An Attribute Grammar Specification of IIS*Case PIM Concepts

Ivan Luković[1], Maria João Varanda Pereira[2], Nuno Oliveira[3], Daniela da Cruz[3], Pedro Rangel Henriques[3]

[1] University of Novi Sad, Faculty of Technical Sciences,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia,
ivan@uns.ac.rs
[2] Politechnique Institute of Bragança, Escola Superior de Tecnologia e Gestão, Campus de Santa Apolónia - Apartado 1134
5301-857 Bragança, Portugal
mjoao@ipb.pt
[3] Universisty of Minho, Department of Computer Science,
Campus de Gualtar - 4710-057 Braga, Portugal
{nunooliveira,danieladacruz,prh}@di.uminho.pt

Abstract. IIS*Case is a model driven software tool that provides information system modeling and prototype generation. It comprises visual and repository based tools for creating various platform independent model (PIM) specifications that are latter transformed into the other, platform specific specifications, and finally to executable programs. Apart from having PIMs stored as repository definitions, we need to have their equivalent representation in the form of a domain specific language. One of the main reasons for this is to allow for checking the formal correctness of PIMs being created. In the paper, we present such a meta-language, named IIS*CDesLang. IIS*CDesLang is specified by an attribute grammar (AG), created under a visual programming environment for AG specifications, named VisualLISA.

Keywords: Information System Modeling; Model-Driven Approaches; Domain Specific Languages; Domain Specific Modeling; Attribute Grammars

1 Introduction

In this paper we present a textual language aimed at modeling platform independent model (PIM) specifications of an information system (IS). Our research goals are to create such a language and couple it with Integrated Information Systems CASE Tool (IIS*Case). IIS*Case is a model driven software tool that provides IS modeling and prototype generation. At the level of PIM specifications, IIS*Case provides conceptual modeling of database schemas and business applications. Starting from such PIM models as a source, a chain of model-to-model and model-to-code transformations is performed in IIS*Case so as to obtain executable program code of software applications and database scripts for a selected target platform. One of the main motives for developing IIS*Case is in the following. For many years, the most favorable conceptual data model is Entity-Relationship (ER) data model. A typical scenario of a database schema design process provided by majority of existing CASE tools is to create an ER database schema first and then transform it into the relational database schema. Such a scenario has many advantages, but also there are serious disadvantages [5]. We overcome them by creating an alternative approach and related techniques that are mainly based on the usage of model driven software development (MDSD) and Domain Specific Language (DSL) paradigms. The main idea was to provide the necessary PIM meta-level concepts to IS designers, so that they can easily model semantics in an application domain. After that, they may utilize a number of formal methods and complex algorithms to produce database schema specifications and IS executable code, without any expert knowledge.

In order to provide design of various PIM models by IIS*Case, we created a number of modeling, meta-level concepts and formal rules that are used in the design process. Besides, we also developed and embedded into IIS*Case visual and repository based tools that apply such concepts and rules. They assist designers in creating formally valid models and their storing as repository definitions in a guided way.

Apart from having created PIM models stored as repository definitions, there is a strong need to have their equivalent representation given in a form of a textual language, for the following reasons. (i) Firstly, despite that we may expect that average users prefer to use visually oriented tools for creating PIM specifications, we should provide more experienced users with a textual language and a tool for creating PIM specifications more efficiently. (ii) Secondly, we need to have PIM meta-level concepts specified formally in a platform independent way, i.e. to be fully independent of repository based specifications that typically may include some implementation details. (iii) The third, but not less important, by this we create a basis for the development of various algorithms for checking the formal correctness of the models being created, as well as for the implementation of some semantic analysis. Therefore, we need a grammar specification of our meta-level concepts and rules. By such a grammar, we specify a DSL [3, 9] that recognizes problem domain concepts and rules that are applied in the conceptual IS design provided by IIS*Case. In the paper, we present a specification of such meta-language, named IIS*CDesLang. IIS*CDesLang is used to create PIM project specifications that may be latter transformed into the other specifications, and finally to programs.

There are a number of meta-modeling approaches and tools suitable for the purpose of creating IIS*CDesLang, as well as many other applications in various problem domains like it is, for example, [16]. One of them is the Meta-Object Facility [18] proposed by the OMG, where the meta-model is created by means of UML class diagrams and Object Constraint Language (OCL). The Generic Modeling Environment (GME) [19] is a configurable toolkit for domain-specific modeling and program synthesis. In MetaEdit+ [20] models are created through a graphical editor and a proprietary Report Definition Language is used to create code from models. The Eclipse Modeling framework (EMF) [21] is also a commonly used meta-modeling framework, where meta-meta-model named Ecore is used to create meta-models, or to import them from UML tools or textual notations like one presented in [17].

To create IIS*CDesLang, a visual programming environment (VPE) for attribute grammar specifications, named VisualLISA [11, 13] is selected. In the paper, we focus on the following application PIM concepts: project, application system, form type, component type, application, call type, and basilar concepts as attribute and domain. We applied VisualLISA Syntactic and Semantic Validators to check the correctness of the specified grammar.

