Domain-Specific Modeling and Languages

Martin Gogolla

July 6, 2011

Domain-specific modeling (DSM) is an approach for constructing systems that fundamentally relies on employing domain-specific languages (DSLs) to represent the different system aspects in the form of models. A DSL is said to offer higher-level abstractions than a general-purpose modeling language and to be closer to the problem domain than to an implementation-platform domain. A DSL catches domain abstractions as well as domain semantics and supports modelers in order to develop models with a direct use of domain concepts. Domain rules can be incorporated into the DSL in the form of constraints, making the development of invalid or incorrect models much harder. Thus, domain-specific languages play a central role in domain-specific modeling. In order to define a domain-specific modeling language, two central aspects have to be taken into account: the domain concepts including constraining rules (which constitute the abstract syntax of the DSL), and the concrete notation employed to represent these concepts (which can be given in either textual or graphical form). In this paper we mainly focus on the abstract syntax. The abstract syntax of a domain-specific language is frequently described by a metamodel. A metamodel characterizes the concepts of the domain, the relationships between the concepts, and the restricting rules that constrain the model elements in order to reflect the rules that hold in the domain. Such an approach supports fast and efficient development of DSLs and corresponding tools (for example, translators, editors, or property analyzers).

Let us explain these ideas with an example. We consider a few elements of the well-known relational database language SQL as a domain-specific language and show in the screenshot in Fig. 1 how these features would be represented and analyzed with our tool USE. We describe the abstract syntax of the considered SQL elements with a metamodel, which embodies structural requirements in the form of a class diagram together with restricting constraints. We show how this metamodel can be validated and analyzed with usage scenarios.

• An overview of the metamodel for the tiny SQL subset is shown in the project browser in the left upper part of the screenshot and in the class diagram in the lower right part. Classes, associations and invariants are pictured in the browser. From the class diagram we learn that a relational schema (class RelSchema representing an SQL table) has attributes (columns) and that an attribute is typed through a data type. A
Figure 1: USE screenshot of Relational DB Metamodel
relational schema is populated with rows (tuples) in which each attribute gets a value by means of attribute map objects.

- Further rules are stated in the form of invariants which restrict the possible instantiations, i.e., the object diagrams of the metamodel. The names of these invariants are shown in the ‘Class invariants’ window in the middle of the screenshot. We hide the OCL details but only informally explain the constraint purpose in the order in which the invariants appear: (a) the set of key attributes of each relational schema has to be non-empty, (b) the attributes names have to be unique within the relational schema, (c) each row must have an attribute value for each of its attributes, and (d) each row must have unique key attribute values.

- In the upper part of the screenshot we see a usage scenario in concrete SQL syntax. One table (relational schema) is created, populated by two SQL insert commands and finally modified with an additional SQL update command.

- This usage scenario is represented in the abstract syntax of the metamodel in the form of an evolving object diagram. The screenshot shows only the last object diagram after the SQL update has been executed: (a) after the create command only the four left-most objects (rs1, a1, a2, dt1) are present; (b) after the first insert command the five middle objects (r1, am1, v1, am2, v2) appear, however we will have \( v1.content='Ada' \); (c) after the second insert the five right-most objects (r2, am3, v3, am4, v4) will show up; up to this point all four invariants evaluate to ‘true’; (d) after the update command the ‘content’ value of v1 changes (\( v1.content='Bob' \)) and the evaluation of the invariant keyMapUnique turns to ‘false’.

- Let us further explain the impact of the invariants by means of changing the stated object diagram: (a) the first invariant would turn to ‘false’ if we would say \( a1.isKey=false \); (b) the second invariant would turn to ‘false’ if we would say \( a2.name='firstName' \); (c) the third invariant would turn to ‘false’ if we would delete the objects am2 and v2; (d) the fourth invariant would turn to ‘true’, if we would say \( a2.isKey=true \).

- The situation is analyzed with the OCL query shown in the screenshot. The OCL query finds the objects which violate the failing constraints: All objects are returned for which another object exists with has the same key attribute values.

Our approach to defining a (domain-specific) RBAC language, which will be explained in the forthcoming parts, follows the principles used above for the tiny SQL subset: Definition of the abstract syntax of the language concepts, and characterization of their dynamic evaluation in the form of a metamodel that consists of a class diagram and restricting constraints.