A Survey of Languages for Specifying Dynamics: A Knowledge Engineering Perspective

Pascal van Eck, Joeri Engelfriet, Dieter Fensel,
Frank van Harmelen, Yde Venema, and Mark Willems

Abstract—During the last years, a number of formal specification languages for knowledge-based systems has been developed. Characteristics for knowledge-based systems are a complex knowledge base and an inference engine which uses this knowledge to solve a given problem. Specification languages for knowledge-based systems have to cover both aspects. They have to provide the means to specify a complex and large amount of knowledge and they have to provide the means to specify the dynamic reasoning behavior of a knowledge-based system. This paper focuses on the second aspect. For this purpose, we survey existing approaches for specifying dynamic behavior in related areas of research. In fact, we have taken approaches for the specification of information systems (Language for Conceptual Modeling and TROLL), approaches for the specification of database updates and logic programming (Transaction Logic and Dynamic Database Logic) and the generic specification framework of Abstract State Machines.

Index Terms—Specification languages, knowledge-based systems, dynamics, inference control, update logics.

1 INTRODUCTION

OVER the last years, a number of formal specification languages have been developed for describing *knowledge-based systems* (KBSs). Examples are DESIRE [1], [2]; KARL [3], [4]; $K_{BS}SF$ [5], [6]; $(ML)^2$ [7]; MLPM [8], and TFL [9]. In these specification languages, one can describe both knowledge about the domain and knowledge about how to use this domain knowledge in order to solve the task which is assigned to the system. On the one hand, these languages enable a specification which abstracts from implementation details: They are not programming languages. On the other hand, they enable a detailed and precise specification of a KBS at a level of precision which is beyond the scope of specifications in natural languages. Surveys on these languages can be found in [10], [11], [12].[1]

A characteristic property of these specification languages results from the fact that they do not aim at a purely functional specification. In general, most problems tackled with KBSs are inherently complex and intractable (see e.g., [13] and [14]). A specification has to describe not just a realization of the functionality, but one which takes into account the constraints of the reasoning process and the complexity of the task. The constraints have to do with the fact that one does not want to achieve the functionality *in theory* but rather *in practice*. In fact, a large part of expert knowledge is concerned exactly with efficient reasoning given these constraints: it is knowledge about *how* to achieve the desired functionality. Therefore, specification languages for KBSs also have to specify control over the use of the knowledge during the reasoning process. A language must therefore combine *nonfunctional* and *functional* specification techniques; on the one hand, it must be possible to express algorithmic control over the execution of substeps. On the other hand, it must be possible to characterize substeps only functionally without making commitments to their algorithmic realization.

The languages mentioned are an important step in the direction of providing means for specifying the reasoning of KBSs. Still, there is a number of open questions in this area. The most important problem is the specification of the dynamic behavior of a reasoning system. The specification of knowledge about the domain seems to be well understood. Most approaches use some variant of first-order logic to describe this knowledge. Proof systems exist which can be used for verification and validation. The central question is how to formulate knowledge about *how* to use this knowledge in order to solve a task (the *dynamics* of the system). It is well-agreed that this knowledge should be described in a declarative fashion (i.e., not by writing a separate program in a conventional programming language for every different task). At the moment, the aforementioned languages use a number of formalisms to describe the dynamics of a KBS: DESIRE uses a metalogic

1. See also, ftp://swi.psy.uva.nl/pub/keml/keml.html on the World Wide Web.

- *P. van Eck is with the University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands. E-mail: vaneck@cs.utwente.nl.*
- *J. Engelfriet, D. Fensel, and F. van Harmelen are with Vrije Universiteit Amsterdam, Faculty of Sciences, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. E-mail: Joeri_Engelfriet@mckinsey.com, {dieter, frankh}@cs.vu.nl.*
- *Y. Venema is with the Institute for Logic, Language, and Computation, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands. E-mail: yde@wins.uva.nl.*
- *M. Willems is with Quintiq B.V., Het Wilsum 10, 5231 BW 's Hertogenbosch, The Netherlands. E-mail: mark@quintiq.nl.*

Manuscript received 16 June 1999; revised 24 Aug. 1999; accepted 22 Nov. 1999.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 106923.

