UDC 007:681.32, DOI: 10.2298/CSIS1002359L

A Tool for Modeling Form Type Check Constraints and Complex Functionalities of Business Applications

Ivan Luković[1], Aleksandar Popović[2], Jovo Mostić[1], and Sonja Ristić[1]

[1] University of Novi Sad, Faculty of Technical Sciences,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia
{ivan, sdristic}@uns.ac.rs, jovom@t-com.me
[2] University of Montenegro, Faculty of Science,
Džordža Vašingtona bb, 81000 Podgorica, Montenegro
aleksandarp@rc.pmf.ac.me

Abstract. IIS*Case is a software tool that provides information system modeling and prototypes generation. At the level of platform independent model specifications, 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. It also provides conceptual modeling of business applications. In the paper, we present new concepts and a tool embedded into IIS*Case, that is aimed at supporting specification of check constraints. We present a domain specific language for specifying check constraints and a tool that enables visually oriented design and parsing check constraints. Also, we present concepts and a tool that is aimed at supporting specification of complex (i.e. "nonstandard") functionalities of business applications. It is provided visually oriented and platform independent specification of business application functions.

Keywords: Information system design; Platform Independent Models and Model Driven Software Development; Check constraint specification; Function specification.

1. Introduction

Integrated Information Systems CASE Tool (IIS*Case) is a software tool aimed at assisting the information system (IS) design and at generating executable application prototypes. Currently, IIS*Case 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. By this approach, the software development process provided by IIS*Case is, in general, evolutive and incremental. We believe that 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 the paper [11] we considered the application of the model-driven software engineering (MDSE) principles in IIS*Case. 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. Such a model may be classified as a Platform-Independent Model (PIM) of the MDA pattern ([9], [16], [17], [21], [22], [23]). 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.

In the paper [1] we argued that IIS*Case and our approach are suitable for end-user development (EUD), as it was considered in [3], [4], [20], and [25]. Besides, there are many EUD approaches and tools that provide the assistance to designers and end-users in creating IS specifications. One of them is presented in [24]. We also considered IIS*Case as a tool from the class of domain oriented design environments (DODE), as it is defined in [20]. In [1] we also 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.

A case study illustrating main features of IIS*Case and the methodological aspects of its usage is given in [10], and accordingly we do not repeat the same explanations here. Apart from [1], [10] and [11], detailed information about IIS*Case may be found in several authors' references, as well as in [15] and [19]. The methodological approach to the application of IIS*Case is presented in more details in [13], while an approach to the formal specification of database constraints provided by IIS*Case is presented in [12].

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. An SQL generator is embedded into IIS*Case. It provides further transformation of database schema into the platform specific SQL/DDL code, for various target DBMS platforms [1]. It is an example of model-to-code transformations provided by IIS*Case. Apart from the generation of key, unique, not null, and native referential integrity constraints, SQL Generator also provides the implementation of the default, partial and full referential integrity constraints, and the selection of an appropriate action from the set {No Action, Cascade, Set Default, Set Null}. It also provides the implementation of the inverse referential integrity constraints [1].

Previous versions of IIS*Case did not provide formal specification of check constraints, at all. Research efforts presented in this paper were directed toward introducing new concepts and a tool that enable a designer to formally specify and validate such constraints. An important expectation was to introduce new concepts that are platform independent, so as to provide formal specification of check constraints at the abstraction level of PIMs.

In the paper we present a domain specific language (DSL) aimed at defining check constraints at the level of PIMs. By means of this language, a designer may specify logical expressions of an arbitrary complexity for validating attribute values. The language provides a recognition and usage of other necessary PIM concepts embedded into IIS*Case, and therefore helps a designer in specifying expressions using problem domain concepts, as it is considered in [6], [8] and [14]. Besides, the language does not comprise any platform specific concepts, so check expressions are created at high abstraction level. In the paper we also present a tool aimed at specifying and parsing check constraints in a visually oriented way.

By this, in the process of database constraint design, we provide designers a possibility to concentrate mainly on the constraint semantics in a problem domain, instead of wasting time on their formal specification and validation. To achieve this goal, we need the appropriate DSLs and PIM concepts embedded into IIS*Case that are mostly problem oriented, instead of using relational data model concepts that are more technology specific, or even SQL DDL syntax, which is fully technology oriented programming language. Therefore, SQL DDL normally may be used to implement database schema specifications under a DBMS, but should not be directly used in the design of IS specifications, particularly at the conceptual level, i.e. at the abstraction level of PIMs.

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 a program code of business applications. In this way, fully executable application prototypes are generated. For these purposes, User Interface Markup Language (UIML) and Java Render by Harmonia Incorporation® are chosen programming and run-time environment [19]. Such a generator is also an example of model-to-code transformations provided by IIS*Case and its development is almost finished.

Transaction programs of business applications may often comprise not only basic operations, but also more complex functionalities that cannot be expressed by a sole retrieve, insert, update, or delete operation. Such functionality may comprise complex calculations, as well as series of database operations. Therefore, such functionality we call specific, complex, or "nonstandard" application functionality. Besides, specifications of check constraints may reference various complex functions that should be specified also formally, i.e. in the same way as complex application functionality.

Basic data operations such as retrieve, insert, update and delete are common for various problem domains and can be easily specified by means of IIS*Case concepts. However, business applications from various problem domains usually comprise complex functionalities. If such functionalities would not be embedded into the PIM of a software system being designed, a programmer has to create latter a program code of such functionalities, or at least has to amend a generated program code, "by hand". In this way, complex functionalities are modeled at the lowest level of abstraction, by means of a target programming language which is always platform specific. As a rule, such created program code becomes unsynchronized with the initial PIM models of the system during the time. As a consequence, the operational maintenance of such systems becomes more difficult, with a lot of problems arising during the software exploitation.

Previous versions of IIS*Case did not provide formal specification of complex application functionalities or functions referenced in check constraints, at the level of PIMs. Research efforts presented in this paper were directed toward introducing new concepts and a tool that enable a designer to formally specify complex functionalities. An important expectation was to introduce new concepts that are platform independent, so as to provide formal specification of complex functionalities at the abstraction level of PIMs.

In the paper we also present concepts and a repository oriented tool aimed at the specification of functions at the level of PIMs. The name of the tool is *Function Specification Editor* or *Function Editor* for short. By means of *Function Editor* a designer may specify functions of an arbitrary complexity. It provides usage of necessary PIM concepts embedded into IIS*Case, and helps a designer in specifying functions using not only programming language concepts, but also problem domain concepts in a certain extent. Besides, *Function Editor* does not comprise any platform specific concepts, so functions are specified at high abstraction level. Also, it provides specifying functions completely in a visually oriented way. On the basis of *Function Editor* and the appropriate repository definitions used by *Function Editor* as a part of IIS*Case, it is possible to create a Domain Specific Language (DSL) for specifying business functions at the level of PIMs, as it is considered in [6], [8] and [14].

