

Software Development with Grammatical Approach

Tomaž Kosar, Marjan Mernik and Viljem Žumer

University of Maribor, Faculty of Electrical Engineering and Computer Science, Slovenia

E-mail: {tomaz.kosar, marjan.mernik, zumer}@uni-mb.si

Pedro Rangel Henriques

University of Minho, Department of Computer Science, Portugal

E-mail: prh@di.uminho.pt

Maria João Varanda Pereira

Polytechnic Institute of Bragança, Portugal

E-mail: mjoao@ipb.pt

Keywords: software design and modelling, software development, context-free grammars, attribute grammars, rapid prototyping

Received: July 12, 2004

The paper presents a grammatical approach to software development. It supports formal software specification using attribute grammars, from which a rapid prototype can be generated, as well as the incremental software development. Domain concepts and relationships among them have to be identified from a problem statement and represented as a context-free grammar. The obtained context-free grammar describes the syntax of a domain-specific language whose semantics is the same as the functionality of the system under implementation. The semantics of this language is then described using attribute grammars from which a compiler is automatically generated. The execution of a particular program written in that domain-specific language corresponds to the execution of a prototype of the system on a particular use case.

Povzetek: članek opisuje razvoj programov na osnovi slovnice.

1 Introduction

One of the well known properties of software systems is that they are subject to frequent changes. A software developer needs to build a software system in such a manner that he can easily adapt it to the user's changeable requirements. Current object-oriented design techniques [7] [8] are well suited for such design supporting changes. However, any changes during the software life cycle are costly. Therefore, it is very important that the user is involved in the software development process from the very beginning and that the software system is delivered to the user before his requirements have the opportunity to change. Rapid prototyping enables the software developer to build executable prototypes and to involve the user in an iterative build-execute-modify loop until his requirements are validated. The prototype is then used to build the final version of the software system through the use of the architecture included in the prototype or it is simply thrown away [21]. In the latter case the prototype is used to clarify the user's needs until reaching a stable and convenient model for the given problem.

The proposed approach, i.e. *software development with grammatical approach*, rests on the success reached by attribute grammars in the specification of language semantics

[12] [6] [16] and in the systematic implementation of language processing tools [9] [10].

In the paper the grammatical approach to problem solving supported by an attribute grammar developed and written in an object-oriented style (OOAG - object-oriented attribute grammar) is proposed. One of the benefits of the proposed approach is that it enables rapid prototyping and the validation of the user's requirements in a pragmatic way. The idea is to translate the OOAG obtained in the 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 domain-specific language [17] [22] [24] defined by that OOAG) describing different uses of the system, and use the generated simulator to process those scenarios computing the desired results.

The organization of the paper is as follows. In Section 2 related work is discussed. The software development with grammatical approach is presented in detail in Section 3 followed by an example in the Section 4. A synthesis and concluding remarks are presented in Section 5.

2 Related Work

The grammatical approach to problem solving (software development) can be seen as an extension (e.g. as in [15]) of object-oriented design methods [20] [7] [8] 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) [2] [14], 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 modeled 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 [2] [14] 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 [2] [14] 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. [4]). In [4] 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 Environment (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.

Resolving the semantical gap between use case diagram and class diagram is also presented in [19]. From the use case diagram agents state machines and values added invariants are derived. The term agent is used to represent an actor collaborating with the system through specific use case. Both techniques are collectively used in iterative converting algorithm, which builds the OCL specification and class diagram. The OCL specification (define a set of preconditions, postconditions and actor invariants) are further used to check the correctness of the model.

3 The Grammatical Approach

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 (UCDs) and conceptual class diagrams (CCDs) [7] which we will take as a starting point for our approach.

The use case diagram [5][1] describes the functionality of the system and its interaction with an environment. The use case diagrams form foundations for further modelling of developing system. They are also helpful for generating system test cases.

While use case diagrams 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 [20]. 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 design model is obtained which should be robust with respect to changes of the user's requirements.