A benefit of introducing IIS*CDesLang is to use VisualLISA attribute grammar (AG) specifications and generate a parser aimed at checking the formal correctness of created project models. In this way, we may help designers in raising the quality of created IS specifications. Currently, we developed an AG specification of IIS*CDesLang. The main goal of this paper is to present a part of such specification and address main future research directions. Apart from having the AG specification of IIS*CDesLang, we also need the appropriate design and syntax checker tools. They are still under development. Therefore, we were not able so far to test the efficiency of the concept as a whole. It remains to be one of our next research tasks.

Apart from Introduction and Conclusion, the paper is organized in four sections. In Section 2, we give a short presentation of IIS*Case. Preliminaries about VisualLISA programming environment are given in Section 3. Selected IIS*CDesLang PIM concepts are briefly described in Section 4. In Section 5 we present an attribute grammar specification of IIS*CDesLang, created by VisualLISA.

2 IIS*Case and Conceptual Modeling

IIS*Case, as a software tool assisting in IS design and generating executable application prototypes, currently provides:

- Conceptual modeling of database schemas, transaction programs, and business applications of an IS;
- Automated design of relational database subschemas in the 3rd normal form (3NF);
- Automated integration of subschemas into a unified database schema in the 3NF;
- Automated generation of SQL/DDL code for various database management systems (DBMSs);
- Conceptual design of common user-interface (UI) models; and
- Automated generation of executable prototypes of business applications.

Apart from the tool, we also define a methodological approach to the application of IIS*Case in the software development process [6, 8]. By this approach, the software development process provided by IIS*Case is, in general, evolutive and incremental. It enables an efficient and continuous development of a software system, as well as an early delivery of software prototypes that can be easily upgraded or amended according to the new or changed users' requirements. In our approach we strictly differentiate between the specification of a system and its implementation on a particular platform. Therefore, modeling is performed at the high abstraction level, because a designer creates an IS model without specifying any implementation details. Besides, IIS*Case provides some model-to-model transformations from PIM to Platform-Specific Models (PSM) and model-to-code transformations from PSMs to the executable program code.

Detailed information about IIS*Case may be found in several authors' references and we do not intend to repeat them here. A case study illustrating main features of IIS*Case and the methodological aspects of its usage is given in [6]. The methodological approach to the application of IIS*Case is presented in more details in [8]. At the abstraction level of PIMs, IIS*Case provides conceptual modeling of database schemas that include specifications of various database constraints, such as domain, not null, key and unique constraints, as well as various kinds of inclusion dependencies. Such a model is automatically transformed into a model of relational database schema, which is still technology independent specification. It is an example of model-to-model transformations provided by IIS*Case. [7]

In [1] we present basic features of SQL Generator that are already implemented into IIS*Case, and aspects of its application. We also present methods for implementation of a selected database constraint, using mechanisms provided by a relational DBMS. It is an example of model-to-code transformations provided by IIS*Case.

At the abstraction level of PIMs, IIS*Case also provides conceptual modeling of business applications that include specifications of: (i) UI, (ii) structures of transaction programs aimed to execute over a database, and (iii) basic application functionality that includes the following "standard" operations: data retrieval, inserts, updates, and deletes. Also, a PIM model of business applications is automatically transformed into the program code. In this way, fully executable application prototypes are generated. Such a generator is also an example of model-to-code transformations provided by IIS*Case and its development is almost finished. [2]

3 Designing DSLs using VisualLISA

As it follows from the definition of AGs [15] they are not as easy to specify as people would desire because there is a gap between the language description and the grammar meta-language that must be interpreted. The user must take care about choosing the appropriate attributes and their evaluation rules. Since the beginning, the literature concerned with the formal approach to compiler development presents AGs drawing syntax trees decorated with attributes. So it is usual to sketch up on paper attributed trees with semantic rules to describe an AG, avoiding spending time with syntactic details. However, such informal drawings must be translated manually (by language experts) into the specific input notation of a compiler generator. This inconvenience discourages language designers to use AGs. Such an attitude prevents them of resorting to systematic ways to implement the languages and their supporting tools [10].

For modeling the new DSL we use here a Visual Language (VL) and its respective programming environment (VPE) called VisualLISA, as it is proposed in [11], and conceived in [13]. The idea of introducing VL is not only about having a nice visual depiction that will be translated into a target notation latter on, but also having a possibility of checking syntactic and semantic consistency.

VisualLISA environment offers a visually oriented and non-errorprone way for AG modeling and an easy translation of AG models into a target language. Three main features of VisualLISA are: (i) syntax validation, (ii) semantics verification and (iii) code generation. The syntax validation restricts some spatial combinations among the icons of the language. In order to avoid syntactic mistakes, the model edition is syntax-directed. The semantics verification copes with the static and dynamic semantics of the attribute grammar meta-language. Finally, the code generation produces code from the drawings sketched up. The target code would be LISA specification language (LISAsl) - the meta-language for AG description under LISA generator - or XAGra specification [13]. LISAsl specification is passed to the LISA system [4, 14] in a straightforward step. XAGra [12] textual language was conceived with the aim of giving to VisualLISA development environment more versatility and further usage perspectives.

