already contained in the list.

```

insert: TABC * (string) --> TABC

NonTerm Customers:
    Inh: {inCS: TABC, inMS: TABM}
    Syn: {outCS: TABC, income: int}

oneMoreCustomer(Customers ->
                 Customers Customer):
    Customers/0.outCS = CUSTOMER.outCS;
    Customers/0.income = Customers/1.income +

```

```

                                Customer.income

Customers/1.inMS = Customers/0.inMS
Customers/1.inCS = Customers/0.inCS

Customer.inMS    = Customers/0.inMS
Customer.inCS    = Customers/1.outCS

CC: (NOT(Customer.name IN Customers/1.outCS))

emptyCustomer(Customers -> &):
  Customers.outCS = Customers.inCS;
  Customer.income = 0.0

```

Semantics constructs of non-terminal `Customer` consist of attributes `name` (type string), `inCS` (inherited enumeration of customers), `outCS` (synthesized enumeration of customers) and `outMS` (synthesized enumeration of movies).

NonTerm `Customer`:

```

  Inh: {inCS: TABC, inMS: TABM}
  Syn: {name: String, outCS: TABC,
        income: int}

```

```

getCustomer(Customer ->
             name Rentals):
  Customer.outCS = Rentals.outCS
  Customer.income = Rentals.income
  Customer.name  = name.lexval

  Rentals.inMS    = Customer.inMS
  Rentals.inCS    = insert( Customer.inCS,
                             new Customer(Customer.name))
  Rentals.name    = Customer.name

```

To define rental items, the non-terminal `Rentals` holds three distinct inherited attributes: `inMS`, `inCS` and `name`. To keep the final value after mapping

rentals to specific customer, the synthesized attribute `outCS` is used. To support the rental charging service, a synthesized attribute `income` is applied.

```
addRental: TABC * (string, TABR) * (mov, int) --> TABC
getCustomer: TABC * string --> (string, TABR)
getMovie: TABM * string --> mov
getCharge: mov * int --> real
```

```
TABR = FF(string, (Movie, int))
```

NonTerm Rentals:

```
Inh: {inMS: TABM, inCS: TABC,
      name: String}
Syn: {outCS: TABC, income: int}
```

```
oneMoreRental(Rentals -> Rentals Rental):
```

```
Rentals/0.outCS =
  addRental(Rentals/1.outCS,
    getCustomer(Rentals/1.outCS,
      Rentals/0.name),
    new Rental(getMovie(Rentals/0.inMS,
      Rental.title),
      Rental.daysRented))
Rentals/0.income = Rentals/1.income +
  getCharge(getMovie(Rentals/0.inMS,
    Rental.title),
    Rental.daysRented)
```

```
Rentals/1.inMS = Rentals/0.inMS
Rentals/1.inCS = Rentals/0.inCS
Rentals/1.name = Rentals/0.name
```

```
CC: ((Rental.title IN Rentals/0.inMS)
     AND (Rentals.name IN
        Rentals/0.inCS))
```

```
emptyRental(Rentals -> &):
  Rentals.outCS = Rentals.inCS
  Rentals.income = 0.0
```

As shown above, for mapping the rental items to customer, the function `addRentals()` is defined. It uses several other functions to provide renting service as depicted in current modul of attribute grammar specifications. The mapping process is prevented if rented movie is not present in inherited attribute `inMS` and also if customer is not present in inherited attribute `inCS`. This is shown above with contextual condition.

The semantic of non-terminal `Rental` is specified using the values returned by the scanner. Therefore, attributes `title` (inherited from non-terminal `Movie`) and `daysRented` are used.

NonTerm Rental:

```
Inh: {}  
Syn: {title: String,  
      daysRented: int}
```

```
getRental(Rental -> daysRented Movie):  
  Rental.title = Movie.title  
  Rental.daysRented = atoi (daysRented.lexval)
```

The non-terminal `Price` represents class `Price` from the conceptual class diagram. This is an abstract class which defines three subclasses, classes `Reg`, `Child` and `New` (non-terminals `Reg`, `Child` and `New`) in the conceptual class diagram. Because of the final class rule, non-terminals are replaced with terminals.

NonTerm Price:

```
Inh: {}  
Syn: {type: Price}
```

```
getPriceNew(Price -> new):  
  Price.type = new New()
```

```
getPriceReg(Price -> reg):  
  Price.type = new Reg()
```

```
getPriceChild(Price -> child):  
  Price.type = new Child()
```

## 5.4 CS4: Cleaning Robot

### The problem:

Consider a small robot whose mission is to clean a rectangular area (for instance, an industrial pavilion). The robot can move straight-ahead up, down, right and left, a given number of steps (each step is one length unit); by default the move instruction corresponds to 1 step. During the movement, the robot can have the cleaning system on or off.

Let us suppose that the area is  $L$  unit length by  $C$  unit high; then the pavilion can be seen as a rectangular grid  $L \times C$ .

The start position—point with coordinates  $(0, 0)$  where the robot is initially placed, before starting its activity—is the left lower corner of the rectangle. Each time the robot receives a command to move forward, it is necessary to validate the movement comparing its current position and the required number of steps against the pavilion dimensions, so that the robot does not go out of the borders (in that case, it will not move); a warning message should be emitted if the command is invalid.

Simulating the behavior of the Cleaning Robot, the goal of this problem is to determine the space cleaned (the grid cells visited by the robot when the cleaning system is on), after a sequence of movement commands.

**The problem specification:** After the analysis of the case study, the discovering of the main functionalities is done with an use case diagram (Figure 11).

For the case study of the Cleaning Robot (Fig. 11) we identified three main services represented with use cases *Define space*, *Robot movement* and *Report*. Description of the scenarios follows:

Scenario for Define space use case:

1. Request the length of space.
  2. Request the width of space.
  3. Construct the pavilion dimensions.
- Use case end.

Scenario for Robot movement use case:

1. Request for the direction of next robot movement.
2. Request number of robot steps.
3. Put the cleaning system on or off.
4. Compute the next robot position.



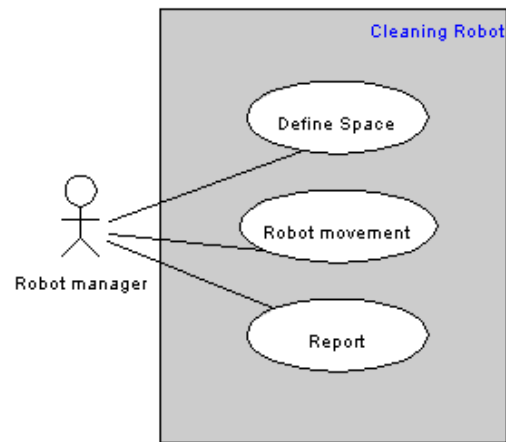


Figure 11: Use case diagram for the case study of Cleaning Robot

5. Move cleaning robot toward new position.  
Use case end.

ALT 4a: New robot position is out of bounds.  
Movement skipped. Use case end.

Scenario for Report use case:

1. Request the cleaning robot to stop.
  2. Provide the state of pavilion.
  3. Compute the robot position.
- Use case end.

**The Class Diagram:** To solve the problem above, all the information that we need to handle is the rectangle dimensions and the sequence of movement commands.

If we call `CleanTask` to the main concept under analysis, we will say that it is defined in terms of two other concepts: `Dimensions` and `Moves`.

The first one has two components, `Length` and `Width`, that express the number of step units in both dimensions of the rectangular area to clean.

`Moves` is a sequence of movements, and a `Move` is described in terms of `Direction` (up, down, left, or right), number of `Steps`, and `CleanSystem` status (on, or off).

The Diagram of figure 12 shows the classes and their relationships that result

from the problem analysis.

