Grammatical Approach to Problem Solving

Pedro Rangel Henriques[1], Tomaž Kosar[2], Marjan Mernik[2], Maria João Varanda Pereira[3], Viljem Žumer[2]

[1] University of Minho, Department of Computer Science, Portugal
prh@di.uminho.pt

[2] University of Maribor, Faculty of Electrical Engineering and Computer Science, Slovenia
{tomaz.kosar, marjan.mernik, zumer}@uni-mb.si

[3] Polytechnic Institute of Bragança, Portugal
mjoao@ipb.pt

Abstract. *The paper presents a grammatical approach to problem solving. It supports formal software specification using attribute grammars, from which a rapid prototype can be generated as well 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 a domain-specific language corresponds to the execution of a prototype of a system on a particular use-case.*

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

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 [5] [6] are well suited for a such design for a change. 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 [16]. In the latter case the prototype is used to clarify the user's needs. In both cases, the rapid prototyping approach is used when the user's requirements are not well defined. The proposed approach, i.e. *the grammatical approach to problem solving*, rests on the success reached by attribute grammars in the specification of language semantics [9] [4] and in the systematic implementation of language processing tools [7] [8]. In the paper the grammatical approach to problem solving supported by an attribute grammar developed and written in an object-oriented style (OOAG) 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] 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 works are described. The grammatical approach to problem solving 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 software development can be seen as an extension of object-oriented design methods [15] [5] [6] where a problem do-

main model is developed from use-cases and class diagram. However, their main goal is to develop good software models (e.g. as in [11]). 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) [1] [10], 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. However, our approach differs from [1] [10] 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. Our approach has also 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. [3]). In [3] 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. In more complex cases 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 [14]. 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) uses use-case diagrams and conceptual class diagrams [5]. The use-case diagram describes the functionality of the system. Use-cases are narrative descriptions of specific tasks, while the conceptual class diagram captures concepts and relationships between them. Guidelines for developing the conceptual class diagram can be found in [15]. 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. For identifying concepts and their relationships in the problem domain our grammatical approach is not limited to object-oriented design. Also other approaches, such as data-flow diagrams and entity-relation diagrams which show the flow of work and the relationship between activities and deliverables, can be applied. However, object-oriented design [2] [6] is now almost the-facto standard for software system design and it is also used in our approach. Therefore, the starting point of our grammatical approach is a conceptual class diagram. To enable rapid prototyping the following steps are used in our approach (Fig. 1):

- deriving the context-free grammar from the conceptual class diagram,
- describing the semantics of every concept,
- generating the rapid prototype of a system.

Domain concepts and relationships among them are identified in the conceptual class diagram and are further 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 the domain specific language is described with attribute grammars (derived from domain concepts) from which a compiler is automatically generated. The execution of a particular program written in a domain-specific language corresponds to the execution of a prototype of a system on a particular use-case. The brief description of the above steps is described in the following 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. expression or declaration in the general-purpose programming language). On the other hand, at more concrete level non-terminals and terminals are used to describe the structure of a concept (e.g. variable declarations consist of variable type and variable name followed by semicolon). Therefore, both the concepts and relationships between them are captured in a context-free grammar. But, this is also

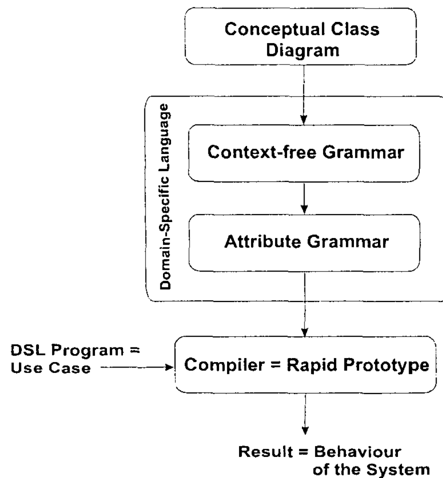

Figure 1: High-level view of the grammatical approach

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.

In general, classes are mapped to non-terminal symbols and attributes are mapped to terminal symbols. The result of this step is a context-free grammar. 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 described 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 on an example for multiplicity 0..m:

```
A     -> MoreB
MoreB -> MoreB B | epsilon
```

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 described 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.

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 [9] 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 the implicit feature of a grammar and is 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. The activities involved in the second step are intellectually demanding and may require significant creativity of the designer. 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 [12] has been used. The LISA system automatically generates a compiler or interpreter and other language-based tools, such as language-knowledgeable editor, inspectors, and animators [8] from 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: Chocolate Vending Machine

Our approach is illustrated on a simple example. More examples can be found in the technical report [13].

The problem: We want a program for the daily management of a Chocolate Vending Machine. 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 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

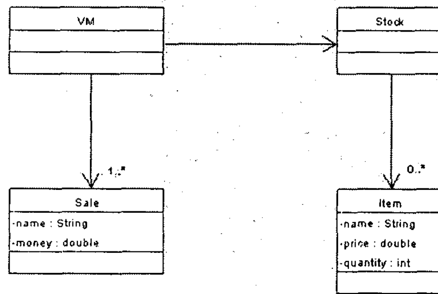

Figure 2: The Conceptual Class Diagram for Vending Machine

- the amount of money is different from the price.