In our case, we are specifying IIS*CDesLang, a new DSL for IIS*Case tool, using VisualLISA; the target code will be LISAsl in order to implement IIS*CDesLang interpreter in LISA system. This specification is further detailed in Section 5.

4 PIM Concepts and IIS*CDesLang

IIS*CDesLang is a meta-language aimed at formal specification of all the concepts embedded into IIS*Case repository definitions. In this paper, we focus on the PIM concepts only. Hereby, we give a brief overview of the following concepts covered by IIS*CDesLang: project, application system, form type, component type, application, call type, as well as fundamental concepts: attribute and domain. In this section we present the PIM concepts only from the technical point of view. Additional and detailed information may be found in several authors' references, as well as in [6, 8].

A work in IIS*Case is organized through projects. Everything that exists in the IIS*Case repository is always stored in the context of a project. A designer may create as many projects as he or she likes. One project is one IS specification and has a structure represented by the project tree. Each project has its (i) name, (ii) fundamental concepts or fundamentals for short, and (iii) application systems. A designer may also define various types of application systems – application types for short, and introduce a classification of application systems by associating each application system to a selected application type. At the level of a project there is a possibility to generate various reports that present the current state of the IIS*Case repository. IIS*Case provides various types of repository reports.

Application systems are organizational parts, i.e. segments of a project. We suppose that each application system is designed by one, or possibly more than one designer. Fundamental concepts are formally independent of any application system. They are created at the level of a project and may be used in various application systems latter on. Fundamental concepts are: domains, attributes, inclusion dependencies and program units. In the paper, we focus on domains, attributes, and functions as a category of program units.

In the following text, we use a notion of domain with a meaning that is common in the area of databases. It denotes a specification of allowed values of some database attributes. We classify domains as (i) primitive and (ii) user defined. Primitive domains exist "per se", like primitive data types in various formal languages. We have a small set of primitive domains already defined, but we allow a designer to create his or her own primitive domains, according to the project needs. User defined domains are created by referencing primitive or previously created user defined domains. Domains are referenced latter from attribute specifications. A list of all project attributes created in IIS*Case belongs to fundamentals. Attributes are used in various form type specifications of an application system.

A concept of a function is used to specify any complex functionality that may be used in other project specifications. Each function has its name as a unique identifier, a description, a list of formal parameters and a return value type. Besides, it encompasses a formal specification of function body that is created by the *Function Editor* tool of IIS*Case.

4.1 Domains and Attributes

A specification of a primitive domain includes: name, description, default value, and a "length required" item specifying if a numeric length: a) not to be, b) may be or c) must be given. User defined domains are to be associated with attributes. A user defined domain specification includes: a domain name, description (like all other objects in IIS*Case repository), default value, domain type, and check condition.

We distinguish the following domain types: (i) domains created by the inheritance rule and (ii) complex domains that may be created by the: a) tuple rule, b) choice rule or c) set rule. Inheritance rule means that a domain is created by inheriting a specification of a primitive domain or a previously defined user defined domain. It inherits all the rules of a superordinated domain and may be "stronger" than the original one.

A domain created by the tuple rule is called a tuple domain. It represents a tuple (record) of values. For such a complex domain, we need to select some attributes as items of a tuple domain. Therefore, we may have a recursive usage of attributes and domains, because we need some already created attributes to use in a tuple domain specification. A domain created by the choice rule – choice domain is technically specified in the same way as tuple domain. Choice domain is the same as choice type of XML Schema Language. Each value of such a domain must correspond to exactly one attribute which is an item in the choice domain. A set domain represents sets (collections) of values over a selected domain. To create it, we only need to reference an existing domain as a set member domain. Each value of this domain will be a set of values, each of them from a set member domain.

Check condition, or the domain check expression is a regular expression that further constrains possible values of a domain. We have a formal syntax developed and the *Expression Editor* tool that assists in creating such expressions. We also have a parser for checking syntax correctness.

Currently we do not have a possibility to define allowed operators over a domain in IIS*Case repository. It is a matter of our future work.

Each attribute in an IIS*Case project is identified only by its name. Therefore, we obey to the Universal Relation Scheme Assumption (URSA) [5], well known in the relational data model. The same assumption is also applicable in many other data models. Apart from the name and description, we specify if an attribute is included into database schema, derived, or renamed.

Most of the project attributes are to be included into the future database schema. However, we may have attributes that will present some calculated values in reports or screen forms that are not included into database schema. They derive their values on the basis of other attributes by some function, representing a calculation. Therefore, we classify attributes in IIS*Case as a) included or b) non-included in database schema. Also we introduce another classification of attributes, by which we may have: a) elementary or non-derived and b) derived attributes. If an attribute is specified as non-derived, it obtains its values directly by the end users. Otherwise, values are dervied by a function that may represent a calculation formula or any algorithm. Any attribute specified as non-included in database schema must be declared as derived one.