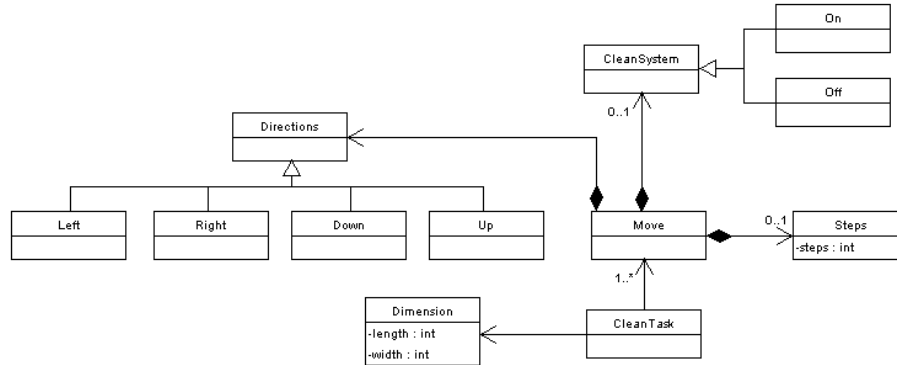


Figure 12: Conceptual class diagram for Cleaning Robot

**The Structure:** Following the proposed approach, and taking into account the conceptual class diagram of figure 12, we can formalize the structure of the problem domain (the complex and primitive concepts and the inter-relations among them) in terms of the CFG presented bellow. Remember that the classes are denoted by non terminal symbols, and the atomic values by terminal symbols.

```

CleanTask  -> Dimensions Moves
Dimensions -> length width
Moves      -> Moves Move
           | Move
Move       -> Direction Steps CleanSystem
Direction -> LEFT | RIGHT | DOWN | UP
Steps      -> steps | &
CleanSystem -> ON | OFF | &
  
```

**The Semantics:**

After the generic overview of the classes and the relationships given by the CCD and the CFG, we describe their semantics, writing an operational diagram and attribute mapping table. Afterwards, specification of attribute grammar can be written.

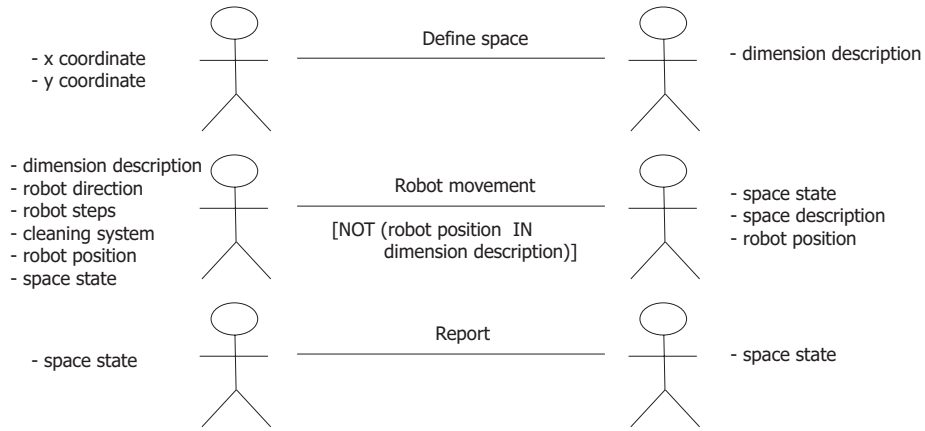


Figure 13: The operational diagram for Cleaning Robot

**The Semantics–1.st and last phase:** Again, use case diagram for system Cleaning Robot helps to find semantic information. From use case diagram three operational diagrams are created, for use cases *Define space*, *Robot movement* and *Report* (Fig. 13). Collaboration between robot manager and system carries attributes possessed and synthesized in the use case of *Define space*. To perform that service, the actor needs to possess following operational diagram attributes: *x coordinate* and *y coordinate*. Synthesized attribute is *dimension description*.

Operational diagram	Non-terminal	Side	Terminal	I(x)	S(x)
x coordinate	Dimensions	left	yes	inAS	width
y coordinate	Dimensions	left	yes		length
dimension description	Dimensions	left, right	yes	inBounds	outBounds
robot direction	Directions	left	yes		direct
robot steps	Steps	left	yes		steps
cleaning system	CleanSys	left	yes		clean
robot position	Moves	left, right	no	inPos	outPos
space state	Moves	left, right	no	inCleanPts	outCleanPts

Table 12: Attributes mapping to non-terminals

Further, the operational diagrams are used to define semantics of the prob-

lem domain (Table 12). The first attribute from operational diagram *Define space* is *x coordinate*. To begin with, the source non-terminal of the attribute has to be defined (**Dimensions**). Next, name of the attribute in AG has to be chosen properly (**x**). Finally, the type of attribute in attribute grammar has to be defined. Attribute *x coordinate* has a terminal source in context-free grammar, therefore attribute is assigned as synthesized.

Attribute	Starting non-terminal
outBounds	CleanTask
outCleanPts	CleanTask
outPos	CleanTask

Table 13: Attributes in starting non-terminal CleanTask

Table 13 shows attributes mapped to starting non-terminal CleanTask. To complete semantics of problem domain, some attributes are propagated through non-terminals. Table 14 shows non-terminals propagating attribute values. Detailed description of defined attributes (Tables 12, 13 and 14) follows in next semantic phase.

Attribute	Other non-terminals
outCleanPts	Move
outBounds	Move
inPos	Move
outPos	Move
inCleanPts	Move

Table 14: Attributes in other nonterminals

**The Semantics–2.nd phase:** The objective of this AG is to define how to evaluate the value of the attribute **outCleanPts** (the set of cleaned points, corresponding to the grid cells visited by the robot while the cleaning mechanism was on) associated with the grammar start symbol **CleanTask**. This value is computed starting from the initial position (the attribute **inPos** (actual position) is initialized with the coordinates (0, 0)) and determining the visited cells according to the list of movement commands. To execute just valid movements, it is necessary to take into Account the area boundaries (denoted by the attributes **inBounds** and **outBounds**) as declared in

the context of the symbol `Dimensions`.  
The following module specifies the statements above.

```
mkPair: int * int --> point

point: int * int
TAB= FF( point, bool )

NonTerm CleanTask:
  Inh: {}
  Syn: { outCleanPts: TAB, outBounds: int*int,
        outPos: point }

cleanManager(CleanTask -> Dimensions Moves):
  CleanTask.outCleanPts = Moves.outCleanPts
  CleanTask.outPos = Moves.outPos
  CleanTask.outBounds = Dimensions.outBounds

  Moves.inCleanPts = {}
  Moves.inPos = mkPair(0,0)
  Moves.inBounds = Dimensions.outBounds
```

The symbol `Dimensions` is semantically described by two integer attributes that represent the two-dimensional measures (`x` and `y`) of the rectangular surface to clean (`outBounds`).

```
NonTerm Dimensions:
  Inh: {}
  Syn: { x,y: int, outBounds: int*int }

calcDim(Dimensions -> length width):
  Dimensions.x = atoi (length.lexval)
  Dimensions.y = atoi (width.lexval)
  Dimensions.outBounds = mkPair(Dimensions.x,
                                Dimensions.y)
```

The main part of this AG is concerned with the evaluation of the cleaned points table and the final position of the robot. Both computations are

specified in the context of the module for symbol Move, that is the element constituent of the sequence Moves. Updating the cleaning space table (outCleanPts) and next robot position (attribute outPos) is performed by function updateTab() and updatePos(). Function inbounderies() (used in context condition) validates the new robot position.

```

updatePos: (int*int) * point * ({LEFT,RIGHT,DOWN,UP}) * int
          --> point
inbounderies: (int*int) * point * ({LEFT,RIGHT,DOWN,UP}) * int
          --> bool
updateTab: (int*int) * point * TAB * ({LEFT,RIGHT,DOWN,UP}) *
          int --> TAB

```

NonTerm Moves:

```

Inh: { inCleanPts: TAB
      inBounds: int*int
      inPos: point    }
Syn: { outCleanPts: TAB
      outPos: point   }

```

oneMoreMove(Moves -> Moves Move):

```

Moves/0.outCleanPts = Move.outCleanPts
Moves/0.outPos = Move.outPos

```

```

Moves/1.inCleanPts = Moves/0.inCleanPts
Moves/1.inPos = Moves/0.inPos
Moves/1.inBounds = Moves/0.inBounds

```

```

Move.inCleanPts = Moves/1.outCleanPts
Move.inPos = Moves/1.outPos
Move.inBounds = Moves/0.inBounds

```

firstMove(Moves -> Move):

```

Moves.outCleanPts = Move.outCleanPts
Moves.outPos = Move.outPos

```

```

Move.outCleanPts = Moves.inCleanPts
Move.inPos = Moves.inPos
Move.inBounds = Moves.inBounds

```

NonTerm Move:

```
Inh: { outBounds: int*int
      inPos: point
      inCleanPts: TAB  }
Syn: { outCleanPts: TAB
      outPos: point    }
```

```
getMove(Move -> Direction Steps CleanSystem):
  Move.outCleanPts = if ( CleanSys.clean==ON )
    updateTab( Move.inCleanPts, Move.inBounds,
              Move.inPos, Direction.direct, Steps.steps )
    else Move.inCleanPts
  Move.outPos = updatePos( Move.inPos, Move.inBounds,
                          Direction.direct, Steps.steps )

CC: ( inboundaries( Move.inBounds, Move.inPos,
                    Direction.direct, Steps.steps ) )
```

The semantic definition of the other concepts, the rest of the non-terminal symbols, is obvious and very easy to write as follows.