to specify control of inferences of the object logic, $(ML)^2$ and MLPM apply dynamic logic ([15], [16]), KARL integrates ideas of logic programming with dynamic logic, and TFL uses process algebra in the style of [17]. With the exception of TFL, the semantics of these languages are based on states and transitions between these states. $(ML)^2$, MLPM, and KARL use dynamic logic Kripke style models, and DESIRE uses temporal logic to represent a reasoning process as a linear sequence of states. On the whole, however, these semantics are not worked out in precise detail for most approaches and it is unclear whether these formalisms provide apt description methods for the dynamics of KBSs. Another shortcoming of most approaches is that they do not provide an explicit proof system for supporting (semi) automatic proofs for verification.

These shortcomings motivate our effort to investigate specification formalisms from related research areas to see whether they can provide insight in the specification of (in particular, the dynamic part of) KBSs. We have analyzed related work in information system development, databases, and software engineering. Approaches have been selected that enable the user to specify control and dynamics. The approaches we have chosen are:

- Language for Conceptual Modeling (LCM, [18]) and TROLL ([19]) as examples from the information systems area. Both languages provide the means to express the dynamics of complex systems.
- Transaction Logic ([20]), (Propositional) Dynamic Database Logic (PDDL, [21] and DDL [22]) as examples for logic programming and database update languages which provide the means to express dynamic changes of databases.
- Abstract State Machines ([23]) from the theoretical computer science and software engineering areas. It offers a framework in which changes between (complex) states can be specified.

The informed reader probably misses some well-established specification approaches from software engineering: algebraic specification techniques (see e.g., [24], [25], [26], [27]), which provide the means for a functional specification of a system and model-based approaches like Z [28], [29] and the Vienna Development Method-Standard Language (VDM-SL) [30], [31], which describe a system in terms of states and operations working on these states. Two main reasons guided our selection process. First, we have looked for novel approaches on specifying the dynamic reasoning process of a system. Traditional algebraic techniques are means for a functional specification of a software system that abstracts from the way the functionality is achieved.[2] However, we are precisely concerned with how a KBS performs its inference process. Although approaches like VDM and Z incorporate the notion of a state in their specification approaches, their main goal is a specification of the functionality and their means to specify control over state transitions is rather limited. In Z, only sequence can be expressed and, in VDM, procedural control over state transitions is a language element introduced during the design phase of a system. A second and more practical reason is the circumstance that a comparison with abstract data types, VDM, Z, and languages for KBSs is already provided in [12]. Finally, one may miss specification approaches like LOTOS [32] that are designed for the specification of interactive, distributed, and concurrent systems with real-time aspects. As most development methods and specification languages for KBSs (a prominent exception is DESIRE) assume one monolithic sequential reasoner, such an approach is outside the scope of the current specification concerns for KBSs. However, future work on distributed problem solving for KBSs may raise the necessity for such a comparison. Notwithstanding, the motivation for our choices given in this paragraph, we are aware that our choice still is somewhat arbitrary.

A complicating factor for our analysis is the fact that some of the selected approaches are meant to be logics (Transaction Logic, Database Update Logic), whereas others are meant to be specification languages (LCM, TROLL, and Abstract State Machines). The specification languages all have formal semantics and the logics allow for the specification of behavior, making the comparison a valid enterprise. However, when comparing the syntax, for instance, this difference has to be kept in mind: Specification languages usually have an extensive syntax (for the benefit of ease for the programmer), whereas logics often have a small (fixed) syntactic vocabulary (making proofs about the logic shorter).

The paper is organized as follows: First, in Section 2, we introduce two dimensions we distinguish to structure our analysis. In order to give the reader an impression of what specifications of KBSs may look like in the different approaches and in order to illustrate some criticisms of the approaches, we will describe an example of a reasoning task in Section 3. This example will be used as a running example in the separate treatment of the approaches in Section 4. We will use a simplified variant of a system for automated design, known as the *propose & revise method* for parametric design ([33]). This method is well-known in knowledge engineering. It was used in the Sisyphus-II or VT-task ([34]), where several modeling approaches in knowledge engineering were compared by applying them to a common case (i.e., developing a KBS for the configuration of a vertical transportation system). In Section 4, we introduce the different approaches that we have studied; the main part of this section consists of a detailed description of the five frameworks mentioned, along with a specification of the example. Section 5 consists of a comparison between the above formalisms according to the two dimensions of our analysis introduced in Section 2.