A derived attribute may reference an IIS*Case repository function as a query function. Query function is used to calculate attribute values on queries. Only a derived attribute may additionally reference three IIS*Case repository functions specifying how to calculate the attribute values on the following database operations: insert, update and delete.

In IIS*Case we have a notion of renamed attribute. A renamed attribute references a previously defined attribute and has to be included in the database schema. It has its origin in the referenced attribute, but with a slightly different semantics. Renaming is a concept that is analogous to the renaming that is applied in mapping Entity-Relationship (ER) database schemas into relational data model. If a designer specifies that an attribute A1 is renamed from A, actually he or she introduces an inclusion dependency of the form $[A1] \subseteq [A]$ at the level of a universal relation scheme.

Each attribute specification also includes: a reference to a user defined domain, default value and check condition. Check condition, or the attribute check expression is a regular expression that further constrains possible values of the attribute. It is defined and used in a similar way as it is for domain check expressions. If the attribute check expression and domain check expression are both defined, they will be connected by the logical AND.

Both user defined domain and attribute specifications also provide for specifying a number of display properties of screen items that correspond to the attributes and their domains. Such display properties are used by the IIS*Case *Application Generator* aimed at generating executable application prototypes. Display properties of an attribute may inherit display properties of the associated domain or may override them.

4.2 Application Systems, Form Types and Applications

Apart from name, type and description, each application system may have many child application systems. In this way, a designer may create application system hierarchies in an IIS*Case project. An application system may comprise various kinds of IIS*Case repository objects. For PIM specifications, only two kinds of objects are important: a) form types and b) business applications, or applications, for short.

A form type is the main modeling concept in IIS*Case. It generalizes document types, i.e. screen forms or reports by means of users communicate with an IS. It is a structure defined at the abstraction level of schema. Using the form type concept, a designer specifies a set of screen or report forms of transaction programs and, indirectly, specifies database schema attributes and constraints. Each particular business document is an instance of a form type.

Form types may be (i) owned, if they are created just in the application system observed, or (ii) referenced, if they are "borrowed" from another application system, regardless if it is referenced as a child application system. If a form type is referenced it is a read-only object in the application system.

Business applications are structures of form types. Each application has its name, description, and a reference to exactly one form type that is the entry form type of the application. To exist, each application must contain at least the entry form type. The execution of a generated application always starts from the entry form type. Form types in an application are related by form type calls. A form type call always relates two form types: a calling form type and a called form type. By a form type call, a designer may formally specify how values are passed between the forms during the call execution. There are also other properties specifying details of a call execution. *Business Application Designer* is a visually oriented tool for modeling business applications in IIS*Case.

Each form type has the following properties: name, title, frequency of usage, response time and usage type or usage for short. By the usage property form types are classified as menus or programs. Menu form types are used to generate just menus without any data items. Program form types specify transaction programs with the UI. They have a complex structure and may be designated as (i) considered or (ii) not considered in database schema design. The first option is used for all form types aimed at updating database, as well as for some report form types. Only the form types that are "considered in database schema design" participate latter on in generating database schema. The former option is used for report form types only.

Each program form type is a tree structure of component types. It must have at least one component type. A component type has a name, reference to the parent component type (always empty for the root component type only), title, number of occurrences, and operations allowed. Number of occurrences may be specified as (i) 0-N or (ii) 1-N. 0-N means that for each instance of the parent component type, zero or more instances of the subordinated component type are allowed. 1-N means that for each instance of the parent component type, we require the existence of at least one instance of the subordinated component type. By operations allowed a designer may specify the following "standard" database operations over the component types: query, insert, delete, and update instances of the component type.

Each component type has a set of attributes included from IIS*Case repository. An attribute may be included in a form type at most once. Consequently, if a designer includes an attribute into a component type, it cannot be included in any other component type of the same form type. Each attribute included in a component type may be declared as: (i) mandatory or optional, and (ii) modifiable, query only or display only. Also, a set of allowed operations over an attribute in a component type is specified. It is a subset of the set of operations {query, insert, nullify, update}. A designer may also specify "List of Values" (LOV) functionality of a component type attribute by referencing a LOV form type and specifying various LOV properties.

Each component type must have at least one key. A component type key consists of at least one component type attribute. Each component type key provides identification of each component instance, but only in the scope of its superordinated component instance. Also, a component type may have uniqueness constraints, each of them consisting of at least one component type attribute. A uniqueness constraint provides an identification of each component instance, but only if it has a non-null value. On the contrary to keys, attributes in a uniqueness constraint may be optional. Finally, a component type may have a check constraint defined. It is a logical expression constraining values of each component type instance. Like domain check expressions, they are specified and parsed by *Expression Editor*.

Both component type and form type attribute specifications provide for specifying a vast number of display properties of generated screen forms, windows, components, groups, tabs, context and overflow areas, and items that correspond to the form type attributes. There is also the *Layout Manager* tool that assists designers in specifying component type display properties, and a tool *UI*Modeler* that is aimed at designing templates of various common UI models. All of these display properties combined with a selected common UI model are used by the IIS*Case *Application Generator*.