Apart from Introduction and Conclusion, the paper consists of six sections. In Section 2 we briefly describe main concepts of the IIS*Case tool that are important for specification of check constraints and function specifications. Check constraint expressions are introduced in Section 3, where grammar rules are presented. The main features and functionalities of the *Expression Editor* tool are presented in Section 4, while the implementation details concerning parsing of check expressions are presented in Section 5. Function specifications and related concepts are introduced in Section 6, while the main features and functionalities of the *Function Editor* tool are presented in Section 7.

2. Preliminaries

At the abstraction level of PIMs, IIS*Case currently provides conceptual modeling of database schemas and software applications of an IS. Starting from such PIM models as a source, a chain of transformations is performed so as to obtain executable program code of software applications and database SQL/DDL scripts for a selected target platform. The similar idea may be found also in [2]. For the purpose of readability, in this section we briefly describe main modeling concepts of IIS*Case that are used at the abstraction level of PIMs and have an influence on the specification of check constraints, as well as on the specification and referencing of functions defined in IIS*Case repository.

A *form type* is the main modeling concept in IIS*Case ([10], [12], [15]). It generalizes document types, i.e. screen forms that users utilize to communicate with an information system. The similar concept of the form type may be found in [5] and [7], as well as in many other references. Using the form type concept in IIS*Case, a designer specifies screen or report forms of transaction programs and, indirectly, specifies (i) an initial set of attributes and constraints, (ii) basic functionalities of future transaction programs and (iii) components of their UI. Each particular business document is observed as an instance of a form type. A form type concept, as well as related concepts of a domain and attribute, is platform independent. Here, we use a notion of the form type instead of a document type, because it is always a structure defined at the abstraction level of schema. It represents not only a layout structure (i.e. screen or a report form) of a document, but also a set of database schema attributes and constraints embedded into a future screen or a report form of an IS transaction program.

A form type is a named tree structure, whose nodes are called component types. Each *component type* is identified by its name in the scope of the form type, and has nonempty sets of attributes and keys, and a set of unique constraints that may be empty. Besides, to each component type must be associated a set of allowed database operations. It must be a nonempty subset of the set of "standard" operations {retrieve, insert, update, delete}. Each attribute of a component type is chosen from the set of all information system attributes.

Attributes are globally identified only by their names. IIS*Case imposes strict rules for specifying attributes and their domains. Attributes in IIS*Case are classified as elementary or derived. An attribute is elementary if it represents values given by end-users directly. Otherwise, it is derived. Values of a derived attribute are generated (i.e. calculated) from the values of the other attributes, by applying some algorithm. Such algorithms in IIS*Case are expressed by a concept of *function*. Therefore, a specification of a derived attribute must reference at least one previously defined (elementary or derived) attribute, and at least one function that is used for calculating its values.

Domains in IIS*Case are also globally identified only by their names. They are classified as primitive and user-defined. Primitive domains are defined "per se" as primitive data types. They are predefined into the repository of IIS*Case. An initial collection of primitive domains stored in the repository may be customized by adding, changing, or even removing specifications of primitive domains. Each user-defined domain in IIS*Case is created by referencing a primitive domain, or an already existing user-defined domain. In this way, user-defined domains are derived from primitive or previously created user-defined domains. There are four derivation rules that may be applied to create a user-defined domain from the existing domains: a) inheritance rule, b) tuple rule, c) set rule, and d) choice rule. A domain obtained by one of the aforementioned rules is called inherited, tuple, set, or choice domain, respectively. Tuple, set, or choice domains are also called complex domains. Recursive multiple application of the aforementioned rules is allowed.

Inherited domain inherits all the properties from its source (parent) domain. If a domain D is defined by the inheritance rule from the parent domain D_s, we denote it by $D = Inherits(D_s)$. Besides, a separate check expression is to be assigned to an inherited domain. Therefore, it is more or at least equally restrictive as its parent domain. If check expressions are defined for both inherited and its parent domain, in evaluation they are connected by the logical AND operator. Consequently, in a recursive application of the inheritance rule, all the domain check expressions in a hierarchy are connected by the logical AND operators.

Tuple domain represents tuples (records) of values over source domains. Therefore, it is defined as a structure $D = Tuple(A_1 : D_1,..., A_n : D_n)$, where D is a tuple domain, and for each $i \in \{1,...,n\}$, $(A_i : D_i)$ is a tuple item, i.e. a member, where A_i is an attribute with an associated source domain D_i.

Set domain represents values that are sets, each over the same source domain. Therefore, it is defined as a structure $D = Set\{D_s\}$, where D is a set domain, and D_s is a source domain.

Choice domain represents values over exactly one of the source domains. Therefore, it is defined as a structure $D = Choice(A_1 : D_1,..., A_n : D_n)$, where D is a choice domain, and for each $i \in \{1,...,n\}$, $(A_i : D_i)$ is a choice item, i.e. a member, where A_i is an attribute with an associated source domain D_i.

Check constraints in IIS*Case may be specified at the level of a domain, attribute or a component type of a form type. A check constraint associated to a domain or attribute is used to specify a logical condition constraining allowable values of a sole attribute. A check constraint associated to a component type is used to specify a logical condition constraining some values of each component type instance. Logical conditions of the check constraints may also reference functions, defined in IIS*Case repository.

3. Check Expressions

The quality of a whole database schema is substantially influenced by the quality of constraint specifications. It is very important to define these specifications at early stages of database schema design process, at abstraction level of PIMs, if possible. IIS*Case provides specification of various types of constraints, such as domain, not null, key and unique constraints, as well as various kinds of inclusion dependencies, at the abstraction level of PIMs.

Commercial CASE tools that provide modeling conceptual database schema specifications by means of Entity-Relationship (ER) data model and their transforming into the relational data model either provide only partial specifications of check constraints at the conceptual level, and/or provide a usage of standard SQL syntax for that purposes. Accordingly, check constraints may be fully defined only at the level of an implementation database schema specification, expressed commonly by relational data model and SQL syntax. For example, Oracle Designer does not allow all kind of check constraints to be formally defined at the level of an ER database schema. Sybase Power Designer provides a usage of SQL syntax for that purposes. On the contrary, check constraints in the IIS*Case tool are defined at the level of a conceptual database schema as a PIM model, which is expressed by a set of created form types. For these purposes, we developed a DSL to create check expressions of various complexity, in a platform independent way. Such a DSL and a tool embedded into IIS*Case enable a designer to specify check constraints using problem domain concepts, in a visually oriented way.

A check expression is a logical expression. In general, it may include attribute references, arithmetic, comparison and logical operators, as well as function calls. As implemented at the level of a target DBMS, it is usually evaluated in a ternary logic as a value from the set {*true*, *false*, *unknown*}, where *true* means that an expression is valid, *false* that it is violated, and *unknown* that it is neither valid nor violated. The value *unknown* is possible to obtain whenever there are null (missing) values of attributes in the evaluation of an expression.

By means of the DSL embedded into IIS*Case, check expressions may be specified at the level of a (i) domain, (ii) attribute or (iii) component type of a form type, in a similar way.

3.1. Domain Check Expressions

IIS*Case provides a "universal" set of all domains of a project as a whole. Domains in IIS*Case are used to express domain constraints, as it is proposed in [12]. Each specification of a user-defined domain allows defining a check expression, as a property of the domain specification. Such check expressions are named domain check expressions.