NonTerm Direction:

```
Inh: { }
Syn: { direct: {LEFT,RIGHT,DOWN,UP} }
```

```
mkDirLeft(Direction -> LEFT):
  Direction.val=LEFT
```

```
mkDirRight(Direction -> RIGHT):
  Direction.val=RIGHT
```

```
mkDirDown(Direction -> DOWN):
  Direction.val=DOWN
```

```
mkDirUp(Direction -> UP):
  Direction.val=UP
```

NonTerm Steps:

```

    Inh: { }
    Syn: { steps: int }

mkSteps(Steps -> int):
    Steps.val = atoi(steps.lexval)

emptySteps(Steps -> &):
    Steps.val = 1

NonTerm CleanSys:
    Inh: { }
    Syn: { val: {ON,OFF} }

mkCleanOn(CleanSys -> ON):
    CleanSys.clean=ON

mkCleanOff(CleanSys -> OFF):
    CleanSys.clean=OFF

emptyClean(CleanSys -> &):
    CleanSys.clean=ON

```

## 6 An OO approach to the implementation of AG-based Processors

Attribute grammars have been introduced by [Knu68] and since then are proved to be useful in specifying the semantics of programming languages, in automatic constructing of compilers/interpreters, in specifying and generating interactive programming environments and in many other areas. While implementation of programming languages is the original and most widely recognized area of attribute grammars, they are also used in many other areas such as: natural language interfaces, graphical users interfaces, visual programming, pattern recognition, hardware design, communication protocols, software engineering, static analysis of programs, databases, etc. However, only a few commercial compilers have been developed using attribute grammars as a design and implementation tool. In [Wai90] it has been argued that attribute grammars are unsuitable for production of high-speed compilers for general-purpose programming languages, since attribute



grammars are just a model of compilation and thus too primitive for a real engineering discipline, and that they do not directly support the generation and optimization of the machine code. The first problem is concerned with pragmatic aspects of ordinary attribute grammars. Ordinary attribute grammars have deficiencies which become apparent in specifications for real programming languages. Such specifications are large, unstructured, and hard to understand, modify and maintain. Yet worse, small modifications of some parts in the specifications have widespread effects on the other parts of the specifications, since specifications are not modular, extensible and reusable. There has been a lot of research work on augmenting ordinary attribute grammars with extensions to overcome the deficiencies of attribute grammars such as lack of modularity, extensibility and reusability. Several concepts, such as remote attribute access, object-orientation, templates, symbol computations, high order features etc., have been implemented in various attribute grammar specification languages. Therefore, the first problem has been solved by introducing concepts and primitives from programming paradigms such as object-oriented, functional and logic programming. A detailed survey of attribute grammar based specification languages is given in [Paa95]. Let us look in little bit more detail to object-oriented attribute grammars. The common paradigm is `Nonterminal = Class` where context-free grammars define the class hierarchy. Nonterminals act as classes organized into hierarchy. A slightly broader paradigm is `Production = Class` where each production is class that specifies the syntactic structure, attributes and semantic rules. All these elements can be inherited, specialized, and overridden in subclasses (more specialized nonterminals in the particular context-free grammar). Our approach is completely different since the attribute grammar as a whole is a subject to inheritance. It is based on paradigm `Attribute grammar = Class` [MLEŽ00]; we named it multiple attribute grammar inheritance. The benefits of multiple attribute grammar inheritance are:

- specifications are extensible since the language designer writes only new and specialized specifications,
- specifications are reusable since specifications are inherited from ancestor specifications, and
- for each language increment a compiler can be generated and the language tested.

## 6.1 LISA system

Multiple attribute grammar inheritance was successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0 [MLAŽ00]. Language designer/implementer is able to add new features (syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications. The tool LISA is compiler generator with the following features:

- LISA is platform independent since it is written in Java
- it offers the possibility to work in a textual or visual environment
- it offers an integrated development environment (Fig. 14) where users can specify - generate - compile-on-the-fly - execute programs in a newly specified language
- lexical, syntax and semantic analyzers can be of different types and can operate standalone; the current version of LISA supports LL, SLR, LALR, and LR parsers, tree-walk, parallel, L-attribute and Katayama evaluators
- visual presentation of different structures, such as finite state automata, BNF, syntax tree, semantic tree, dependency graph
- animation of lexical, syntax and semantic analyzers
- the specification language supports multiple attribute grammar inheritance which enable to design a language incrementally or reuse some fragments from other programming language specifications.

Let us look at the informal definition of multiple attribute grammar inheritance; the formal definition of multiple attribute grammar inheritance is described in [MLAŽ99]. Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications, may override some specifications from ancestors or even defeat some ancestor specifications. With inheritance we can extend the lexical, syntax and semantic parts of the programming language specification. Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overridden from ancestor specifications. In object-oriented languages the properties that consist of instance variables and methods are subject to modification. Since in

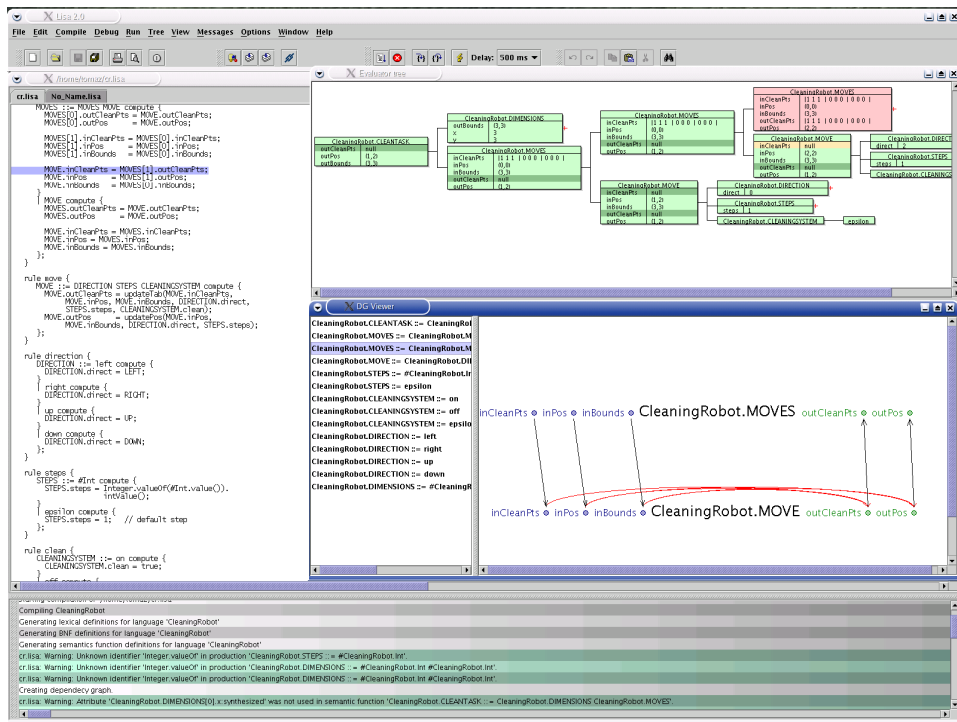


Figure 14: LISA Integrated Development Environment

attribute grammars semantic rules are tightly coupled with particular production rules, properties in multiple attribute grammar inheritance consist of:

- lexical regular definitions,
- attribute definitions,
- rules which are generalized syntax rules that encapsulate semantic rules, and
- methods on semantic domains.