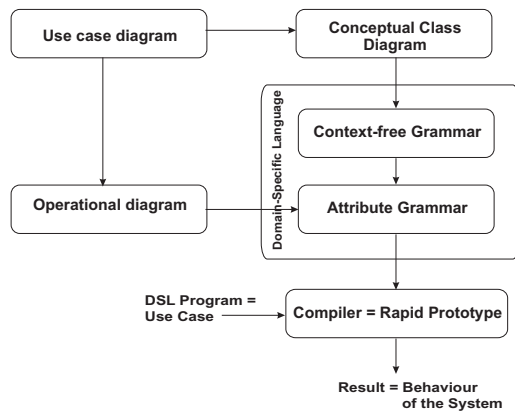


Figure 1: High-level view of the grammatical approach

To identify concepts and their relationships in 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 [3] [8] is now almost the-facto standard for software system design, and on account of that, it was also our choice.

Our approach (described in Fig. 1) is based on the following steps:

- describe the syntax of the problem (the structure of the classes that characterise 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 steps above will be detailed in the next subsections.

3.1 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 table 1 and table 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 table 2. For generalization we propose the production $A \rightarrow B | C$. 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 .

Classes can collaborate with more than just one class. For example, a class A associates with classes B , C and D . In our approach, this collaboration is described with context-free grammar 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.

3.2 Describing the semantics of each concept

To describe the semantics or the meaning of a concept an attribute grammar is used. Attribute grammars [12] [6] [16] 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

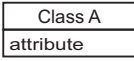


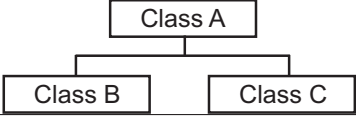


Description	Class diagram element	Grammar
Class		A (non-terminal) instance variable (terminal)
Association		$A \rightarrow B$
Navigability		$A \rightarrow B$
Generalization		$A \rightarrow B \mid C$
Composition		$A \rightarrow B$ $(\neg \exists X \in N, X \Rightarrow B)$ $\wedge X \neq A$
Aggregation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

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



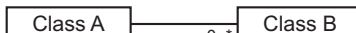

Cardinality	Class diagram element	Grammar
Multiplicity exactly one		$A \rightarrow B$
Optional multiplicity		$A \rightarrow B \mid \epsilon$
Multiplicity [0..m]		$A \rightarrow \text{MoreB}$ $\text{MoreB} \rightarrow \text{MoreB B} \mid \epsilon$
Multiplicity many		$A \rightarrow \text{MoreB}$ $\text{MoreB}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

Table 2: Association multiplicity

the other hand, it is implicit in 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). 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.

3.3 Generating the rapid prototype of a system

To generate the rapid prototype of a system our compiler-generator LISA [18] 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 [10]—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.

4 An Example: Video Store

The Video Store (VS) case study is one of the basic examples of the refactoring [7][23]. The case study represents

a prototype program for customer charges at a video store. The program calculates the charge, which depends on how long a 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.

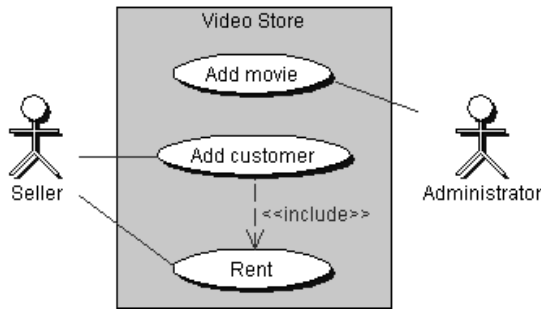


Figure 2: Use case diagram

For the case study of the Video Store we identify three main services represented with use cases *Add movie*, *Add customer* and *Rent* (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 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.

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 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.

The Conceptual Class Diagram: The use case diagram (Fig. 2) is crucial to find the basic entities and to derive the conceptual class diagram. There are no specific rules to support this derivation, but you can find many guidelines in [11][13].