A presentation of the whole IIS*CDesLang syntax is out of scope of this paper. However, in the following example, we illustrate a form type created in an IIS*Case project named *FacultyIS*, and the corresponding IIS*CDesLang program. Figure 1 presents a form type defined in the child application system *Student Service* of a parent application system *Faculty Organization*. It refers to information about student's grades (STG). It has two component types: STUDENT representing instances of students, and GRADES, representing instances of grades for each student. By the form type STG, we allow having students with zero or more grades. Component type attributes are presented in italic letters. *StudentId* is the key of the component type STUDENT, while *CourseShortName* is the key of GRADES. By this, each grade is uniquely identified by *CourseShortName* within the scope of a given student. Allowed database operation for STUDENT is only *read* (shown in a small rectangle on the top of the rectangle representing the component type), while the allowed database operations for GRADES are *read*, *insert*, *modify* and *delete*.

Figure 2 presents a fragment of IIS*CDesLang program that corresponds to the form type specification from Figure 1.

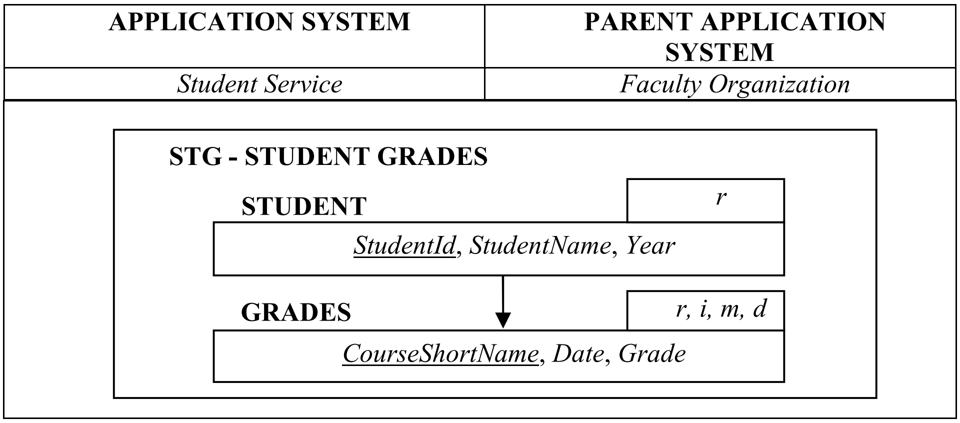

Fig. 1. A form type in the application system *Student Service*

```
Project: Faculty IS
  Application System: Faculty Organization
  ...
  Application System: Student Service
                      is-child-of <<Faculty Organization>>
    FormType: "STG - Student Grades"
      ComponentType: STUDENT
        Allowed Operations:  read
        Position: sameWindow
        DataLayout: TableLayout
        Window Position: Center
        Component Type Attributes:
          Name: StudentID
            CTAMandatory: Yes
            ...
          Name: StudentName
            ...
          Name: Year
            ...
        Component Type KEY: StudentID
      ComponentType: GRADES is-child-of <<Student>>
        NoOfOccurrences:(0:N)
        Allowed Operations: read, insert, modify,delete
        ...
        Component Type Attributes:
          Name: CourseShortName
            ...
        Component Type KEY: CourseShortName
  ...
```

Fig. 2. A fragment of IIS*CDesLang program that corresponds to the form type in Figure 1

5 The Attribute Grammar Specification of IIS*CDesLang

In this section, we discuss how VisualLISA (whose principles and functionalities were introduced in section 3) is used to create IIS*CDesLang. Due to space limitations, we only present a small set of productions and semantic calculations, just to show how we use the visual editor to model the language. Before that we present a short description of VisualLISA look and feel, and main usage.

Figure 3 shows the editor look and feel; it exhibits its main screen with four sub-windows. To specify an attribute grammar a user starts by declaring the productions (in *rootView* – bottom-left sub-window) and rigging them up by dragging the symbols from the dock to the editing area (in *prodsView* – upper-left sub-window), as commonly done in VPEs. The composition of the symbols is almost automatic, since the editing is syntax-directed. When the production is specified, and the attributes are attached to the symbols, the next step is to define the attribute evaluation rules. Once again, the user drags the symbols from the dock, in *rulesView* (upper-right sub-window), to the editing area. To draw the computations links should connect some of the (input) attributes to an (output) attribute using functions. Functions can be pre-defined, but sometimes it is necessary to resort to user-defined functions that should be described in *defsView* (bottom-right sub-window). In this sub-window it is also possible to import packages, define new data-types or define global lexemes.