Therefore, a language is specified in the following manner:

language  $L_1$  [extends  $L_2, \dots, L_N$ ] {  
 lexicon {  
 [[P] overrides | [P] extends] R regular expr.

```

        :
    }
    attributes type A1, ..., AM
        :
    rule [[Y] extends | [Y] overrides] Z {
        X ::= X11 X12 ... X1p compute {
            semantic functions }
        :
        |
        Xr1 Xr2 ... Xrt compute {
            semantic functions }
        ;
    }
        :
    method [[N] overrides | [N] extends] M {
        operations on semantic domains
    }
        :
    }

```

In lexical analysis or scanning, the stream of characters representing the source program is read from left to right and grouped into tokens. The lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit. Regular expressions, which are the most frequently used formal method for specifying patterns are also used in LISA. More precisely, LISA uses regular definitions where each regular expression is associated with a unique name:

```

R1  regularexpr1
R2  regularexpr2
:
Rn  regularexprn

```

The name of regular expression ( $R_i$ ) is an identifier which is described with following regular expression  $[a - zA - Z\_][a - zA - Z\_0 - 9]^*$ . Regular expression *regularexpr* <sub>$i$</sub>  is a sequence of characters until the end of current

line (spaces are ignored). Regular expressions describe patterns with whom strings of characters in the source program are matched. LISA uses following rules for describing regular expressions:

expression	matches
x	the character "x"
\x	"x" even if x is a special character
\0xFF	character represented by ASCII code
[s]	any character in the string s
[x-y]	any character in the range from x to y
[^s]	any character not in the string s
x*	0 or more instances of x
x+	1 or more instances of x
x?	0 or 1 instance of x
x y	an x or y
(x)	an x
#s	an expression defined by s

Table 15: Regular expressions in LISA

Where special characters (meta-symbols) in LISA's lexical specifications are:

| \ ( ) \* + ? [ ] - # ^

Some examples of regular expressions written in LISA are:

```

comment      /\*[\^\\*]+\*/
Some_ink     [pr]?ink
Keyword      \0x42egin | end
ID           [a-zA-Z_][a-zA-Z0-9_]*
NUMBER       [0-9]
Integer      #NUMBER+
Operator     \+ | \- | \* | / | < | > | = | := | \#
Delimiter    ; | . | \( | \) | ,
WhiteSpace   [\ \0x0A\0x0D\0x09]+

```

The first regular expression describes multi-line comment start with characters /\* and end with \*/ (e.g. /\* this is comment \*/). It is important to notice

that above regular expression for comment do not allow that characters between `/*` and `*/` contain the character `*`. The second regular expression match with the words *pink*, *rink*, and *ink*; but not with *prink*. The third regular expression match with the words *Begin* and *end*. Other regular expressions above are simple and do not need extra explanation.

In syntax analysis, tokens of a source program are grouped into grammatical phrases. The task of the syntax analyzer or parser is to determine if a string of tokens can be generated by a grammar phrase. The syntax of the programming language is usually described by the well known BNF notation. In LISA standard BNF conventions are used; context-free productions are specified in the rule part of language definition using following conventions:

```
X ::= X11 X12 ... X1p
      |
      X21 X22 ... X2r
      |
      ⋮
      |
      Xn1 Xn2 ... Xns
      ;
```

where  $X$  is a left-hand nonterminal and  $X_{ij}$  is a terminal or nonterminal symbols. To distinguish between nonterminal and terminal symbols LISA uses following convention. Nonterminal is described with following regular expression  $[A-Z][A-Z0-9]^*$ . Therefore, first character must be capital letter followed by capital letters, digits or character `_`. All other symbols are terminal symbols. Empty production is written as:  $X ::= \textit{epsilon}$ . Special symbols in syntax specifications are:

```
::= | epsilon ;
```

The BNF for arithmetic expressions written in LISA is:

```
E ::= T EE ;
EE ::= + T EE | epsilon ;
T ::= F1 TT ;
TT ::= * F1 TT | epsilon ;
F1 ::= #Integer ;
F1 ::= ( E ) ;
```

When the syntax of sentences is correct the meaning of sentences or semantics can be computed. The meaning of programs in LISA is described with

attribute grammars. An attribute grammar is based on a context-free grammar and associates attributes with the nodes of a parse tree, thus obtaining an attributed or semantic tree. Attribute evaluation rules are associated with the each context-free production. In LISA attribute evaluation rules are written in a block { } which start with keyword *compute*. This block is inserted between the last symbol of particular production and the character ; or |. Semantic rule in LISA is actually Java assignment statement; For example, to the first two above productions the following semantic rules are associated:

```

E ::= T EE compute {
        E.val = EE.val;
        EE.inVal = T.val; }
    ;

EE ::= + T EE compute {
        EE[0].val = EE[1].val;
        EE[1].inVal = EE[0].inVal + T.val; }
    | epsilon compute {
        EE.val = EE.inVal; }
    ;

```

Attributes in the node can be of two kinds: the inherited attributes, whose values are obtained from the siblings and the parent of that node in the parse tree, and the synthesized attributes, whose values are obtained from the children of that node in the parse tree. The type of attributes (inherited or synthesized) is derived by LISA and hence is no need to be specified by LISA user. However, the type of attributes and nonterminals to which attributes are attached have to be specified. In LISA the type of attribute can be any valid Java type. In LISA attributes are defined in the block which starts with the keyword *attributes*. Usually, the same attribute name is attached to many different nonterminals. To avoid unpleasant repeating the wild character \* can be used instead. An example of attribute definitions for above example is:

```

attributes int *.val, EE.inVal, TT.inVal;

```

## 6.2 Case Studies Implementation

### 6.2.1 CS1: Vending Machine

Next subsection shows implementation of vending machine according to grammatical specifications described in subsection 5.1. Each stock is described with its name, price and quantity. The semantics of language construct is described by attributes (StkTab, FStkTab, price, etc.) already presented in subsection 5.1. Each attribute has type which have to be defined in concrete implementation. To represent stock, attributes StkTab and FStkTab, the Hashtable has been chosen. The type of other attributes are double, int and String. Notice how contextual constraints have been implemented. They are coded directly in the LISA methods.

Keywords "stock\_description" and "sales\_description" have been introduced to explicitly show the difference between stock and sales descriptions. Without these keywords the grammar is not LR(1).

```
language VM_EXAMPLE{

  lexicon {
    Word      [a-zA-Z_] [a-zA-Z0-9_]*
    Int       [0-9]+
    Real      [0-9]+\.[0-9]*
    ignore    [\ \0x0A\0x0D\0x09]+
    Keyword   stock | sales_description
  }

  attributes
    Hashtable *.StkTab, *.FStkTab;
    double PRODPRICE.price;
    int PRODQTY.quantity;
    String PRODNAME.name;
    double MONEY.amount;
    double VM.income;
    double *.sum;

  rule VM {
    VM ::= stock_description STOCKS sales_description SALES
    compute {
      VM.FStkTab = SALES.FStkTab;
    }
  }
}
```



```

    VM.income = SALES.sum;

    STOCKS.StkTab = new Hashtable();

    SALES.StkTab = STOCKS.FStkTab;
};
}

rule Stocks {
    STOCKS ::= STOCKS STOCK compute {
        STOCKS[0].FStkTab = STOCK.FStkTab;

        STOCKS[1].StkTab = STOCKS[0].StkTab;

        STOCK.StkTab = STOCKS[1].FStkTab;
    }
    | STOCK compute {
        STOCKS.FStkTab = STOCK.FStkTab;

        STOCK.StkTab = STOCKS.StkTab;
    };
}

rule Sales {
    SALES ::= SALES SALE compute {
        SALES[0].FStkTab = SALE.FStkTab;
        SALES[0].sum = SALES[1].sum + SALE.sum;

        SALES[1].StkTab = SALES[0].StkTab;

        SALE.StkTab = SALES[1].FStkTab;
    }
    | epsilon compute {
        SALES.FStkTab = SALES.StkTab;
        SALES.sum = 0.0;
    };
}

rule Stock {