A formal specification of the grammar for domain check expressions is shown in Table 1, in the Extended Backus-Naur Form (EBNF) notation.

Table 1. Specification of the grammar for domain check expressions

```
Exp = Exp bin_operator Exp | un_operator Exp | Primary_Exp;
Primary_Exp = constant | value ['.' fieldName] |  function_name
'(' [Exp_List] ')' | '(' Exp ')';
Exp_List = Exp { ',' Exp_List};
```

The list of standard operators includes the following ones:

- Additive (+, -),
- Multiplicative (*, /),
- Comparison (<, <=, >, =>),
- Equality (==, !=),
- Concatenation (||),
- Boolean (NOT, AND, OR, XOR, =>),
- Inclusion (IN), and
- Pattern matching (LIKE).

All the operators and parentheses are introduced with the common meaning and priorities when applying the rules for evaluation of expressions.

Apart from introducing standard arithmetic, string, comparison and logical operators existing in all general-purpose languages, we decided also to introduce the operators LIKE and IN, which are common in database languages, like SQL. In this way, the language for check expressions becomes more problem oriented.

The grammar in Table 1 also provides function calls by referencing the appropriate function names. It is allowed to reference only the functions existing in the IIS*Case repository. It is supposed that both built-in and user-defined functions are stored in the repository. IIS*Case also provides a specialized DSL and a visually oriented tool for specifying various functions in a project. By this, it is possible to specify function header, a list of formal parameters, return value, all local declarations, function body and the exception handler in a structural way. Functions are specified by means of the technology independent concepts, at the abstraction level of PIMs, as it is presented in Sections 6 and 7.

The grammar in Table 1 allows the use of constants in check expressions. The common rules for specification and interpretation of constants are applied, and accordingly we do not describe them in more detail.

The only variable symbol allowed in domain check expression is *value* symbol (VALUE). VALUE denotes any value for which a domain check expression is validated.

Only in check expressions associated to a tuple or choice domain it is possible to qualify VALUE by the attribute name of an item. Therefore, VALUE.Ai denotes a value of a tuple or choice member (A_i : D_i), while nonqualified VALUE denotes a complete tuple or a choice value.

Example 1. A domain check expression for a numeric domain *DGRADE* is given:

```
VALUE >= 5 AND VALUE <= 10.
```

It constrains allowable values of *DGRADE* to the interval from 5 to 10. □

Example 2. A domain check expression for a string domain *DPHONE* is given:

```
VALUE LIKE '5%' AND StrLen(VALUE) == 7.
```

It constrains allowable values of *DPHONE* to exactly the 7 character long strings, beginning with '5'. StrLen is a function call that references a function already specified in the IIS*Case repository. □

Example 3. A domain check expression for a string domain *DSEMESTER* is given:

```
VALUE IN {'I', 'II','III','IV','V','VI','VII','VIII','IX', 'X'}.
```

It constrains allowable values of *DSEMESTER* to the list of string values specified after the inclusion operator IN. □

Example 4. A tuple domain *DDATE* is defined as *DATE* = *Tuple*(*DAY* : *INTEGER*, *MONTH* : *INTEGER*, *YEAR* : *INTEGER*), where INTEGER is primitive domain. A domain check expression for a tuple domain *DDATE* is given:

```
VALUE.DAY <= 31 AND VALUE.DAY >= 1.
```

It constrains allowable values of DAY member to the interval from 1 to 31. □

3.2. Attribute Check Expressions

IIS*Case provides a "universal" set of all attributes of a project as a whole. According to the universal relationship existence assumption (URSA) adopted from the relational data model, each attribute in IIS*Case is uniquely identified only by its name. Exactly one domain must be associated to each attribute in a project. In this way, allowable values of an attribute are constrained by the appropriate domain constraint.

IIS*Case allows defining a check expression as a property of the attribute specification. Such check expressions are named attribute check expressions. Our DSL has the appropriate grammar rules for specification of attribute check expressions.

Suppose that we have an attribute *A* to which a domain *D* is associated. We denote it as (*A* : *D*). If a domain check expression is associated to *D*, then each attribute *A* with the associated domain *D* inherits its domain check expression. Besides, if we have an attribute check expression assigned to an attribute *A*, and a domain check constraint assigned to *D*, where (*A* : *D*) holds, in evaluation they are connected by the logical AND operator. Obviously, if we have (possibly a recursive) application of the inheritance rule for the domain *D*, all the domain check expressions in a hierarchy are connected alongside with the attribute check expression by the logical AND operators.

A formal specification of the grammar for attribute check expressions is shown in Table 2, in EBNF notation. It is almost identical to the grammar specification for domain check constraints given in Table 1. The only difference is in the following. If we specify the attribute check expression for an attribute with the name *A*, the only variable symbol allowed in attribute check constraints, which may replace `attName`, is *A*. It is with the same meaning as it is the symbol VALUE in domain check expressions. Analogously to the domain check constraints, we may additionally qualify *A* in the case of a tuple or choice domain associated to *A*. Therefore, A.Ai denotes a value of a tuple or choice member (A_i : D_i), while nonqualified A denotes a complete tuple or a choice value.

Table 2. Specification of the grammar for attribute check expressions

```
Exp = Exp bin_operator Exp | un_operator Exp | Primary_Exp;
Primary_Exp   =   constant   |   attName   ['.'   fieldName]   |
function_name '(' [Exp_List] ')' | '(' Exp ')';
Exp_List = Exp { ',' Exp_List};
```

Example 5. An attribute check expression for a numeric attribute *GRADE* is given:

```
GRADE >= 6.
```

It constrains allowable values of *GRADE* to be greater or equal 6. If (*GRADE* : *DGRADE*) holds, where *DGRADE* is a domain from Example 1, then this check expression is connected to the one from Example 1 by the operator AND. Consequently, allowable values of *GRADE* are constrained to the interval from 6 to 10. □

3.3. Component Type Check Expressions

In IIS*Case, a form type is a hierarchical tree structure of component types, each of them having nonempty sets of attributes and keys, and a possibly empty set of unique constraints. Each attribute of a component type is selected from the set of all attributes of a project, i.e. from the IIS*Case repository. Therefore, it inherits all its constraints defined at the levels of the appropriate attribute and domain specifications.

IIS*Case also allows defining a check expression as a property of the component type specification. Such check expressions are named component type check expressions. Our DSL has the appropriate grammar rules for specification of component type check expressions.

The main purpose of domain and attribute check expressions is to constrain allowable values of a sole attribute. On the contrary, component type check constraints are used to specify logical conditions that constrain a tuple of values representing each component type instance.

A formal specification of the grammar for component type check expressions is shown in Table 3, in EBNF notation. It is almost identical to the grammar specification for attribute check constraints given in Table 2. The only difference is in the following. If we specify the component type check constraint for a component type *N*, we may use as variable symbols that are to replace `cmpattName`, any of attributes from the component type *N*, as well as any of attributes from any superordinated component type in a form type hierarchy.