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

As shown on figure 3, the *VideoStore* is identified as the main concept. The two other important concepts in the management of the *VideoStore* are: *Customer*, and *Movie*. *Movie* associates with class *Price* which describes the type of a movie. Generalization class *Price* is further implemented with classes *New*, *Child*, and *Reg*. The data for each rental are kept in class *Rental*.

The Structure: Remember that, in our approach, a problem concept is denoted by a grammar symbol. The context-free grammar below formalizes the problem syntax in the sense that it specifies the structure of the problem domain, relating the concepts among them. The following context-free grammar is obtained using transformations described in Section 3. To be able to read context-free grammar see the transformation table 1 and table 2.

```

VIDEO_STORE -> MOVIES CUSTOMERS
MOVIES      -> MOVIES MOVIE
            | &
MOVIE       -> title PRICE
CUSTOMERS   -> CUSTOMERS CUSTOMER
            | &
CUSTOMER    -> name RENTALS
RENTALS     -> RENTALS RENTAL
            | &
RENTAL      -> daysRented MOVIE
PRICE       -> new | child | reg

```

To follow the transformations from table 1 abstract class *Price* defines non-terminal *PRICE* and its subclasses (Fig. 3) define non-terminals in the production

```
PRICE -> NEW | CHILD | REG
```

Unfortunately, this subclasses have no terminals and represent the last classes in every traverse through the conceptual class diagram. Described classes are named final classes. Each non-terminal of final class can be replaced with terminal in productions (see the above context-free grammar).

It may happen, that deriving context-free grammar from a conceptual class diagram through transformations in table 1 and table 2 does not show an optimal grammar. Such grammar can have useless non-terminals, which can be reduced. Try to imagine the video store example as stated above, except the rental service changes a bit. Now, the

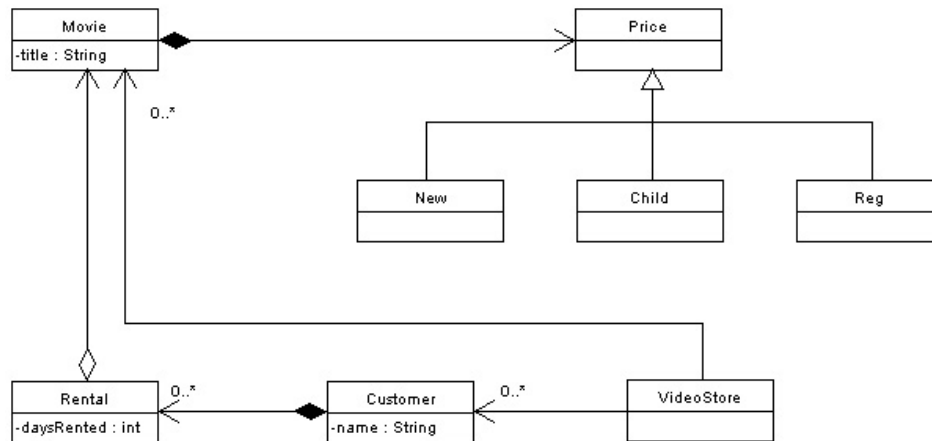


Figure 3: Conceptual Class Diagram for Video Store

rental length for all movies of one customer is the same (in our example the rental length is defined separately for each movie).

```

CUSTOMER    -> name daysRented RENTALS
RENTALS     -> RENTALS RENTAL
            |
            &
RENTAL      -> MOVIE
MOVIE       -> title PRICE
...
    
```

In the partial context-free grammar we have useless production for non-terminal MOVIE. The production, that have just one non-terminal and no terminal on right side, can be rearranged or even removed (e.g. obtaining just the context-free grammar production `RENTAL -> title PRICE`).

Removing the non-terminal from the context-free grammar brings the question, if class is reasonable in the conceptual class diagram at all. We believe that, if there is no other association with this conceptual class, the class can be removed. Looking from this prospective, building context-free grammar can help in evolving the optimized conceptual class diagram.

Semantics (1. 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 Section 3.2. 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) has been used.

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

Information 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. Explanation follows later in the paper.

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. 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.