2 The Two Dimensions of Our Analysis

In the analysis of the different frameworks, it will be convenient to distinguish two dimensions (see Fig. 1). On the horizontal axis, we list a number of concepts which should be represented in a framework. On the vertical axis, we list a number of aspects to be looked at for each of the concepts. We will explain these dimensions in some more detail.

2. Process Algebra [17] is an exception. In fact, the semantics of LCM (see Section 4.5) is based on Process Algebra. In the area of KBSs, Process Algebra is used in TFL [9].

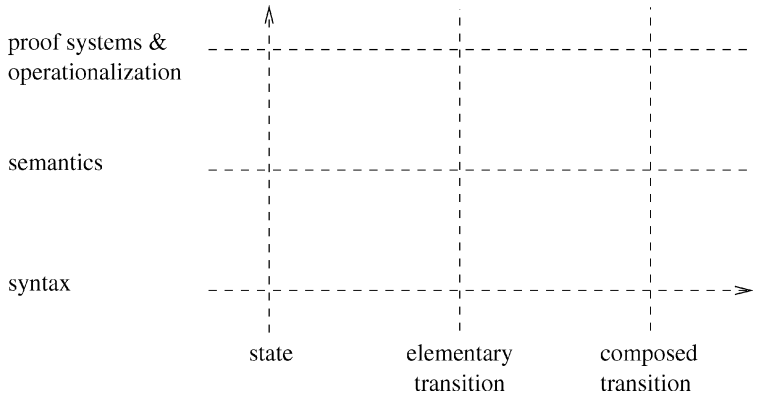

Fig. 1. The two dimensions of our analysis.

The behavior of a KBS can, from an abstract point of view, be seen as follows: It starts in some initial *state* and, by repeatedly applying some inferences, it goes through a sequence of states and may finally arrive at a terminal state. So, the first element in a specification of a KBS concerns these states. What are states and how are they described in the various approaches? Second, we look at the *elementary transitions* that take a KBS from one state to the next. Third, it should be possible to express control over a sequence of such elementary transitions by composing them to form *composed transitions*. This defines the dynamic behavior of a KBS.

The second dimension of our analysis concerns three aspects of each of the concepts described above. First of all, we look at the language of each of the formalisms (the syntax). Second, we examine the semantics of the language. In the third place, we look at proof systems and operationalization. In Section 2.1 and Section 2.2, the concepts and aspects introduced here are illustrated in more detail.

2.1 The Three Concepts Involved in the Reasoning of KBSs

Before explaining the three concepts in more detail, we will first introduce a distinction between two types of specification for systems: external and internal. The former specifies a system as a black box in terms of its externally visible behavior. It defines *what* should be provided by the system. The latter specifies a system in terms of its internal structure and the interaction between parts of its internal structure: It describes *how* the system reaches its goals. Both types of specification appear in specification languages for KBSs: External descriptions may appear at the lowest and at the highest level of specification of a KBS, while internal specifications provide relations between the descriptions at the lowest and highest levels. Current KBS specification languages do not provide this at all levels of specification. Actually, the equivalence of the external specification of the goals (the highest level) and the internal specification of the reasoning process of the KBS at lower levels is a proof obligation for the verification of the KBS.

Internal specification techniques are necessary to express the dynamic reasoning process of a KBS. A complex reasoning task may be decomposed into less complex inferences and control is defined that guides the interaction of the elementary inferences in achieving the specified reasoning goals. This also allows successive refinement. A complex task can be hierarchically decomposed into (easier) subtasks. These subtasks are specified externally and treated as elementary inferences. If a subtask defines a computationally hard problem, it can again be decomposed into a number of subtasks, along with an internal specification of how and when to invoke these subtasks.