```

```

    STOCK ::= PRODNAMe PRoDPRICe PRoDQTY compute {
        STOCK.FStkTab = insert(STOCK.StkTab, PRODNAMe.name,
                               PRoDPRICe.price, PRoDQTY.quantity);
    };
}

rule Sale {
    SALE ::= PRODNAMe MONEy compute {
        SALE.FStkTab = update(SALE.StkTab, PRODNAMe.name,
                              MONEy.amount);
        SALE.sum = MONEy.amount;
    };
}

rule ProdName {
    PRODNAMe ::= #Word compute {
        PRODNAMe.name = #Word.value();
    };
}

rule ProdPrice {
    PRoDPRICe ::= #Real compute {
        PRoDPRICe.price = Double.valueOf(#Real.value()).
                           doubleValue();
    };
}

rule ProdQty {
    PRoDQTY ::= #Int compute {
        PRoDQTY.quantity = Integer.valueOf(#Int.value()).
                              intValue();
    };
}

rule Money {
    MONEy ::= #Real compute {
        MONEy.amount = Double.valueOf(
            #Real.value()).doubleValue();
    };
}

```

```

method M_Stock {
  class Stock {
    String name;
    double price;
    int qty;
    Stock(String name, double price, int qty) {
      this.name = name;
      this.price = price;
      this.qty = qty;
    }

    public int getQty() {
      return qty;
    }

    public double getPrice() {
      return price;
    }

    public String toString() {
      return "(" + price + " Euro," + qty + ")";
    }

  } // class
} // method

method M_insert {
  import java.util.*;
  Hashtable insert(Hashtable stock, String name,
                  double price, int qty) {
    Item i = (Item)stock.get(name);
    if (i == null) {
      stock.put(name, new Item(name, price, qty));
    }
    else {
      System.out.println("Item " +name +
                        " is already in the stock");
    }
  }
  return stock;
}

```

```

    } // java method
} // Lisa method

method M_update {
    Hashtable update(Hashtable stock, String name,
                    double money) {
        Item i = (Item)stock.get(name);
        if (i != null) {
            if ((i.getQty() > 0) && (i.getPrice()==money))
                stock.put(name, new Item(name, money,
                    i.getQty()-1));
            else
                System.out.println("Item " + name +
                    "is out of stock or invalid amount of money");
        }
        else {
            System.out.println("Item " +name +
                "is not in the stock"); }
        return stock;
    } // java method
} // Lisa method

} // language VM

```

From above specifications an VM compiler is automatically generated by LISA system. Keywords "stock\_description" and "sales\_description" have been introduced to explicitly show the difference between stock and sales descriptions. Without these keywords the grammar is not LR(1). One of the possible scenarios is now described with the following program:

```

stock_description
mars    0.50 10
kitkat  0.60 15
twix    0.60 5
sales_description
twix    0.60
twix    0.60
mars    0.50
twix    0.60

```

The meaning of above program is the following stock list and income:

```
FStkTab: {twix    =(0.60 Euro, 2),
          kitkat =(0.60 Euro, 15),
          mars   =(0.50 Euro, 9)}
income: 2.30
```

## 6.2.2 CS2: Automatic Teller Machine

The implementation of ATM with previous defined grammatical specification (see Subsection 5.2) are presented in following subsection. Each account is described with id of user, his personal identical number (pin) and balance. To represent Bank attributes, inAS and outAS has been defined. Attributes are collections of accounts. To show this, Hashtable has been chosen.

Notice how additional code for inserting object of account in Hashtable was implemented in method insert. LISA specifications of language ATM\_EXAMPLE follows below.

To use accounts inserted in the Bank, we defined ATM withdraw service. To add withdraw functionality some attributes are added (amount).

```
language ATM_EXAMPLE {
  lexicon {
    id [A-Z][0-9]+
    pin [0-9]+
    real [0-9]+\.[0-9]*
    reserved \account | withdraw
    ignore [\ \0x0A\0x0D\0x09]+
  }

  attributes
    Hashtable *.outAS,*.inAS;
    String    ID.id;
    int       PIN.pin;
    double    BALANCE.balance;
    double    MONEY.amount;

  rule ATM {
    ATM ::= BANK WITHDRAW compute {
      ATM.outAS    = WITHDRAW.outAS;
    }
  }
}
```

```

        WITHDRAW.inAS = BANK.outAS;
    };
}

rule Bank {
    BANK ::= ACCOUNTS compute {
        BANK.outAS = ACCOUNTS.outAS;

        ACCOUNTS.inAS = new Hashtable();
    };
}

rule Accounts {
    ACCOUNTS ::= ACCOUNTS ACCOUNT compute {
        ACCOUNTS[0].outAS = ACCOUNT.outAS;
        ACCOUNTS[1].inAS = ACCOUNTS[0].inAS;
        ACCOUNT.inAS = ACCOUNTS[1].outAS;

    }
    | ACCOUNT compute {
        ACCOUNTS.outAS = ACCOUNT.outAS;
        ACCOUNT.inAS = ACCOUNTS.inAS;
    };
}

rule Account {
    ACCOUNT ::= account ID PIN BALANCE compute {
        ACCOUNT.outAS = insert(ACCOUNT.inAS, new
            Account(ID.id, PIN.pin, BALANCE.balance));
    };
}

rule Withdraw {
    WITHDRAW ::= withdraw ID PIN MONEY compute {
        WITHDRAW.outAS = update (WITHDRAW.inAS,
            setAccount(get(WITHDRAW.inAS, ID.id, PIN.pin),
                MONEY.amount));
    };
}

```

```

rule Id {
    ID ::= #id compute {
        ID.id = #id.value();
    };
}

rule Pin {
    PIN ::= #pin compute {
        PIN.pin = Integer.valueOf(#pin.value()).intValue();
    };
}

rule Balance {
    BALANCE ::= #real compute {
        BALANCE.balance = Double.valueOf(#real.value()).
            doubleValue();
    };
}

rule Money {
    MONEY ::= #real compute {
        MONEY.amount = Double.valueOf(#real.value()).
            doubleValue();
    };
}

method A_Item {
    class Account {
        String id;
        int pin;
        double balance;
        Account( String id, int pin, double balance) {
            this.id = id;
            this.pin = pin;
            this.balance = balance;
        }

        public String toString() {
            return "(" + this.id + ", " + this.pin + "," +
                this.balance + ")";
        }
    }
}

```

```

    }

    public String getId() {
        return this.id;
    }

    public int getPin() {
        return this.pin;
    }

    public double getBalance() {
        return this.balance;
    }

    public void setBalance(double amount) {
        this.balance-=amount;
    }

}

}

method A_Insert {
    import java.util.*;
    Hashtable insert (Hashtable aAccounts, Account aAccount) {
        Account hAccount=(Account)aAccounts.get(aAccount.getId());
        if (hAccount==null)
            aAccounts.put(aAccount.getId(), aAccount);
        else
            System.out.println("Account" + aAccount.getId() +
                "is already in bank");
        return aAccounts;
    } // java method
} // Lisa method

method A_update {
    import java.util.*;
    Hashtable update (Hashtable aAccounts, Account aAccount) {
        if (aAccount!=null){
            aAccounts.remove(aAccount.getId());
            aAccounts.put(aAccount.getId(), aAccount);
        }
    }
}

```



```

    }
    else
        System.out.println("Account empty");
    return aAccounts;
} // java method
} // Lisa method

method A_get {
import java.util.*;
Account get (Hashtable aAccounts, String id, int pin) {
    Account aAccount = (Account)aAccounts.get(id);
    if (aAccount == null) {
        System.out.println("No such account " + id + " " + pin);
        return null;
    }
    else {
        if (aAccount.getPin()==pin)
            return aAccount;
        else {
            System.out.println("Not valid PIN");
            return null;
        }
    }
} // java method
} // Lisa method

method A_set {
import java.util.*;
Account setAccount (Account aAccount, double amount) {
    if (aAccount != null){
        double balance =aAccount.getBalance();
        if (balance - amount >= 0 )
            aAccount.setBalance(amount);
    }
    else
        System.out.println("Service rejected - to big amount");
    return aAccount;
} // java method
}
}