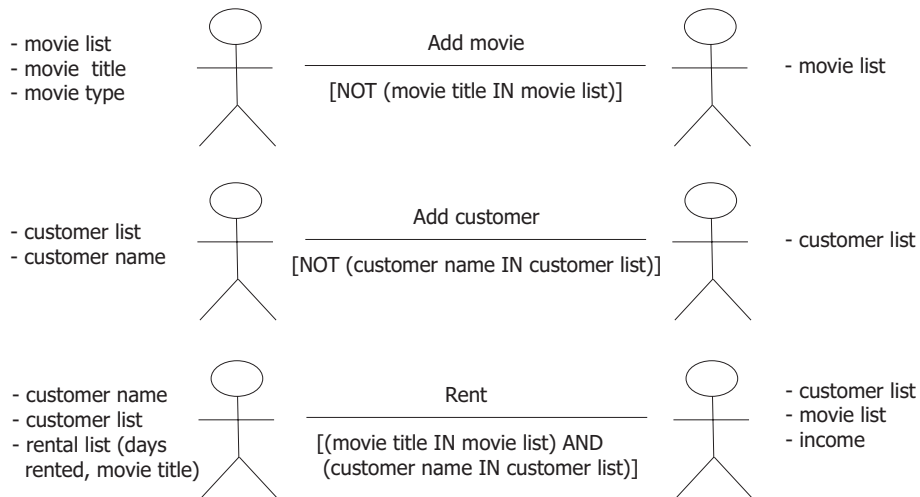


Figure 4: The operational diagram

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

Table 3: Attributes mapping to nonterminals

Attribute	Starting non-terminal
outCS	Video_Store
outMS	Video_Store
income	Video_Store

Table 4: Attributes in starting non-terminal Video_Store

The right side attributes from operational diagrams are important to find 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 4). 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 should each attribute be carried and with what purpose.

The alternatives in use case scenarios are basics to find the contextual conditions. The contextual conditions are

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

Table 5: Attributes in other nonterminals

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.

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

The first production is VIDEO_STORE \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. Both attributes are of type TAB, which is a mapping function.

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

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

mkVideoStore(VIDEO_STORE ->
              MOVIES CUSTOMERS):
  VIDEO_STORE.outMS = MOVIES.outMS
  MOVIES.inMS = {}
  VIDEO_STORE.outCS = CUSTOMERS.outCS
  CUSTOMERS.inCS = {}
  CUSTOMERS.inMS = MOVIES.outMS
  VIDEO_STORE.income = CUSTOMERS.income
```

For collecting the elements of entity Movie, we use non-terminals MOVIES and MOVIE (see Section 3). The semantic of the non-terminal is described with attributes inMS and outMS, 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).

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

mkMovies(MOVIES -> MOVIES MOVIE):
  MOVIES/1.inMS = MOVIES/0.inMS
  MOVIES/0.outMS =
    insert(MOVIES/1.outMS,
           new Movie(MOVIE.title,
                     MOVIE.type))
  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.

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

mkCustomers(CUSTOMERS ->
            CUSTOMERS CUSTOMER):
  CUSTOMERS/1.inCS =
    CUSTOMERS/0.inCS
  CUSTOMERS/0.outCS =
    CUSTOMER.outCS
  CUSTOMER.inMS =
    CUSTOMERS/0.inMS
  CUSTOMERS/1.inMS =
    CUSTOMERS/0.inMS
  CUSTOMER.inCS =
    CUSTOMERS/1.outCS;
  CUSTOMERS/0.income =
    CUSTOMERS/1.income +
    CUSTOMER.income
  CC: (NOT(CUSTOMER.name IN
           CUSTOMERS/1.outCS))

emptyCustomers(CUSTOMERS -> &):
  CUSTOMERS.outCS = CUSTOMERS.inCS;
  CUSTOMER.income = 0.0
```