In the rest of this section, we discuss the three concepts of a specification in more detail. In Section 4, the five specification approaches chosen are discussed in terms of these concepts. In fact, we discuss the representation of states, elementary transitions, internal specifications of complex transitions (i.e., hierarchical refinement and control), and external specifications of complex transitions together with the relationship to their internal description.

2.1.1 States

With regard to the representation of the states of the reasoning process, one can distinguish

1. whether it is possible to specify a state at all,
2. whether a state can be structured (i.e., decomposed into a number of substates)[3], and
3. how an individual state is represented.

Not each specification approach in software or knowledge engineering provides the explicit notion of a state. An alternative point of view would be an *event-based* philosophy useful to specify parallel processes (compare [35]). TFL uses processes as elementary modeling primitives that are further characterized by abstract data types in the style of process algebra [17]. No explicit representation of the reasoning state is provided. The other approaches from knowledge engineering agree on providing the notion of a state but differ significantly in the way they model it. $(ML)^2$, MLCM, and KARL represent a global state. Still, it may be decomposed in what is called *knowledge roles* or *stores*. DESIRE provides decomposition of a global state of the reasoner into local states of different reasoning modules (subcomponents of the entire system).

3. In a distributed system, a state of the system can be composed of states of the system's distributed parts. These states are usually called *local states*.

Semantically, the main descriptions of a state are: as a propositional valuation (truth assignments to basic propositions, as used in the propositional variants of dynamic logic and temporal logic ([36])), as an assignment to program variables (as in the first-order variant of Dynamic Logic), as an algebra (we will see that in Abstract State Machines), or as a full-fledged first-order structure (as in the first-order variants of temporal logic).

2.1.2 Elementary Transitions

Elementary transitions should be describable without enforcing any commitments to their algorithmic realization. A purely external definition is required, as a specification should abstract from implementational aspects. Still, "elementary" does not imply "simple." An elementary transition can describe a complex inference step, but it is a modeling decision that its internal details should not represent. However, later on it can be decided that at a lower level of specification, an internal specification should be given. Ideally, a modeling framework should provide support for such refinements. After refinement, the elementary transition is considered to be a composed transition.

There is a distinction between different modeling frameworks with respect to the elementary transitions provided. Some modeling frameworks provide a fixed set of predefined elementary transitions, such as a set consisting of a few general database update operators. The semantics of these modeling frameworks give an external specification of these elementary transitions. Other modeling frameworks enable the user to define elementary transitions, instead of providing a fixed set. In this case, the user gives their names and external specifications of their functionality.

2.1.3 Composed Transitions

One can distinguish nonconstructive and constructive manners to specify control over state transitions. A *nonconstructive* or *constraining* specification of control defines *constraints* obeyed by legal control flows. That is, they exclude undesired control flows but do not directly define actual ones. Examples for such a specification can be found in the domain of information system specifications, e.g., $\mathcal{TR}$ and TROLL. *Constructive specifications* of control flow define directly the actual control flow of a system and each control flow which is not defined is not possible. In general, there is no clear cutting line between both approaches, as constructive definitions of control could allow nondeterminism which again leads to several possibilities for the actual control. As an example, consider Dynamic Logic. The control in Dynamic Logic is specified using programs (with assignment, test, iteration, etc.). Such an imperative language directly specifies the control flow; thus, specification of control in Dynamic Logic is constructive. (Indeterminism may be introduced if a choice operator is present.) This is in contrast to specifications in temporal logic. A formula that expresses, for example, that a parameter may never have a certain value in the future (or fairness and liveliness constraints), only constrains the possible models, but does not allow us to directly construct the control flow. Such specifications in temporal logic are therefore nonconstructive.

Another distinction that can be made is between *sequence-based* and *step-based* control. In sequence-based control, the control is defined over entire sequences of states. That is, a constraint or constructive definition may refer to states anywhere in a sequence. In a step-based control definition, only the begin state and the end state of a composed transition are described. For example, in Dynamic Logic, a program is represented by a binary relation between initial and terminal states. There is no *explicit* representation of intermediate states of the program execution. Other approaches represent the execution of a program by a sequence of states (for example, approaches based on temporal logic). It begins with the initial state and, after a sequence of intermediate states, the final state is reached if there is a final state (a program may also run forever, as in process monitoring systems).