```

The LISA from above specifications automatically generate a compiler. To show functionality of this specification following program is introduced.

```
account A1032 123 234.32
account B1002 213 34343.34
account D2134 344 35345.80
```

```
withdraw A1032 123 20.50
```

The above program produces following result:

```
OutAs {A1032={A1032, 123, 213,82},
       B1002={B1002, 213, 34343,34},
       D2134={A2134, 344, 35345,80}}
```

### 6.2.3 CS3: Video Store

The transformation of attribute grammar specifications (see Subsection 5.3) to LISA specifications are very straightforward. They following principles described in previous examples.

```
language VIDEO_STORE_EXAMPLE {
  lexicon {
    daysRented [0-9]+
    reserved new | reg | child
    name [A-Z][A-Za-z0-9_]*
    title [a-z][a-z0-9_]*
    ignore [\ \0x0A\0x0D\0x09]+
  }

  attributes
    Hashtable *.outMS,*.inMS;
    Hashtable *.outCS,*.inCS;
    Price *.type;
    String *.name;
    String *.title;
    int *.daysRented;
    double *.income;

  rule Store {
    VIDEO_STORE ::= MOVIES CUSTOMERS compute {
```

```

        VIDEO_STORE.outMS = MOVIES.outMS;
        VIDEO_STORE.outCS = CUSTOMERS.outCS;
        VIDEO_STORE.income = CUSTOMERS.income;

        MOVIES.inMS      = new Hashtable();

        CUSTOMERS.inMS   = MOVIES.outMS;
        CUSTOMERS.inCS   = new Hashtable();
    };
}

rule Movies {
    MOVIES ::= MOVIES MOVIE compute {
        MOVIES[0].outMS = insert(MOVIES[1].outMS,
            new Movie(MOVIE.title, MOVIE.type));

        MOVIES[1].inMS = MOVIES[0].inMS;
    }
    | epsilon compute {
        MOVIES.outMS = MOVIES.inMS;
    };
}

rule Movie {
    MOVIE ::= #title PRICE compute {
        MOVIE.title = #title.value();
        MOVIE.type = PRICE.type;
    };
}

rule Customers {
    CUSTOMERS ::= CUSTOMERS CUSTOMER compute {
        CUSTOMERS[0].outCS = CUSTOMER.outCS;
        CUSTOMERS[0].income = CUSTOMERS[1].income +
            CUSTOMER.income;

        CUSTOMERS[1].inMS = CUSTOMERS[0].inMS;
        CUSTOMERS[1].inCS = CUSTOMERS[0].inCS;

        CUSTOMER.inMS = CUSTOMERS[0].inMS;
    };
}

```

```

        CUSTOMER.inCS      = CUSTOMERS[1].outCS;
    }
    | epsilon compute {
        CUSTOMERS.outCS = CUSTOMERS.inCS;
        CUSTOMERS.income = 0.0;
    };
}

rule Customer {
    CUSTOMER ::= #name RENTALS compute {
        CUSTOMER.outCS = RENTALS.outCS;
        CUSTOMER.income = RENTALS.income;
        CUSTOMER.name = #name.value();

        RENTALS.inMS = CUSTOMER.inMS;
        RENTALS.inCS = insert(CUSTOMER.outCS,
            new Customer(CUSTOMER.name));
        RENTALS.name = CUSTOMER.name;
    };
}

rule Rentals {
    RENTALS ::= RENTALS RENTAL compute {
        RENTALS[0].outCS = addRental(
            RENTALS[1].outCS, getCustomer(
                RENTALS[1].outCS, RENTALS[0].name),
            new Rental( getMovie( RENTALS[0].inMS,
                RENTAL.title), RENTAL.daysRented));
        RENTALS[0].income = RENTALS[1].income +
            getCharge(getMovie(
                RENTALS[0].inMS, RENTAL.title),
                RENTAL.daysRented);

        RENTALS[1].inMS = RENTALS[0].inMS;
        RENTALS[1].inCS = RENTALS[0].inCS;
        RENTALS[1].name = RENTALS[0].name;
    }
    | epsilon compute {
        RENTALS.outCS = RENTALS.inCS;
        RENTALS.income = 0.0;
    }
}

```

```

    };
}

rule Rental {
    RENTAL ::= #daysRented MOVIE compute {
        RENTAL.title      = MOVIE.title;
        RENTAL.daysRented = Integer.
            valueOf( #daysRented.value()).intValue();
    };
}

rule Price {
    PRICE ::= new compute {
        PRICE.type = new New();
    }
    | reg compute {
        PRICE.type = new Reg();
    }
    | child compute {
        PRICE.type = new Child();
    };
}

method M_Movie {
    class Movie {
        String name;
        Price type;
        Movie ( String name, Price type) {
            this.name = name;
            this.type = type;
        }

        public String toString(){
            return "(" + this.name + ", " + this.type + ")";
        }

        public String getTitle(){
            return this.name;
        }
    }
}

```

```

        public Price getType(){
            return this.type;
        }
    }
}

method M_Insert {
    import java.util.*;
    Hashtable insert (Hashtable aMovies, Movie aMovie) {
        Movie hMovie=(Movie)aMovies.get(aMovie.getTitle());
        if (hMovie==null)
            aMovies.put(aMovie.getTitle(), aMovie);
        else
            System.out.println("Item" + aMovie.getTitle() +
                "is already in video store");
        return aMovies;
    } // java method
} // Lisa method

method C_Item {
    class Customer {
        String name;
        Hashtable rentals;
        Customer ( String name) {
            this.name = name;
            this.rentals = new Hashtable();
        }

        public String toString(){
            return "(" + this.name + ", " +
                rentals.toString() + ")";
        }

        public String getName(){
            return this.name;
        }

        public Rental getRental(String aTitle){
            return (Rental)this.rentals.get(aTitle);
        }
    }
}

```

```

    public Hashtable getRentals(){
        return this.rentals;
    }

    public void putRental(Rental aRental){
        this.rentals.put(aRental.getMovie().getTitle(),
            aRental);
    }
}

method C_Insert {
    import java.util.*;
    Hashtable insert (Hashtable aCustomers,
        Customer aCustomer) {
        Customer hCustomer=(Customer)aCustomers.get(
            aCustomer.getName());
        if (hCustomer==null){
            aCustomers.put(aCustomer.getName(), aCustomer);
        }
        else
            System.out.println("Item" + aCustomer.getName() +
                "is already in video store");
        return aCustomers;
    } // java method
} // Lisa method

method R_Item {
    class Rental {
        Movie movie;
        int days;
        Rental (Movie movie, int days) {
            this.movie = movie;
            this.days = days;
        }

        public String toString() {
            return "(" + this.movie.toString() + ", " +
                this.days + ")";
        }
    }
}

```

```

    }

    public Movie getMovie() {
        return this.movie;
    }

    public int getDays() {
        return this.days;
    }
}

method R_Insert {
    import java.util.*;
    Hashtable addRental (Hashtable aCustomers,
    Customer aCustomer, Rental aRental) {
        aCustomer.putRental(aRental);
        aCustomers.remove(aCustomer.getName());
        aCustomers.put(aCustomer.getName(),aCustomer);
        return aCustomers;
    } // java method
} // Lisa method

method R_getCustomer {
    import java.util.*;
    Customer getCustomer(Hashtable aCustomers, String aName) {
        Customer aCustomer=(Customer)aCustomers.get(aName);
        System.out.println( "Customer " +
            aCustomer.toString() );

        return aCustomer;
    } // java method
} // Lisa method

method R_getMovie {
    import java.util.*;
    Movie getMovie (Hashtable aMovies, String aTitle) {
        Movie aMovie=(Movie)aMovies.get(aTitle);
        if (aMovie==null){
            System.out.println( "Item " + aTitle +
                "doesn't exist!" );
        }
    }
}