Table 3. Specification of the grammar for component type check expressions

```
Exp = Exp bin_operator Exp | un_operator Exp | Primary_Exp;
Primary_Exp   =   constant   |   cmpattName   ['.'   fieldName]   |
function_name '(' [Exp_List] ')' | '(' Exp ')';
Exp_List = Exp { ',' Exp_List};
```

Analogously to the attribute check constraints, we may additionally qualify variable *A* in the case of a tuple or choice domain associated to *A*. Therefore, A.Ai denotes a value of a tuple or choice member (A_i : D_i), while nonqualified A denotes a complete tuple or a choice value.

Example 6. In Fig. 1 it is presented a form type *Student Records*. The form type is structured as a tree having two component types, STUDENT and GRADES, which are graphically represented by rectangles. The component type attributes are shown in italic letters. The key attribute of each component type is underlined by a solid line, whereas the attribute of a uniqueness constraint is underlined by a dashed line. Allowed operations for both component types are shown in small rectangles in the upper-right corners.

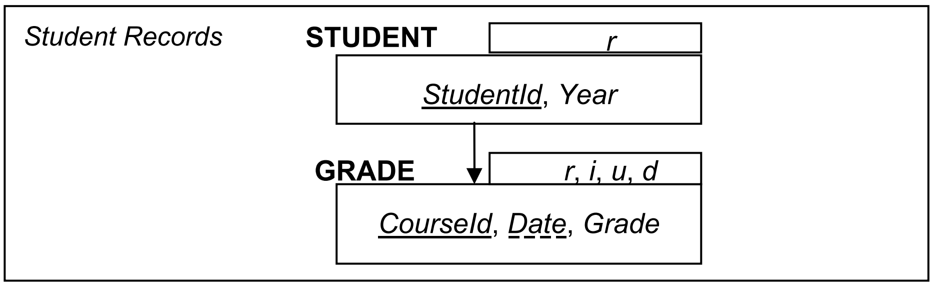


Fig. 1. A representation of the form type *Student Records*.

A check expression for the *GRADES* component type is given:

```
(Year IN {1, 2, 3} => Grade IN {1, 2, 3, 4})
                          AND (Year IN {4, 5} => Grade IN {4, 5}).
```

It constrains the possible combinations of values for Year and Grade. If Year is 1, 2, or 3, Grade must be 1, 2, 3, or 4, and if Year is 4 or 5, Grade must be 4 or 5. □

4. Check Expression Editor

Check Expression Editor, or *Expression Editor* for short, is a tool that we developed and embedded into IIS*Case. It is aimed at specification and validation of check expressions. It may be called from the

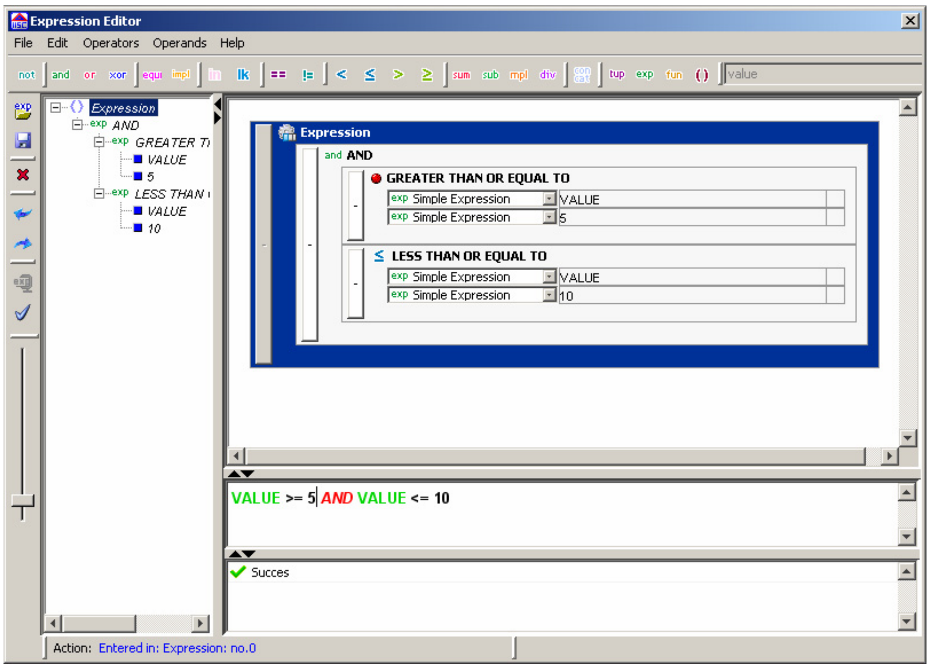
- Domain specification form of IIS*Case, if a domain check constraint need to be defined;
- Attribute specification form of IIS*Case, if an attribute check constraint need to be defined; or
- Component type specification form of IIS*Case, if a component type check constraint need to be defined.

By this, *Expression Editor* will support the appropriate check expression grammar, in a context-sensitive way.

Expression Editor provides two options for specification of check expressions: (i) guided, by means of a *Visual Editor*, and (ii) "free form", by means of a *Text Editor*. The first option is more suitable for less experienced users, not knowing the precise grammar rules and therefore needing a guide in specifying check expressions. The second one is more suitable for more experienced users, well knowing the precise grammar rules, and wishing to be as fast as possible in specifying check expressions. The main screen form of *Check Expression Editor* is presented in Fig. 2. *Visual Editor* is positioned in the center, while *Text Editor* is positioned in the bottom of the main form of *Expression Editor*.

Text Editor provides direct writing check expressions in a free form way. Besides, it supports context-sensitive syntax highlighting, as well as standard text processing commands such as: cut, copy, undo, etc. These commands are included in the *Edit* submenu of the main menu, and also in the toolbar positioned on the left hand side of the main form. Also, the toolbar comprises a command for performing expression validation.

By means of *Visual Editor*, check expressions are modeled by building the expression trees. Expression tree navigator, as a part of *Visual Editor*, is positioned on the left hand side of the main form from Fig. 2. Each node of an expression tree represents a subexpression, while the root node represents the main expression. Non-leaf nodes are named complex nodes, because they represent complex expressions, for example the expressions enclosed by parentheses, or operator inclusions. Leaf nodes are named simple nodes, because they correspond to simple, i.e. primary expressions, like constants, variables (such as VALUE or attribute names), or function calls.



```
VALUE >= 5 AND VALUE <= 10
```

```
Succes
```

```
Action: Entered in: Expression: no.0
```

Fig. 2. Main screen form of *Expression Editor*

Visual Editor provides all common functions for editing an expression tree. These are: inserting, deleting, moving, and editing a node. The last function is available only for leaf nodes, representing simple expressions.

When a user wants to insert a complex node, he or she has to select a language operator or the parentheses symbol from the main toolbar. Each operator of the language is represented by an appropriate iconic button in the main toolbar.

Inserting a simple node into the expression tree is performed by selecting the *exp* command from the main toolbar. After selecting the *exp* command, a node is inserted and a textbox for specifying the simple expression appears within the node. According to grammar rules, simple expressions may be constants from a domain, variables, or function calls. A combo box positioned on the upper-right corner is aimed to assist a user to select an appropriate attribute, or a function from the IIS*Case repository.