Semantics constructs of non-terminal CUSTOMER consist of attributes name (String type), 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.name = name.lexval
  RENTALS.name = CUSTOMER.name
  CUSTOMER.outCS = RENTALS.outCS
  RENTALS.inCS = insert(
    CUSTOMER.inCS,
    new Customer(CUSTOMER.name))
  RENTALS.inMS = CUSTOMER.inMS
  CUSTOMER.income = RENTALS.income
```


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.

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

```
NonTerm RENTALS:
```

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

```
mkRentals(RENTALS -> RENTALS RENTAL):
  RENTALS/1.inCS = RENTALS/0.inCS
  RENTALS/1.inMS = RENTALS/0.inMS
  RENTALS/1.name = RENTALS/0.name
  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)
CC: ((RENTAL.title IN RENTALS/0.inMS)
     AND (RENTALS.name IN
          RENTALS/0.inCS))
```

```
emptyRentals (RENTALS -> &):
  RENTALS.outCS = RENTALS.inCS
  RENTALS.income = 0.0
```

As shown above, for mapping the rental items to customer, the function addRentals is defined. 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 (see Section 4), 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()
```

As described in above specifications, attribute evaluation is derived through semantic functions. Functions open the next question. Can this functions help to derive information to obtain methods in class diagram (fig. 5). In that case, the part of prototype could be reused in developing the complete system. This part of our approach is under investigation.

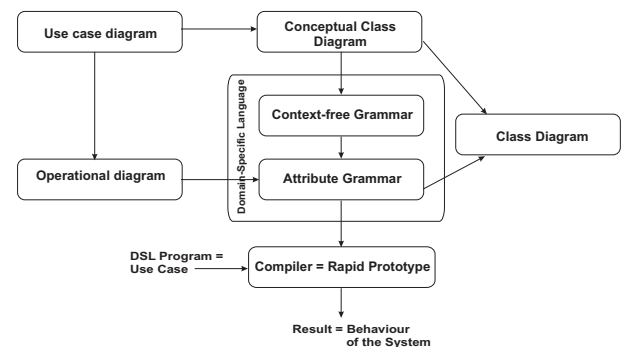


Figure 5: Developing class methods from functions

The rapid prototype: The attribute grammar specified in the previous step is then written using our compiler generator system LISA. The inherent modularity of attribute grammars enables iterative design of prototype. Therefore, more functionalities of a system can be implemented.

A part of these specifications is shown below. Note the straightforward translation from above specifications to LISA. Notice that, the contextual conditions are not shown below. They are implemented with functions which appear in the LISA method part.

```
language VIDEO_STORE {
  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;
rule Store {
  VIDEO_STORE ::= MOVIES CUSTOMERS
  compute {
    VIDEO_STORE.outMS = MOVIES.outMS;
    MOVIES.inMS = new Hashtable();
    VIDEO_STORE.outCS =
      CUSTOMERS.outCS;
    CUSTOMERS.inCS = new Hashtable();
    CUSTOMERS.inMS = MOVIES.outMS;
    VIDEO_STORE.income =
      CUSTOMERS.income;
  };
}
rule Movies {
  MOVIES ::= MOVIES MOVIE compute {
    MOVIES[1].inMS = MOVIES[0].inMS;
    MOVIES[0].outMS = insert(
      MOVIES[1].outMS,
      new Movie(MOVIE.title,
        MOVIE.type));
  }
  | 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[1].inCS =
      CUSTOMERS[0].inCS;
    CUSTOMERS[0].outCS =
      CUSTOMER.outCS;
    CUSTOMER.inMS =
      CUSTOMERS[0].inMS;
    CUSTOMERS[1].inMS =
      CUSTOMERS[0].inMS;
    CUSTOMER.inCS =
      CUSTOMERS[1].outCS;
    CUSTOMERS[0].income =
      CUSTOMERS[1].income +
      CUSTOMER.income;
  }
  | epsilon compute {
    CUSTOMERS.outCS = CUSTOMERS.inCS;
    CUSTOMERS.income = 0.0;
  };
}
rule Customer {
  CUSTOMER ::= #name RENTALS compute {