The Conceptual Class Diagram: After the analysis of the problem stated above, we identified the *vending machine (VM)* as the main concept. The two other important concepts in the management of the *vending machine* are: *Stock*, and *Sale*. *Stock* is a set of items, and each *Item* has a *name*, a *price*, and a *quantity*. The data to be kept for each *Sale* operation is the item *name* 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 in Fig. 2.

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:

```
VM          -> Stock Sales
Stock       -> Stock Item
            |  &
Item        -> name price quantity

Sales       -> Sales Sale
            |  Sale
Sale        -> name money
```

The Semantics: The next step is to define the semantics for each non-terminal symbol. Due to limited page length only a part of the whole attribute grammar specification is given below. The complete example is given in [13]. According to the problem description, the *vending machine* should have two attributes: `FStkTab` (the final stock table) and `Income` (the final income). These attributes denote the two required value computations (the problem goals). The inherent modularity of attribute grammars enables us to write separate attribute grammar modules for each attribute associated to a grammar symbol. The complete specification is the attribute grammar obtained by composition of all modules. Only attribute grammar for VM module are shown below.

We will start with stock table evaluation. Notice that its value is not computed locally to the `VM` symbol definition; it depends on the sales processing. To perform such computation, it is necessary to know the stock table before to start selling (initially the stock table is empty). 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).

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

NonTerm VM:
        Inh: {}
        Syn: { FStkTab: TAB }

        vm-manager(VM -> Stock Sales):
           // associated semantics
           VM.FStkTab    = Sales.FStkTab
           Stock.StkTab = {}
           Sales.StkTab = Stock.FStkTab
```

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

```
NonTerm VM:
        Inh: {}
        Syn: { Income: int }

        vm-manager(VM -> Stock Sales):
           VM.Income = Sales.Sum
```

Notice that the stock declaration part plays no role in the *Income* evaluation. The partially presented attribute grammar represents the formal specification for the given problem.

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. An example of the mentioned inherited modularity is shown below: in the first language (VM_1) only stock description is introduced and sales description is added to the next language (VM_2).

A part of these specifications are shown below. Note the straightforward translation from above specifications to LISA.

```
language VM_1 {
  ...
  attributes Hashtable *.StkTab, *.FStkTab;
  ...
  rule VM {
    VM ::= STOCK compute {
      VM.FStkTab = STOCK.FStkTab;
      STOCK.StkTab = new Hashtable();
    };
  }
  rule Stock {
    STOCK ::= STOCK ITEM compute {
      STOCK[0].FStkTab = ITEM.FStkTab;
      STOCK[1].StkTab  = STOCK[0].StkTab;
      ITEM.StkTab = STOCK[1].FStkTab;
    }
    | epsilon compute {
      STOCK.FStkTab = STOCK.StkTab;
    };
  }
  ...
 } // language VM_1

language VM_2 extends VM_1 {
  ...
  rule overrides VM {
    VM ::= stock_description STOCK
           sales_description SALES
           compute {
      VM.FStkTab   = SALES.FStkTab;
      STOCK.StkTab = new Hashtable();
      SALES.StkTab = STOCK.FStkTab;
    };
  }
  ...
} // language VM_2

language VM_3 extends VM_2 {
 attributes double *.income,
 *.sum, *.amount;
 rule extends VM {
    compute {
       VM.income = SALES.sum;
    }
  }
 rule extends Sales {
    SALES ::=  SALE compute {
       SALES.sum = SALE.amount;
    }
    | SALES SALE compute {
       SALES[0].sum = SALES[1].sum
                      + SALE.amount;
    };
  }
  ...
 } // language VM_3
}
```

From above specifications the VM_3 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 written in our DSL.

```
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 the above program is the following stock and money income:

```
FStkTab: {twix=(0.60 Euro, 2),
  kitkat=(0.60 Euro, 15),
  mars=(0.50 Euro, 9)}
Income: 2.3 Euro
```

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 used if the user's requirements are not well defined. 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 explained in Section 3.

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-ease 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] 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.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[3] 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.

[4] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Definitions, systems and bibliography. volume 323. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[5] M. Fowler. *UML Distilled. Applying the Standard Object Modeling Language.* Addison-Wesley Longman, 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[7] Jan Heering and Paul Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, March 2000.

[8] Pedro Henriques, Maria Varanda Pereira, Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem 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.

[9] D. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[10] 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.

[11] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

[12] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem 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.

[13] Maria Varanda Pereira, Marjan Mernik, Pedro Henriques, Tomaž Kosar, and Viljem Žumer. Object-oriented attribute grammar based grammatical approach to problem specification. Technical report, University of Minho, Department of Computer Science, Braga, 2002.

[14] Boris 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.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[16] I. Sommerville. *Software Engineering.* Addison-Wesley, 5th edition, 1996.

[17] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.