4. State-Based Semantics

“He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.”

– Leonardo da Vinci (1452-1519)

In Part I, we described the first prototype for model management, called Rondo, which offers a set of high-level operators for solving metadata-related problems. Using Rondo, we developed scripts for several practically relevant scenarios, such as change propagation, view reuse, and reintegration. We found that the scripts produce intuitively correct results and that the structural operator definitions that we give are useful for solving practical problems.

In Chap. 2, we defined the semantics of the model-management operators for morphisms, a very simple mapping language. A morphism is represented as a set of arcs connecting the elements of two schemas. Although the desired result of the operators seems intuitively clear when morphisms are utilized, the treatment in Chap. 2 provides little guidance with respect to what results the operators should return if SQL views, XQuery, Datalog, or other more expressive languages are deployed in scripts instead of morphisms.

In this chapter, we present a way of defining the semantics of the operators in a truly generic fashion, without assuming any specific model and mapping languages. The main idea of our approach is to express the effect of applying the operators to models in terms of what the operators do to the instances of these models. For example, the effect of applying the operators to a database schema is expressed in terms of the valid database states described by the schema. In this way, we can characterize the semantics of operators without relying on any particular meta-model or meta-meta model. We call this kind of semantics state-based, or instance-based, semantics. In contrast, the semantics defined in Chap. 2 is driven by the structural properties of models, i.e., by the relationships between the individual models elements. To distinguish the state-based definitions from the structural definitions, in this chapter we use a distinct font face for the operators. Thus, we write Extract instead of Extract, and denote the composition using \( \circ \) instead of \( \ast \).
This chapter is structured as follows:

- In Sect. 4.1, we introduce the state-based approach and formal notation used in this chapter. We define the state-based semantics of model-management scripts and explain what it means to execute a script.
- In Sect. 4.2, we specify the state-based semantics of the operators. We derive alternative formulations of operator definitions that are substantially easier to verify for concrete schema and mapping languages. We present detailed examples of how the operators can be applied to relational schemas and SQL views.
- In Sect. 4.3, we consider in more detail the problem of computing the results of a script, which we refer to as materialization.

In Chap. 5, we revisit the change propagation scenario from the state-based perspective, and address the relationship between the structural and state-based operator definitions in Chap. 6.

Specifying the state-based semantics of the operators allows us to lay out a clear extensibility path for supporting more complex mapping languages in our prototype. Furthermore, it helps us study formally the properties of the operators and the behavior of model-management systems: the existing ones, such as Rondo, as well as systems that will be built in the future.

4.1 Basic Concepts

In this section we present the concepts of a model and a mapping and explain the notation used in the rest of the chapter. For clarity, in the examples that we give we put schema and mapping definitions in French quotation marks «...» For example, «R(A,B), S(C)» denotes a relational schema with two tables, R and S. Furthermore, we distinguish between the set semantics for relational tables, when the table is not allowed to contain duplicate tuples, and the multiset semantics used in SQL. For the former we write «R[A,B]», for the latter we use square brackets: «R[A,B]». We abbreviate SELECT DISTINCT clauses in view definitions as SELECTD.

4.1.1 Models

A model is a formal description of an application artifact, such as a relational schema, a workflow definition, or an interface specification. Typically, a model serves as a template for instances. For example, an instance of a relational schema is a valid database state; an instance of a workflow is a valid transition graph; an instance of a programming interface is an implementation that conforms to the interface. Let Inst(m) denote the set of all possible instances of m. If m is a database schema, every instance db ∈ Inst(m) must satisfy the constraints present in m.