```

```

    CUSTOMER.name = #name.value();
    RENTALS.name = CUSTOMER.name;
    CUSTOMER.outCS = RENTALS.outCS;
    RENTALS.inCS =
      insert(CUSTOMER.inCS,
        new Customer(CUSTOMER.name));
    RENTALS.inMS = CUSTOMER.inMS;
    CUSTOMER.income = RENTALS.income;
  };
}
rule Rentals {
  RENTALS ::= RENTALS RENTAL compute {
    RENTALS[1].inCS = RENTALS[0].inCS;
    RENTALS[1].inMS = RENTALS[0].inMS;
    RENTALS[1].name = RENTALS[0].name;
    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);
  }
  | 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 part
} //Language

```

One of the possible scenarios is now described with the following program:

```

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)}, 3.5),
  Andy=(Andy, {road_trip=(
  (road_trip,reg), 3)},
  jurassic_park=(
  (jurassic_park,child), 2)},
  4.5)}
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();
}

```

5 Conclusion

In the paper our approach to developing a formal specification for a given problem using a complementary syntax/semantics approach is described. Not least, our approach can be also seen as a formal approach to program construction with all benefits of formal approaches. The proposed approach can be also applied if the user's requirements are not well defined; more symbols or attributes (attribute rules or constraints) can be easily added in a later phase (when the user comes up with new requirements/functionalities), and a new prototype will be immediately generated. The essence of our approach is the development of a domain-specific language that describes the user interaction with a system or the functionality of a system. While executing programs written in a specified domain-specific language

the functionality of a system and user's requirements can be validated. The starting point of our approach is the identification of concepts in the problem domain. Here, well known techniques from object-oriented design, such as use case diagrams and conceptual class diagrams, are used. However, our approach can be used also with data-flow diagrams and entity-relation diagrams. In that case just new transformation rules have to be defined, similar to those presented in table 1 and table 2.

In our future work we would like to investigate the possibility to obtain a domain-specific language only from a use case diagram which describes the functionality of a system. It is well known that use case diagrams and class diagrams represent different views on a given problem and that there is no direct transformation between those two techniques. Has such context-free grammar some valuable information for constructing a conceptual class diagram? Is it possible that a context-free grammar of a domain-specific language, derived from use case diagram, describes the class diagram for a given problem? Such findings might have some impact on current object-oriented design. Hence, our future work is to explore this connection.

References

- [1] S. Adolph. *Patterns for Effective Use Cases*. Addison-Wesley, 2002.
- [2] A. 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.
- [3] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.
- [4] B. Bryant and B. 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.
- [5] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.
- [6] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323. Lecture Notes in Computer Science, Springer-Verlag, 1988.
- [7] M. Fowler. *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley Longman, 1997.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [9] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, March 2000.
- [10] 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.
- [11] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [12] D. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.
- [13] C. Larman. *Applying UML and Patterns—an Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 2nd edition, 2002.
- [14] K. Levi and A. Arsanjani. A goal-driven approach to enterprise component identification and specification. *Communications of the ACM*, 45(10):45–52, October 2002.
- [15] K.J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
- [16] M. Mernik and D. Parigot (Eds.). Attribute grammars and their applications. *Informatica*, 24(3), September 2000.
- [17] M. Mernik, J. Heering, and T. Sloane. When and how to develop domain-specific languages. Technical Report SEN-E0309, University of Maribor, CWI Amsterdam, and Macquarie University, 2003.
- [18] 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.
- [19] B. Roussev. Generating ocl specifications and class diagrams from use cases: A newtonian approach. In *IEEE CD ROM Proceedings of 36th Hawaii International Conference on System Sciences*, 2003.
- [20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [21] I. Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1996.
- [22] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
- [23] A. van Deursen and L. Moonen. The video store revisited thoughts on refactoring and testing. In *Proceedings of the 3rd International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002)*, University of Cagliari, 2002, pages 71–76, 2002.
- [24] D. Wile. Supporting the DSL Spectrum. *Journal of Computing and Information Technology, Special Issue on Domain-Specific Languages*, R. Laemmel and M. Mernik, eds., 9(4):263–287, Dec 2001.