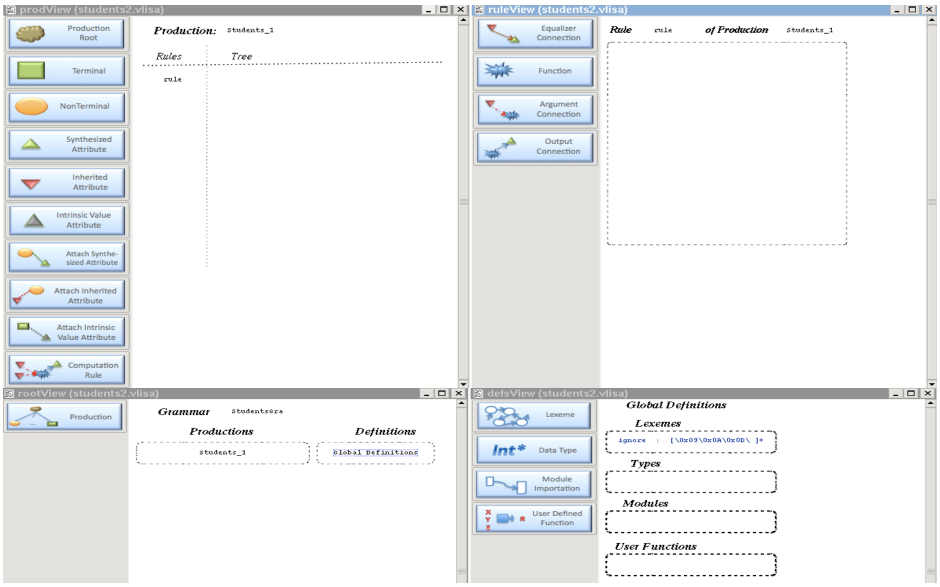

Fig. 3. VisualLISA main windows

In this example of how we use VisualLISA to rapidly develop a formal specification for IIS*CDesLang, we will show how the following condition is formalized and verified using the visual editor: *"The application types associated to an application system should be previously defined"*.

Figure 4 shows the first production of IIS*CDesLang (the one having the grammar axiom as the tree root). As can be observed the root (*Project*) derives in three other non-terminal symbols (*ApplicationTypes*, *ApplicationSystems*, and *Fundamentals*) and two terminals. Apart from that structural description, the production shown in Figure 4.a states that the attribute *verify* of the root symbol has the same value as the synthesized attribute *verify* (green triangle) of the non-terminal *ApplicationSystems*. In Figure 4.b it is presented a detail of the same production, specifying that the inherited attribute *setof_types* (red inverted-triangle) of non-terminal *ApplicationSystems,* inherits the value of the attribute *setof_types* of the non-terminal *ApplicationTypes*.

In Figure 5, we present how the attribute *setof_types* of the non-terminal *AplicationTypes*, is computed. First notice that the production for this non-terminal has two options: (i) a non-recursive one, where *AplicationTypes* derives only one *Aplication-Type* (Figure 5.a) and (ii) a recursive case, where the left-hand side (LHS) non-terminal derives into an *AplicationType* and recursively calls itself.

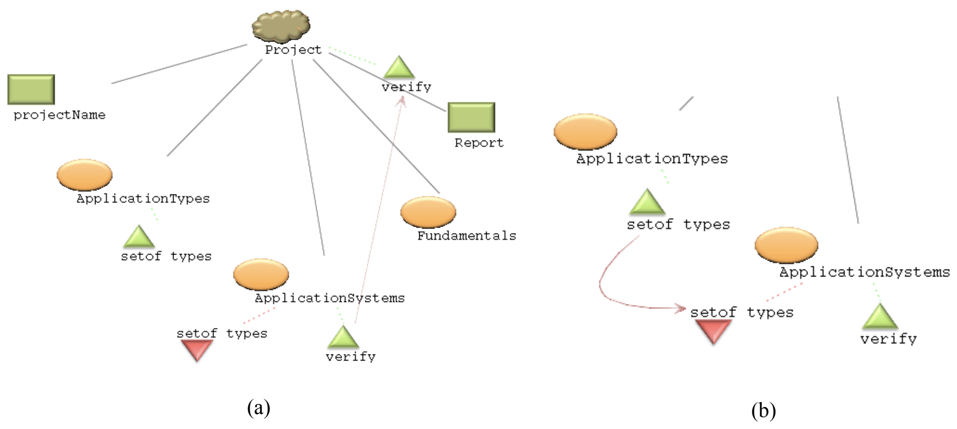

Fig. 4. Production structure and computation rules for non-terminal Project. (a) computation rule for attribute *verify*; (b) computation rule for inherited attribute *setof_types*

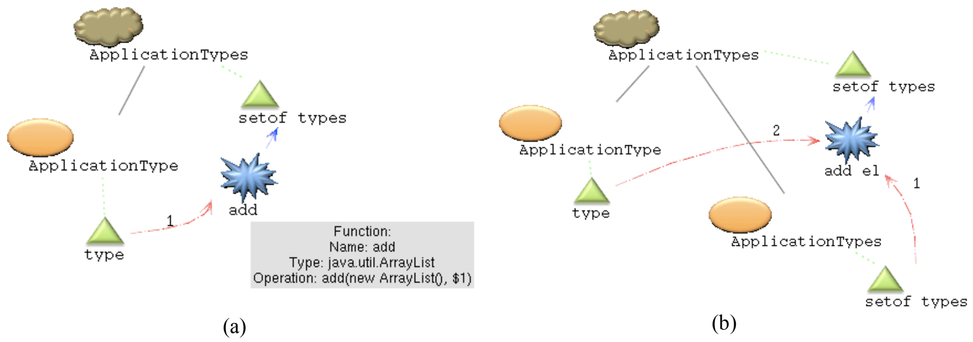

Fig. 5. Production structure and computation of attribute *setof_types* of the element *ApplicationTypes*. (a) non-recursive case; (b) recursive case