Example 7. Suppose the following domain complex expression has to be specified by means of *Visual Editor*:

```
VALUE >= 5 AND VALUE <= 10.
```

A user needs first to insert a complex node for AND operator, and then two descendant complex nodes, one for ">=" and the other for "<=" operators.

Below the ">=" complex node, he or she needs to insert two simple nodes, one for variable VALUE, and the other for a constant 5. In a similar way, two simple nodes are to be defined below the "<=" complex node, one for VALUE, and the other for 10. □

Expression Editor always keeps *Visual Editor* and *Text Editor* synchronized. When a user creates and validates an expression by means of *Visual Editor*, the expression will be also shown in its full syntax in *Text Editor*. Also, when a user creates and validates an expression by means of *Text Editor*, the corresponding expression tree will be shown in *Visual Editor* automatically.

5. Validation of Check Expressions

Expression Editor provides validation of check expressions. Parser is created by means of the ANTRL 4.0 tool. ANTRL enables a user to formally specify grammar. Furthermore, it supports transformation of grammar specifications into the program code of a parser for target programming environment. As a result, it is obtained a recursive-descent parser expressed in a program code that is human-readable and easily customizable. [18].

According to the specified language definition presented in Section 3, ANTLR is used to generate Java program code of a parser that checks whether sentences created by *Expression Editor* conform to the language specification.

ANTLR generally provides amending grammar rules by adding source program code, i.e. code snippets to the grammar definition. Then, such code snippets are inserted into the program code of a generated parser, "as is". In our case, grammar rules for check expressions are amended by inserting code snippets that translate input sentences into an XML specification, and perform some semantic analysis, at the same time. In this way, apart from syntax validation, *Expression Editor* provides some semantic analysis. For example, check constraints may contain variables that reference members of a tuple or choice domain. The semantic analyzer verifies if reference to a tuple or choice member is valid, by seeking the appropriate domain specifications from the IIS*Case repository. Currently, type checking is not supported, at all. It is because the domain specification in our repository model still does not provide specification of allowed operators over a domain.

Example 8. In Table 4 two grammar rules for domain check expressions are presented. These rules contain code snippets that provide performing semantic analysis and creating a node in the appropriate XML specification. The grammar rules are specified in ANTLR notation.

Table 4. Grammar rules for domain check expressions containing code snippets

```
sentence returns [String val]
@init{ tmp = "";}
:
tmp = expression
{val="<block name=\"Expression\" group=\"1\">"+ tmp+"</block>";
val = val.replaceAll("><",">\n<") + "\n\n";}
;

domain_ref
@init{ tmp = "";}
: value ( '.' tmp = memberName )?
{checkMember(tmp);}
;
```

A code snippet that provides creating a node in the XML specification of a check expression is included in the *sentence* grammar rule in Table 4. It is given as follows:

```
{val = "<block name=\"Expression\" group=\"-1\">" + tmp +
                                                        "</block>";
val = val.replaceAll("><",">\n<") + "\n\n";}
```

A code snippet that provides performing semantic analysis is included in the *domain_ref* grammar rule in Table 4. It is given as follows:

```
{checkMember(tmp);}
```

When member name is identified, the snippet verifies if a reference to a tuple or choice member is valid, by seeking the appropriate tuple or choice domain specifications from the IIS*Case repository. □

Apart from being used for a semantic analysis, XML specifications of check expressions may also be used to provide further necessary transformations of check constraints. Our future research work is oriented towards providing a chain of transformations that result in PSM specifications of check constraints, expressed as the SQL/DDL program code.

The main idea how to design the transformation process from check expressions specified at the level of PIMs to the SQL/DDL program code is as follows. The process should be generally organized in two phases. By our methodology ([10], [13]), in the first phase, a set of form types representing a PIM model of a conceptual database schema is transformed into a relational database schema. Accordingly, all the constraints specified at the conceptual PIM level should be transformed into the equivalent relational database schema constraints. Therefore, each component type check expression specified at the level of a PIM, should be transformed into the one or more appropriate check or extended check expressions ([12]) defined at the level of the corresponding relation schemes. It is an issue how to create and implement an algorithm that will (i) provide inference problem solving for check expressions and (ii) preserve logical equivalency during transformations of component type check constraints. In this phase, domain and attribute check expressions remain unchanged.

A relational database schema generated in the first phase is still technology independent of any particular DBMS. Therefore, in the second phase, it is transformed into the SQL/DDL specification justified to the syntax of a chosen DBMS or ANSI SQL standard ([1]). Accordingly, each check expression defined at the level of a sole domain, attribute or a relation scheme, should be transformed into a corresponding SQL/DDL check constraint. Such a transformation is easily possible because of using a syntax for our check expressions that is very similar to the syntax for expressions in SQL check constraints. It is an issue here how to transform check expressions that contain references to the members of tuple or choice domains if a target DBMS does not support necessary object-relational concepts. On the other hand, with respect to the current level of supporting ANSI SQL standard by commercial DBMSs, extended check constraints in a relational database schema may only be transformed into the SQL code of a target DBMS that includes triggers and stored procedures.

6. Modeling Complex Functionalities in IIS*Case

Software development in IIS*Case is organized through projects. Each project in IIS*Case is further organized trough application systems and represented by a project tree. A set of fundamental specifications, comprising domains, attributes, inclusion dependencies, and program units is associated to each project. Fundamental specifications are independent of any application system given in a project. IIS*Case provides the following program unit concepts from the class of fundamental concepts necessary to express complex application functionalities at the level of PIMs: (i) Function; (ii) Package; and (iii) Event. A part of IIS*Case project tree representing these concepts is presented in Fig. 3.

A concept of a function is used to specify complex functionalities. Functions in IIS*Case are defined at the level of a project, and may be referenced from various IIS*Case specifications. A concept of a function is presented in the following text in more details.

A package is a collection of arbitrary selected functions defined in IIS*Case repository. Usually, packages are organized in a "thematic" way. Depending on a selected layer for the package deployment in multi-tier distributed software architecture, at the level of PIMs, we differentiate between database server, application server and client packages. Database server packages are to be deployed at the database server layer. The analogous is for application server and client packages.

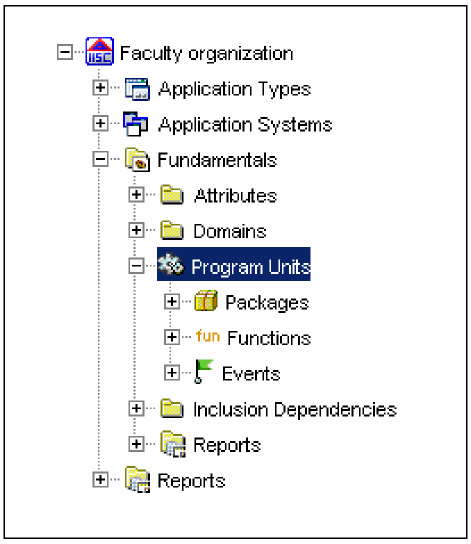


Fig. 3. A part of IIS*Case Project Tree.