```



```

        return null;
    }
    else
        return aMovie;
} // java method
} // Lisa method

method R_getPrice {
import java.util.*;
double getNewPric (int daysRented) {
    double result =2;
    if (daysRented > 2 )
        result += (daysRented -2) * 1.5;
    return result;
} // java method

double getRegularPrice (int daysRented) {
    return daysRented * 1.5;
} // java method

double getChildrenPrice (int daysRented) {
    double result =1.5;
    if (daysRented > 3 )
        result += (daysRented - 3) * 1.5;
    return result;
} // java method
} // Lisa method

method M_Price {
import java.util.*;
abstract class Price {
    abstract double getPrice(int daysRented);
}

class New extends Price{
    New() {}
    double getPrice (int daysRented) {
        double result =2;
        if (daysRented > 2 )
            result += (daysRented -2) * 1.5;
    }
}

```

```

        return result;
    }

    public String toString(){
        return "new";
    }
}

class Reg extends Price{
    Reg(){

    }

    double getPrice (int daysRented) {
        return daysRented * 1.5;
    } // java method

    public String toString(){
        return "reg";
    }
}

class Child extends Price{
    Child(){

    }
    double getPrice (int daysRented) {
        double result =1.5;
        if (daysRented > 3 )
            result += (daysRented - 3) * 1.5;
        return result;
    } // java method

    public String toString(){
        return "child";
    }
} // java method
} // Lisa method

method R_charge {
    double getCharge(Movie aMovie, int daysRented){
        return aMovie.getType().getPrice(daysRented);
    }
} // Lisa method

```

```
} // Language
```

To represent the functionality of LISA specifications, the following program is shown below.

```
jurassic_park child
road_trip reg
the_ring new
Andy 3 jurassic_park child 2 road_trip reg
Mary 3 the_ring new
```

The meaning of the above program is the following movie table (attribute outMS), customer table (attribute outCS) and money income (attribute income).

```
outMS:{
  jurassic_park={Jurassic_park, child},
  road_trip={road_trip, reg},
  the_ring={the_ring, new}}
outCS:{Mary=(Mary,{the_ring=(
  (the_ring,new),3)}),
  Andy=(Andy,{road_trip=(
  (road_trip,reg),3)},
  jurassic_park=(
  (jurassic_park,child),2))}}
income:8.0
```

Note that for the same scenario the following Java program has to be executed, which is much more verbose and less intuitive for the end-user:

```
public static void main(String[] args){
  double income = 0.0;
  Movie m1 = new Movie(
    "jurassic_park", Movie.CHILDRENS);
  Movie m2 = new Movie(
    "road_trip", Movie.REGULAR);
  Movie m3 = new Movie(
    "the_ring", Movie.NEW_RELEASE);
  Customer c1 = new Customer("Andy");
  Customer c2 = new Customer("Mary");
```

```

Rental r1 = new Rental(m1, 3);
Rental r2 = new Rental(m2, 2);
Rental r3 = new Rental(m3, 3);
c1.addRental(r1);
c1.addRental(r2);
c2.addRental(r3);

income += c1.evaluateCharge();
income += c2.evaluateCharge();
}

```

#### 6.2.4 CS4: Cleaning Robot

As shown in previous case studies, the translation from attribute grammar specifications of cleaning robot case study to LISA specifications are straightforward.

```

language CleaningRobot {
  lexicon {
    Int      [0-9]+
    switch   on | off
    keywords left | right | up | down | stop
    ignore  [\x0D\x0A\ ]
  }

  attributes
    Room *.inCleanPts, *.outCleanPts;
    Point *.inPos, *.outPos,
          *.inBounds, *.outBounds;
    boolean CLEANSYS.clean;
    int DIRECTIONS.direct, STEPS.steps,
        DIMENSIONS.x, DIMENSIONS.y;

  rule start {
    CLEANTASK ::= DIMENSIONS MOVES compute {
      CLEANTASK.outCleanPts = MOVES.outCleanPts;
      CLEANTASK.outPos      = MOVES.outPos;
      CLEANTASK.outBounds   = DIMENSIONS.outBounds;

      MOVES.inCleanPts = new Room(DIMENSIONS.outBounds.x,

```

```

                                DIMENSIONS.outBounds.y);
MOVES.inPos      = new Point(0, 0);
MOVES.inBounds   = DIMENSIONS.outBounds;
};
}

rule dimensions {
  DIMENSIONS ::= #Int #Int compute {
    DIMENSIONS.x = Integer.valueOf(#Int.value()).
                    intValue();
    DIMENSIONS.y = Integer.valueOf(#Int.value()).
                    intValue();
    DIMENSIONS.outBounds = new Point(DIMENSIONS.x,
                                      DIMENSIONS.y);
  };
}

rule moves {
  MOVES ::= MOVES MOVE compute {
    MOVES[0].outCleanPts = MOVE.outCleanPts;
    MOVES[0].outPos      = MOVE.outPos;

    MOVES[1].inCleanPts = MOVES[0].inCleanPts;
    MOVES[1].inPos      = MOVES[0].inPos;
    MOVES[1].inBounds   = MOVES[0].inBounds;

    MOVE.inCleanPts = MOVES[1].outCleanPts;
    MOVE.inPos      = MOVES[1].outPos;
    MOVE.inBounds   = MOVES[0].inBounds;
  }
  | MOVE compute {
    MOVES.outCleanPts = MOVE.outCleanPts;
    MOVES.outPos      = MOVE.outPos;

    MOVE.inCleanPts = MOVES.inCleanPts;
    MOVE.inPos      = MOVES.inPos;
    MOVE.inBounds   = MOVES.inBounds;
  };
}
}

```

```

rule move {
  MOVE ::= DIRECTIONS STEPS CLEANSYSTEM compute {
    MOVE.outCleanPts = updateTab(MOVE.inCleanPts, MOVE.inPos,
      MOVE.inBounds, DIRECTIONS.direct, STEPS.steps,
      CLEANSYSEM.clean);
    MOVE.outPos      = updatePos(MOVE.inPos, MOVE.inBounds,
      DIRECTIONS.direct, STEPS.steps);
  }
  | stop compute {
    MOVE.outCleanPts = MOVE.inCleanPts;
    MOVE.outPos      = MOVE.inPos;
  };
}

rule directions {
  DIRECTIONS ::= left compute {
    DIRECTIONS.direct = LEFT;
  }
  | right compute {
    DIRECTIONS.direct = RIGHT;
  }
  | up compute {
    DIRECTIONS.direct = UP;
  }
  | down compute {
    DIRECTIONS.direct = DOWN;
  };
}

rule steps {
  STEPS ::= #Int compute {
    STEPS.steps = Integer.valueOf(#Int.value()).
      intValue();
  }
  | epsilon compute {
    STEPS.steps = 1; // default step
  };
}

rule clean {

```

```

CLEANSYSTEM ::= on compute {
    CLEANSYSTEM.clean = true;
}
| off compute {
    CLEANSYSTEM.clean = false;
}
| epsilon compute {
    CLEANSYSTEM.clean = true; // default cleaning mode
};
}

method M_Types {
    public static final int UP = 0, DOWN = 1, LEFT = 2,
                        RIGHT = 3;

    class Point {
        int x;
        int y;

        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        Point(Point p) {
            this.x = p.x;
            this.y = p.y;
        }

        public String toString() {
            return "(" + x + "," + y + ")";
        }
    } // class

    class Room {
        int[][] room;
        Room(int x, int y) {
            room = new int[x][y];
            // robot start on (0, 0) with clean switch ON
            room[0][0]=1;
        }
    }
}

```

```

public void clean(int x, int y) {
    room[x][y]=1;
}

public String toString() {
    String text="|";
    int x,y;
    for (y=0; y < room.length; y++) {
        for (x=0; x < room[y].length; x++)
            text+=room[x][y]+" ";
        text+=" | ";
    }
    return text;
}
} // class
} // Lisa method

method M_Calc {
    Point updatePos(Point in, Point bounds,
                    int dir, int steps) {
        Point out = new Point(in);
        int i;
        for (i=0; i < steps; i++) {
            switch (dir) {
                case UP:    if (out.y < bounds.y - 1) {
                            out.y = out.y + 1;
                        }
                            break;
                case DOWN:  if (out.y > 0) {
                            out.y = out.y - 1;
                        }
                            break;
                case RIGHT: if (out.x < bounds.x - 1) {
                            out.x = out.x + 1;
                        }
                            break;
                case LEFT:  if (out.x > 0) {
                            out.x = out.x - 1;
                        }
            }
        }
    }
}