In this production, we are interested in collecting the application type names that can be associated to the application systems, as explained before. To describe this in VisualLISA we created a function that adds a string to a list, and this function is used to collect the types that are synthesized from each non-terminal *ApplicationType*. The blue explosion symbol denotes the function, the dashed-arrows define the arguments of these functions, and the straight blue arrows denote to which attribute the output of the function is assigned. The numbers in the dashed-arrows indicate the order of the arguments in the function, which are then used as '$i' in the function body.

Recall Figure 4.b, where an inherited attribute is assigned the value of the attribute we just compute in Figure 5. The reason why we need to inherit this attribute is in the fact that we must check whether the type of each application system is in this list. Otherwise the language is not correct according to the contextual condition that we try to verify in this example. Figure 6 presents the recursive option of the production with the *ApplicationSystems* as LHS symbol.

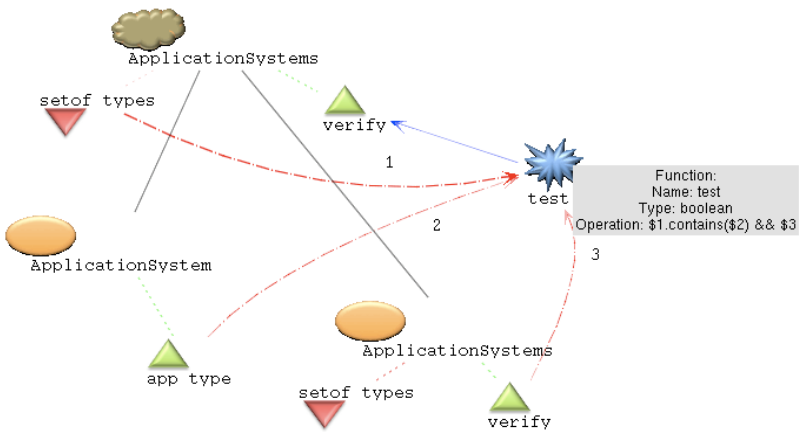

Fig. 6. Recursive case for production of the symbol *ApplicationSystems* and computation of the attribute *verify*

From each application system we synthesize its application type (attribute *app_type*). Then, we use the inherited attribute *setof_types* and the value that results from applying this computation to the rest of the application systems in the language, to inject these three arguments in a function that tests if the *setof_types* ($1 in the operation description of Figure 6) contains the value of the synthesized attribute *app_type* ($2 in the operation description). As this operation returns a boolean value, we check using the logic *and* operation, if this value and the value of the attribute *verify* ($3 in the operation description) are both true. The output of the function is also a Boolean and is assigned to attribute *verify* of the LHS symbol.

The non-recursive option of this production is similar, but the computation of the final attribute is only based on the list of types and the type that comes from the *ApplicationType* symbol.

Although the drawings presented in Figures 4 to 6 have been formally constructed, for those that read the visual grammar it is not necessary to know if attributes are synthesized or inherited, neither the way evaluation rules are built – it is enough to understand the way they are connected to understand the new language semantics. The remaining parts of the formalization follows the same structure as the one presented in this section.

With VisuaLISA we defined a model of IIS*CDesLang PIM concepts. This model can be turned into a valid attribute grammar, and in a straightforward step, we have not only a new language, but also a compiler for the language.

6 Conclusion

Attribute Grammars (AG) are widely used to specify the syntax (by the underlying Context Free Grammar) and the semantics (by the set of attributes and theirs computation rules and contextual conditions) of computer languages. This formalism is well defined and so its usage is completely disciplined; but, more than that, it has the unique property of supporting the specification of syntax and semantics under the same framework. Moreover, an AG can be automatically transformed into a program to process the sentences of the language it defines. AG can also be used to give semantics to other systems, if we look them following that grammar approach.

The research presented in this paper resulted from the collaborative research project between Serbia and Portugal. To formally describe the Integrated Information Systems CASE Tool (IIS*Case) – a model driven software tool that provides IS modeling and prototype generation developed at University of Novi Sad – we define a DSL, named IIS*CDesLang, that encompasses problem domain concepts and rules that are applied in the conceptual IS design provided by IIS*Case. In the paper, we present such a meta-language resorting to a visual programming environment for attribute grammar specifications, named VisualLISA, developed at University of Minho. VisualLISA makes the process of AG development easier and safer; it allows the drawing of the AG productions (grammar rules) in the form of attributed trees decorated with attribute evaluation rules. These visual productions are syntactically and semantically checked for correctness.