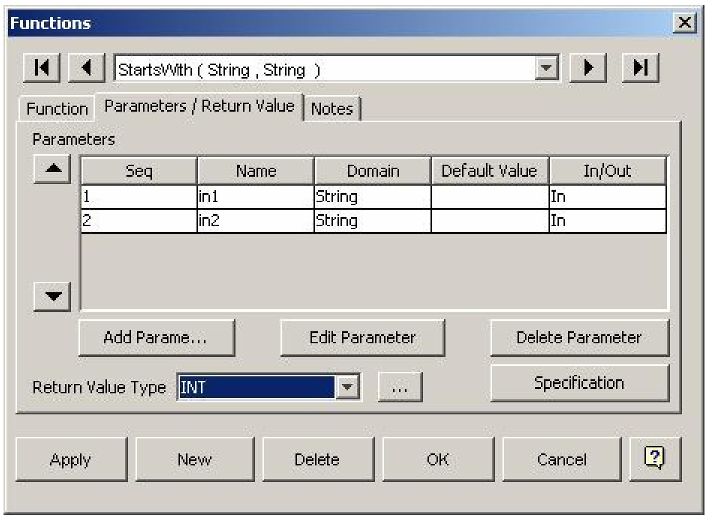
A concept of event is used at the level of PIMs, to represent any software event that may trigger some action under a specified condition. We also differentiate between database server, application server and client events. Database server events may be database triggers or exceptions. Application server and client events may be: keyboard events, mouse events, or exceptions. Each event should be associated to a PIM specification. For example, a database trigger should be associated to a relation scheme. A keyboard event may be associated to a form type, component type, or an attribute of a component type. A concept of event is not fully implemented in IIS*Case yet. Its full implementation is a matter of further research.

A formal specification of a function in IIS*Case includes the following:

- Function name that is unique in the IIS*Case project;
- List of formal parameters (i.e. arguments);
- Return value type; and
- Function body.

In Fig. 4 it is presented the IIS*Case screen form for specifying a function with the list of formal parameters and the return value type. The "Specification" button invokes the *Function Editor* tool aimed at formal specification of the function body. *Function Editor* is presented in the next section.

For each function, an arbitrary number of formal parameters may be defined. Each formal parameter is specified by the following properties: (i) sequence number defining a position of the parameter in the list; (ii) name; (iii) reference to IIS*Case domain defining a data type of a parameter; (iv) default value; and (v) type, where possible parameter types are: input (In), output (Out) and input/output (InOut), with a usual meaning inherited from various programming languages. Return value type is a reference to the domain previously defined in IIS*Case repository.



Seq	Name	Domain	Default Value	In/Out
1	in1	String		In
2	in2	String		In

Return Value Type: INT

Fig. 4. A screen form for specification of a function, its formal parameters and a return type.

Function body is specified by means of PIM concepts that are mostly inherited from the third generation languages, particularly database procedural languages, and structural programming paradigm. Function body is a tree structure comprising blocks, declarations, statements, and comments. We differentiate between execution blocks and declaration blocks. Execution blocks may include nested declaration and execution blocks. In this way, multi-level nesting of blocks is provided. The following concepts are provided for specifying a function body:

- Sequential structures defining sequences of statements, declarations or comments;
- Declaration blocks that represent sequences of various declarations and comments;
- Declarations of local types, variables, constants, functions, cursors and exceptions;
- Execution blocks that represent sequences of embedded blocks, various statements and comments;
- Iteration structures with FOR, DO-WHILE, and WHILE-DO statements;
- Selection structures with IF-THEN-ELSE and ELSEIF-THEN-ELSE statements;
- Exception handler structure with TRY, CATCH, and FINALLY statements;
- Simple statements, like various kind of expressions and assignment statements; and
- Single-line comments denoted as /* */.

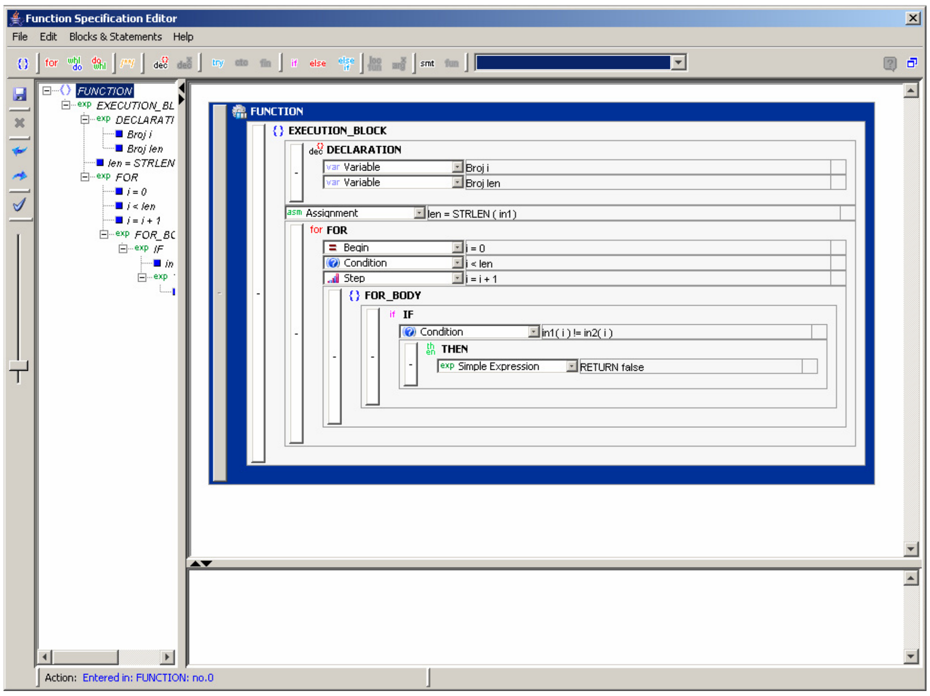
Despite that these concepts are mostly inherited from the third generation languages, they are syntactically independent of any particular programming language. Therefore, function specifications in IIS*Case are platform independent.

A specified function may be referenced many times in the same IIS*Case project. Currently, a function may be referenced in:

- Declarations and expressions of other IIS*Case functions;
- Packages, to express an inclusion of the function into a package;
- Events, to express the activity of an event associated to a PIM specification;
- Logical expressions of domain check constraints, attribute check constraints and component type check constraints; or
- Specifications of derived attributes.

7. The Function Editor Tool

Function Editor is the IIS*Case tool that provides repository based specification of a function body in a visually oriented way. The main screen form of *Function Editor* is presented in Fig. 5.



```
FUNCTION
  EXECUTION_BLOCK
    DECLARATION
      Variable              Broj i
      Variable              Broj len
    Assignment              len = STRLEN ( in1)
    FOR
      Begin                 i = 0
      Condition             i < len
      Step                  i = i + 1
      FOR_BODY
        IF
          Condition         in1( i ) != in2( i )
          THEN
            Simple Expression   RETURN false
```

```
FUNCTION
  EXECUTION_BL
    DECLARATI
      Broj i
      Broj len
    len = STRLEN
    FOR
      i = 0
      i < len
      i = i + 1
      FOR_BC
        IF
          in
```

Fig. 5. The main screen form of *Function Specification Editor*.