```



```

        break;
    }
}
return out;
} // java method

Room updateTab(Room tab, Point in, Point bounds,
               int dir, int steps, boolean clean) {
    Point out = new Point(in);
    int i;
    for (i=0; i < steps; i++) {
        switch (dir) {
            case UP:    if (out.y < bounds.y - 1) {
                        out.y = out.y + 1;
                    }
                        break;
            case DOWN:  if (out.y > 0) {
                        out.y = out.y - 1;
                    }
                        break;
            case RIGHT: if (out.x < bounds.x - 1) {
                        out.x = out.x + 1;
                    }
                        break;
            case LEFT:  if (out.x > 0) {
                        out.x = out.x - 1;
                    }
                        break;
        }
        if (clean) tab.clean(out.x, out.y);
    }
    return tab;
} // java method
} // Lisa method

} // language CleaningRobot

```

To show functionalities of above specifications, following programs are introduced.

Example No. 1:

3 3 right 2 on up 2 on left 2 off down 1 on

The meaning is:

outCleanPts: | 1 1 1 | 1 0 1 | 0 0 1 |

The table is read as:

0 0 1  
1 0 1  
1 1 1

and position (0, 1).

Example No. 2:

3 3 right up left down

The meaning is:

outCleanPts: | 1 1 0 | 1 1 0 | 0 0 0 |  
outPos: (,)

and position (3, 3).

The table is read as:

0 0 0  
1 1 0  
1 1 0

Example No. 3:

3 3 right 2 up 2 off left 1 on stop up 1

The meaning is:

outCleanPts: | 1 1 1 | 0 0 0 | 0 1 0 |

The table is read as:

0 1 0  
0 0 0  
1 1 1

with the final position of robot (1, 2).

## 7 Conclusion

The main contribution of this document is the proposal of a new method to specify complex tasks in the context of computer problem solving.

The proposed method is based on a grammatical specification (attribute grammar) written in an object-oriented style (OOAG). UML diagrams are also used in the analysis phase to help in the design of the context free grammar (the first step of this method), as well as in the design of the attribute grammar (the second step of the method). In this last phase, the UML diagrams can be helpful in the choice of the attributes, their evaluation rules and the necessary contextual conditions. This grammatical specification so far obtained allows us to use a rapid prototype technique (the automatic generation of a language processor, a compiler, to interpret sentences of the new language) to obtain a simulator for the problem under study. Then, we can test the system and validate its behavior.

The main idea of the proposed method is to follow the well-known syntactic/semantic approach<sup>6</sup> based on context-free grammars (that defines the structure, or syntax) and attribute grammars (that specifies the semantics). This is a secure way to get a formal specification from which we produce a rapid prototype using compiler generators.

It is also possible to split the semantic specification into several grammars, where each one will be used to describe a different task. The grammars all together will represent the semantics for the problem solving.

As told above, this oo-grammatical approach can be improved adding a previous phase to map the concepts described in UML diagrams into grammars (structure and attributes).

Since our approach is object-oriented, LISA System is an adequate tool to get the desired prototype. LISA System is modular, uses object-oriented grammars and implements multiple attribute grammar inheritance. To prove the usability of our approach, four complete different examples were shown in this report; for each one, we state the problem, design the UML use-cases, class and operational diagrams, derive the context free grammar, and write the attribute grammar. Then, using LISA, we generate a processor, and check the specification verifying the system behavior when faced with different input sequences.

---

<sup>6</sup>That typically supports the development of grammars and influences the language processing strategy.

## References

- [Ado02] S. Adolph. *Patterns for Effective Use Cases*. Addison-Wesley, 2002.
- [Ars01] Ali Arsanjani. Grammar-oriented object design: Creating adaptive collaborations and dynamic configurations with self-describing components and services. In *Proceedings of TOOLS 2001*, volume 65. IEEE Computer Society Press, 2001.
- [BL02] Barrett Bryant and Beum-Seuk Lee. Two-level grammar as an object-oriented requirements specification language. In *IEEE CD ROM Proceedings of 35th Hawaii International Conference on System Sciences*, 2002.
- [Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.
- [Coc01] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.
- [Fow97] M. Fowler. *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley Longman, 1997.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [HPM<sup>+</sup>02] P. Henriques, M. V. Pereira, M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Automatic generation of language-based tools. In Mark van den Brand and Ralf Laemmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.
- [Knu68] D. Knuth. Semantics of contex-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.
- [LA02] Keith Levi and Ali Arsanjani. A goal-driven approach to enterprise component identification and specification. *Communications of the ACM*, 45(10):45–52, October 2002.
- [LBNE00] Luqi, V. Berzins, N. Nada, and C. Eagle. Computer aided prototyping system (CAPS) for heterogeneous systems development

- and integration. In *Proceedings of the 2000 Command and Control Research and Technology Symposium, Naval Postgraduate School, Monterey, CA, June 26-28, 2000*.
- [ME00] M. Mernik and D. Parigot (Eds.). Attribute grammars and their applications. *Informatica*, 24(3), September 2000.
- [MLAŽ99] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA '99*, pages 57–76, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.
- [MLAŽ00] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Compiler/interpreter generator system LISA. In *IEEE CD ROM Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.
- [MLAŽ02] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An Interactive Environment for Programming Language Development. In Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 1–4. Lecture Notes in Computer Science, Springer-Verlag, 2002.
- [MLEŽ00] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, September 2000.
- [Paa95] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.
- [RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [vDM02] A. van Deursen and Leon Moonen. The video store revisited thoughts on refactoring and testing. In *Proceedings of the 3d International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002), University of Cagliari, 2002*, pages 71–76, 2002.

- [Wai90] W.M. Waite. Use of attribute grammars in compiler construction. In M. Jourdan P. Deransart, editor, *Proceedings of 1st WAGA, Lecture Notes in Computer Science Vol. 461, Springer - Verlag*, pages 255–265, 1990.

## A Meta-Grammar for OOAG Descriptions

```
OOAGSpecf          -> OOAGComposition OOAGDescription
OOAGComposition    -> agId '=' AgLst
AgLst              -> agId
                  | AgLst '+' agId
OOAGDescription    -> SimpleOOAG
                  | OOAGDescription SimpleOOAG
SimpleOOAG         -> 'OOAG:' agId SymbDescriptions
SymbDescriptions   -> SymbDesc SymbDescriptions
                  | SymbDesc
SymbDesc           -> NonTermDesc
                  | TermDesc
TermDesc           -> 'Term' simbid ':'
                  | 'Syn' ':' '{' SyntesizedATT '}'
NonTermDesc        -> 'NonTerm' simbid ':'
                  | 'Inh' ':' '{' InheritATT '}'
                  | 'Syn' ':' '{' SyntesizedATT '}'
RegExpDescs        -> RegExpDesc
                  | RegExpDescs RegExpDesc
RegExpDesc         -> regId '(' regExp ')'
InheritATT         -> Decls
SyntesizedATT     -> Decls
Decl               -> &
                  | Decls Decl
Decl               -> attId ':' typeId
ProductionDescs   -> ProdDesc
                  | ProductionDescs ProdDesc
ProdDesc           -> prodId '(' Prod ')' ':'
                  | EvalRules ContextConds
Prod               -> simbid '->' Simbs
Simbs              -> &
                  | Simbs simbid
EvalRules          -> &
                  | EvalRules EvalRule
EvalRule           -> Att '=' Exp
Att                -> simbid '.' attId
                  | simbid '/' num '.' attId
```

```
ContextConds    -> &
                 | ``CC:'' boolexp
Exp              -> SimpleExp
                 | Exp op SimpleExp
SimpleExp        -> Function
                 | Att
                 | constant
Function         -> funId '(' AttLst ')'
AttLst           -> &
                 | AttLst Att
```