As future work, we plan to complete the specification to cover all the IIS*Case and then use the compiler generator system LISA to produce a compiler for IIS*DesLang. On the basis of the grammar specifications and the problem domain knowledge, it is also possible to design tools providing some semantic analyses of the designed specifications and further assist designers in raising the quality of their work. Two characteristic examples are domain compatibility analysis and check constraint equivalence analysis. Finally, we plan to embed a textual editor for IIS*DesLang into IIS*Case and couple the language and generated compiler with IIS*Case repository.

Acknowledgments. A part of the research presented in this paper was supported by Ministry of Science and Technological Development of Republic of Serbia, Grant TR-13029, Title: *A Development of an Intelligent Environment for the Design and Implementation of Information Systems*.

References

1. Aleksić, S., Luković, I., Mogin, P., Govedarica, M.: A Generator of SQL Schema Specifications. Computer Science and Information Systems (ComSIS), Consortium of Faculties of Serbia and Montenegro, Novi Sad, Serbia, ISSN:1820-0214, Vol.4, No. 2, 79–98. (2007)
2. Banović J.: An Approach to Generating Executable Software Specifications of an Information System. Ph.D. Thesis (currently in final stage). University of Novi Sad. (2010)
3. Deursen van, A., Klint, P. Visser, J.: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, Association for Computing Machinery, USA, Vol. 35, No. 6, 26–36. (2000)
4. Pereira, M. João, Mernik, M., Cruz, D., Rangel Henriques, P.: Program Comprehension for Domain-Specific Languages. Computer Science and Information Systems (ComSIS), ISSN:1820-0214, Vol. 5, No. 2, 1–17. (2008)
5. Luković I.: From the Synthesis Algorithm to the Model Driven Transformations in Database Design", In: Proceedings of 10th International Scientific Conference on Informatics (Informatics 2009), Herlany, Slovakia, ISBN 978-80-8086-126-1, 9–18 (2009).

6. Luković, I., Mogin, P., Pavićević, J., Ristić, S.: An Approach to Developing Complex Database Schemas Using Form Types. Software: Practice and Experience, John Wiley & Sons Inc, Hoboken, USA, DOI: 10.1002/spe.820, Vol. 37, No. 15, 1621–1656. (2007)
7. Luković, I., Ristić, S., Aleksic, S., Popović, A.: An Application of the MDSE Principles in IIS*Case. In: Proceedings of III Workshop on Model Driven Software Engineering (MDSE 2008), Berlin, Germany, TFH, University of Applied Sciences Berlin, 53–62. (2008)
8. Luković, I., Ristić, S., Mogin, P., Pavicević, J.: Database Schema Integration Process – A Methodology and Aspects of Its Applying. Novi Sad Journal of Mathematics, Serbia, ISSN: 1450-5444, Vol. 36, No. 1, 115–150. (2006)
9. Mernik, M., Heering, J., Sloane, M. A.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR), Association for Computing Machinery, USA, Vol. 37, No. 4, 316–344. (2005)
10. Rangel Henriques, P., Pereira, M. João, Mernik, M., Lenič, M.: Automatic Generation of Language-based Tools. Workshop on Language, Descriptions, Tools and Applications under ETAPS'02 (LDTA2002), Grenoble, France, April 2002.
11. Pereira, M. João, Mernik, M., Cruz, D., Rangel Henriques, P.: VisualLISA: a Visual Interface for an Attribute Grammar based Compiler-Compiler, In proceedings of 2nd Conference on Compilers, Related Technologies and Applications (CoRTA08), IPB, Bragança, July 2008.
12. Oliveira, N. Rangel Henriques, P. Cruz, D. Pereira, M. João: XAGra - An XML Dialect for Attribute Grammars, In Proceedings of Conferene on XML and Associated Technologies and Applications (XATA09) under INForum'09 - Simpósio de Informática, FCT-UL. (2009)
13. Oliveira, N. Pereira, M. João, Rangel Henriques, P. Cruz, D. Kramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications (ComSIS), Lukovic, I. and Leitão A, Slivnik B. (Guest Eds.), ISSN:1820-0214, Vol. 7, No. 2, 265–289. (2010)
14. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. Compiler Contruction, 1–4. (2002)
15. Knuth, D. E.: Semantics of Context-free Languages. Theory of Computing Systems, Vol 2, No. 2, 127–145. (1968)
16. Krahn H., Rumpe B., Völkel S.: Roles in Software Development using Domain Specific Modelling Languages, In Proceedings of 6th OOPSLA Workshop on Domain-Specific Modeling, Portland, USA, 150–158. (2006)
17. Jouault F., Bézivin J.: KM3: a DSL for Metamodel Specification, In Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, Springer LNCS 4037, 171–185. (2006)
18. Meta-Object Facilty: http://www.omg.org/mof/
19. The Generic Modeling Environment: http://www.isis.vanderbilt.edu/Projects/gme/
20. MetaCase Metaedit+: http://www.metacase.com/
21. Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/