By means of *Function Editor*, a function body is represented as a tree, whose nodes represent blocks, declarations, statements, or comments. *Tree Structure Navigator* is placed on the left hand side of the *Function Editor* screen form from Fig. 5, while the complete specification of a function body is represented in a panel placed in the central part of the screen form from Fig. 5. At the bottom of the screen form a message panel is placed.

Function Editor provides common tree operations, like creating a new node, removing an existing node, or reconnecting (cut & paste) a node in the tree. A notion of a current node in *Tree Structure Navigator* is recognized and all the tree operations are performed in the context of the current node. The current node is marked by a different color. Tree operations are available from the main menu, horizontal and vertical toolbars, as well as from the right-mouse-click context menu.

Creating a new node is a context sensitive operation. It is performed by selecting an appropriate toolbar option or "Blocks & Statements" menu option. A designer may select only one of the options that are available in the context of the current node. In this way, he or she specifies the type of the node being created. A list of all possible node types with their descriptions is given in Table 5 included in Section 10, Appendix.

By a context sensitive selection of options for node types that are available in the context of current node, *Function Editor* assists a designer in creating valid function specifications. For example, if the current node represents a FOR statement, a creation of ELSE descendant node is unavailable. According to common structural programming rules imposed by general purpose procedural languages, *Function Editor* only allows the combinations of node types that make sense in specifying a function body. In this way, *Function Editor* just allows building valid structures of a function body.

Besides, *Function Editor* also provides a syntax and semantic analysis tool. A designer may use the tool during the whole process of creating function specifications, just by selecting an appropriate toolbar option. The syntax analysis also checks validity of the structure of function body specification. As it concerns semantic analysis, currently *Function Editor* only checks variable and constant declarations, if specified data type is a reference to a domain specification from the IIS*Case repository. Type checking is not supported, at all. It is because the domain specification in our repository model still does not provide specification of allowed operators over a domain.

8. Conclusion

Commercial CASE tools that provide modeling conceptual database schema specifications by means of ER data model and their transforming into a relational data model either provide only partial specifications of check constraints at the conceptual level, and/or provide a usage of standard SQL syntax for that purposes. Therefore, check constraints are usually fully defined at the level of an implementation database schema. On the contrary, in our approach, check constraints in the IIS*Case tool are defined at the level of a conceptual database schema as a PIM model. For these purposes, we developed a DSL and the *Check Expression Editor* tool to create and parse check expressions defined in a platform independent way. In this way, a designer may specify check constraints using problem domain concepts, in a visually oriented way.

Besides, by our approach, function specifications, which may be referenced from check constraint expressions as well as from the other IIS*Case specifications, are defined at the level of a conceptual specification of an IS, as a PIM model. For these purposes, we developed a specialized tool, named *Function Editor*, by means of it is possible to create and analyze function specifications defined in a platform independent way. In this way, a designer may specify functions using not only programming concepts, but also problem domain concepts, in a visually oriented way.

Among all, our current or future research and development efforts are oriented towards the following:

- Development of the algorithms providing transformations of check constraint specifications created at the level of form types as PIMs, to the equivalent specifications at the level of an implementation database schema (usually expressed by the relational data model), and then to the executable PSM specifications expressed as the SQL/DDL program code;
- Development of a DSL for an equivalent representation of the current repository based function specifications at the level of PIMs;
- Extensions of the IIS*Case repository definition and the appropriate specifications (like event specifications) by new concepts, so as to make better foundation for (i) semantic analysis of check constraint expressions; and (ii) using function specifications in specifying business application logic, as well as their syntax and semantic analysis;
- Development of the algorithms providing transformations of function specifications created at the level of PIMs, to the equivalent executable PSM specifications expressed in a target programming environment and in the context of generated business applications; and
- Using the Meta-Object Facility Specification (MOF) in order to raise our repository based DSL specifications at meta-meta abstraction level.

Acknowledgment

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. ARTech. *DeKlarit*^{TM} (TheModel-Driven Tool for Microsoft Visual Studio 2005). Chicago, USA (June, 2009). [Online]. Available: http://www.deklarit.com.
3. Berti, S., Paterno, F., Santoro, C.: Natural Development of Ubiquitous Interfaces. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 63-64. (2004)
4. Burnett, M., Cook, C., Rothermel, G.: End-User Software Engineering. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 53-58. (2004)
5. Choobinch, J., Mannio, V. M., Nunamaker, F. J., Konsynski, R. B.: An Expert Database Design System Based on Analysis of Forms. IEEE Transactions on Software Engineering, Vol.14, No 2, 242-253. (1988)
6. 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)
7. Diet, J., Lochovsky, F.: Interactive Specification and Integration of User Views Using Forms. In: Proceedings of the Eight International Conference on Entity-Relationship Approach, Toronto, Canada, 171-185. (1989)
8. João Pereira, M., Mernik, M., Cruz, D., Rangel Henriques, P.: Program Comprehension for Domain-Specific Languages. Computer Science and Information Systems (ComSIS), Vol. 5, No. 2, 1-17. (2008)
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise, Addison Wesley. (2003)
10. 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)
11. 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)
12. Luković, I., Ristić, S., Mogin, P.: On The Formal Specification of Database Schema Constraints. In: Proceedings of I Serbian – Hungarian Joint Symposium on Intelligent Systems, Subotica, Serbia, 125-136. (2003)
13. Luković, I., Ristić, S., Mogin, P., Pavicević, J.: Database Schema Integration Process – A Methodology and Aspects of Its Applying. Novi Sad Journal of Mathematics (Formerly Review of Research, Faculty of Science, Mathematic Series), Serbia, Vol. 36, No. 1, 115-150. (2006)
14. 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)
15. Mogin, P., Luković, I., Karadžić, Z.: Relational Database Schema Design and Application Generating using IIS*CASE Tool. In: Proceedings of International Conference on Technical Informatics, Timisoara, Romania, Vol. 5, 49-58. (1994)
16. Object Management Group: MDA Guide. Version 1.0.1, Volume 1, document omg/03-06-01. (2003)

17. Object Management Group: Unified Modeling Language Specification. Version 1.4.2, document formal/05-05-01. (2005)
18. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Programmers, USA. (2007).
19. Pavićević, J., Luković, I., Mogin, P., Govedarica, M.: Information System Design and Prototyping Using Form Types. In: Proceedings of INSTICC I International Conference on Software and Data Technologies (ICSOFT), Setubal, Portugal, Vol. 2, 157-160. (2006)
20. Reppening, A., Ioannidou, A.: Agent Based End-User Development. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 43-46. (2004)
21. Rumbaugh, J., Jacobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley, USA. (1999)
22. Schmidt, D. C.: Model-driven engineering. IEEE Computer, Vol.39, No.2, 25-31. (2006)
23. Seidewitz, E.: What models mean. IEEE Software, Vol. 20, No. 5, 26-32. (2003)
24. Stanley, E., Mogin, P., Andreae, P.: S.E.A.L.-A Query Language for Entity-Association Queries. In Proceedings of the 20th Australasian Database Conference (ADC 2009), Wellington, New Zealand, Vol 92, 67-76. (2009)
25. Sutcliffe, A., Mehandjiev, N.: End-User Development. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 31-32. (2004)

Appendix

In Table 5 it is presented a list of all possible *Function Editor* node types with their descriptions.

Table 5. A list of node types available when creating a new node.

Node Type	Description
Execution Block	A new execution block as a sequence of statements, blocks and comments is created. The node is named EXECUTION_BLOCK. In its context, it is possible to create new subordinated nodes, and therefore such a node is called the complex node.
FOR structure	A new node named FOR and representing the counting FOR structure is created. Four new subordinated nodes are automatically created, denoted as: (i) Begin, (ii) Condition, (iii) Step, and (iv) FOR_BODY. The first three are text items that define: start value, end value and the step of a FOR program counter. These are the simple nodes, because they cannot have any subordinated nodes. FOR_BODY is a complex node. It represents a sequence of state-ments and blocks defining the body of a FOR structure.

WHILE-DO structure	A new node named WHILE and representing the WHILE-DO structure is created. Two new subordinated nodes are automatically created, denoted as: (i) Condition and (ii) WHILE_BODY. Condition is a text item that defines "pre-while" test condition. It is a simple node. WHILE_BODY is a complex node. It represents a sequence of statements and blocks defining the body of a WHILE-DO structure.
DO-WHILE structure	A new node named DO_WHILE and representing the DO-WHILE structure is created. Two new subordinated nodes are automatically created, denoted as: (i) DO_WHILE_BODY and (ii) Condition. Condition is a text item that defines "post-while" test condition. It is a simple node. DO_WHILE_BODY is a complex node. It represents a sequence of statements and blocks defining the body of a DO-WHILE structure.
IF-THEN-ELSE structure	A new node named IF and representing the IF selection structure is created. Three new subordinated nodes are automatically created, denoted as: (i) Condition, (ii) THEN, and (iii) ELSE, as an optional node. Condition is a text item that defines IF test condition. It is a simple node. THEN and ELSE are complex nodes. They represent sequences of statements and blocks defining the main body and the alternative body of an IF structure.
ELSE clause	A new node named ELSE in the context of an IF selection structure is created, with the same role as it would be created initially trough an IF-THEN-ELSE structure.
ELSEIF structure	A new node named ELSEIF in the context of an IF selection structure is created with a usual meaning. Three new subordinated nodes are automatically created, denoted as: (i) Condition, (ii) THEN, and (iii) ELSE, as an optional node. Condition is a text item that defines ELSEIF test condition. It is a simple node. THEN and ELSE are complex nodes. They represent sequences of statements and blocks defining the main body and the alternative body of an ELSEIF structure.

TRY-CATCH-FINALLY structure	Three complex nodes named TRY, CATCH and FINAL-LY are automatically created to specify an exception handler structure. CATCH and FINALLY nodes are the optional ones. They represent sequences of statements and blocks defining the exception handler. In the scope of CATCH, two new subordinated nodes are automatically created, denoted as: (i) Exception and (ii) CATCH_BLOCK. Exception is a simple node. It is a text item that references a previously declared exception. CATCH_BLOCK is a complex node. It represents a sequence of statements and blocks aimed to handle a raised exception. Multiple nesting of TRY nodes is allowed. In the scope of a current TRY node it is possible to create many CATCH or FINALLY nodes.
Statement	A new node representing a simple statement is created in the context of a block. It is a simple node structured as a text item. Currently, there are two types of simple statements: assignments and expressions. In the future research, we also plan to embed SQL statements.
Declaration Block	A new declaration block as a sequence of declarations and comments is created. The node is named DECLARATION. It is a complex node. In its context, it is possible to create new declarations of types, variables, constants, cursors, exceptions, and local functions.
Declaration	A new declaration is created in the context of a declaration block. A declaration is a simple node. It represents a text item that defines particular declaration of a type, variable, constant, cursor, exception or function inclusion.
LOCAL_FUNCTION declaration	A new node named LOCAL_FUNCTION is created in the scope of a declaration block. It represents a declaration of a local function. Three new subordinated nodes are automatically created, denoted as: (i) Function Name, (ii) ARGUMENTS, and (iii) LOCAL_FUNCTION_BODY. Function Name is a simple node. It is a text item that defines local function name. ARGUMENTS is a complex node. It comprises declarations of local function arguments only. LOCAL_-FUNCTION_BODY is a complex node. It represents a whole function body of a local function being declared.

Local Function Argument	A new node in the context of an ARGUMENTS node in a LOCAL_FUNCTION declaration is created. It is a simple node structured as a text item. It represents a formal argument of a local function given with the name and an association to a domain from the repository.
Comment	A new node in the context of a block is created. It is a simple node structured as a text item. It represents a single-line comment.

Ivan Luković received his M.Sc. (5 year, former Diploma) degree in Informatics from the Faculty of Military and Technical Sciences in Zagreb in 1990. He completed his Mr (2 year) degree at the University of Belgrade, Faculty of Electrical Engineering in 1993, and his Ph.D. at the University of Novi Sad, Faculty of Technical Sciences in 1996. Currently, he works as a Full Professor at the Faculty of Technical Sciences at the University of Novi Sad, where he lectures in several Computer Science and Informatics courses. His research interests are related to Database Systems and Software Engineering. He is the author or coauthor of over 70 papers, 4 books, and 30 industry projects and software solutions in the area.

Aleksandar Popović graduated from Faculty of Science at the University of Montenegro. He completed his Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, he is a Ph.D. student and teaching assistant at the University of Montenegro, Faculty of Science. He assists in teaching several Computer Science and Informatics courses. His research interests include Software Engineering, Database Systems and Domain Specific Languages.

Jovo Mostić received his M.Sc. (4 year, former Diploma) degree from the University of Montenegro, Faculty of Science in Podgorica. He completed his Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, he works as an IT project manager in Erste & Steiermärkische Bank in Podgorica. His research interests are related to Information Systems, Database Systems and Software Engineering.

Sonja Ristić is holding a position of an associate professor at the University of Novi Sad, Faculty of Technical Sciences, Serbia. She received two bachelor degrees (4 year, former Diploma) from University of Novi Sad, one in Mathematics, Faculty of Science in 1983, and the other in Economics from Faculty of Economics, in 1989. She also received her Mr (2 year) and Ph.D. degrees in Informatics, both from Faculty of Economics, in 1994 and 2003, respectively. From 1984 till 1990 she worked with the Novi Sad Cable Company "NOVKABEL" – Factory of Electronic Computers. From 1990 till 2006 she was with High School of Professional Business Studies -Novi Sad, and since 2006 she has been with the Faculty of Technical Sciences, University of Novi Sad. Her research interests are related to Database Systems and Software Engineering. She is the author or coauthor of over 40 papers in the area.

Received: November 12, 2009; Accepted: January 27, 2010.