For the representation of the reasoning process of KBSs, this distinction has two important consequences: 1) in a state-pair oriented representation, a control decision can only be made on the basis of the actual state. A state-sequence oriented representation provides the history of the reasoning process. Not only the current state but also the reasoning process that leads to this state is represented. Therefore, strategic reasoning on the basis of this history information becomes possible. For example, a problem-solving process that leads to a dead-end can reflect on the reasoning sequence that led to it and can modify earlier control decisions (by backtracking) and 2) with a representation as a sequence of states it becomes possible to define dynamic constraints that do not only restrict valid initial and final states but that restrict also the valid intermediate states. Such constraints are often used in specifications of information systems or database systems.

2.2 The Three Aspects of a Specification of the Reasoning of KBSs

Perpendicular to the three specification concepts are the three aspects syntax, semantics, and proof systems/operationalization. For each of the concepts, these three aspects together determine how and to which extent a concept can be used in a specification: They constitute the practical materialization of the concepts state and (elementary and composed) transition.

2.2.1 Syntax

Each of the three concepts of a specification is represented by a part of the syntax of a specification framework. A spectrum of flavors of syntax can be distinguished. At one end of this spectrum, specification languages with an extensive syntax can be found, resembling (conventional) programming language syntax. Usually, such a language is specified by EBNF grammar rules, and operators and other syntactic elements are represented by keywords easily handled by software tools that support the specification process. At the other end of the spectrum, languages can be given by defining a notion of well-formed formulae composed of logical operators and extra-logical symbols, using a few grammar rules.

TABLE 1
Evaluation Criteria

	State	Elementary transition	Composed transition
Syntax	The type of formulae used to describe states, e.g. propositional, first-order, or equational formula	Fixed: summary of vocabulary provided User-defined: characterisation of syntax for defining elementary transitions	Constructive: summary of vocabulary provided Constraining: the type of expressions used to constrain possible sequences
Semantics	The type of of structure denoted, e.g. valuation, algebra, first-order structure	The type of relation between states denoted by elementary transitions, e.g. number of substates that may differ	The type of structure over which formulae are interpreted, e.g. begin/end state pairs (step based) or sequences (sequence-based)
Proof system & operationalization	Availability and type of proof system, e.g. Hilbert-style, Gentzen-style, for proving properties of states	Availability of proof system for properties of transitions, possibility to carry out transitions in operationalization	Availability of proof system and operational semantics or interpreter, restrictions imposed on full language for applicability

2.2.2 Semantics

Semantics of specification elements can be viewed as a function that interprets well-formed formulae or syntactic expressions in some semantical domain, usually a mathematical structure. A semantic serves two purposes: It enables the definition of a precise meaning of language expressions and it enables proofs of statements over language expressions. To support these purposes, such a semantics should be formal. Proofs can be formalized and semiautomatic proof support can be provided if a proof system based on a formal semantics has been developed. The semantics should be intuitive and relatively easy to understand so users are able to precisely comprehend what a specification means.

2.2.3 Proof Systems and Operationalization

One of the main reasons for developing formal specifications of a system is to be able to rigidly prove properties of the system specified. To support such proofs, specification frameworks should include a formal proof system, which precisely specifies which properties can be derived from a given specification. At the very least, such a proof system should be sound: It must be impossible to derive statements about properties of a specification that are false. Second, a proof system should ideally be complete, which means that it is powerful enough to derive all properties that are true.

Formal specification frameworks can enable the automatic development of prototypes of the system being specified. Such prototypes can then be evaluated to assess soundness and completeness of the specification with respect to the intended functionality of the system being specified, thus providing restricted but still very useful support for the validation of specifications. The "operationalization" of a specification framework is meant to refer to the possibilities and techniques for such automatic prototype generation.

2.2.4 Overview

The two dimensions of our analysis are summarized in Table 1. In Section 4, for each of the five approaches chosen, the three aspects of the concepts state, elementary transitions, and composed transitions are discussed with a focus according to Table 1. Section 5 summarizes the five approaches chosen in a similar way.

3 The Running Example

The aim of the example we describe below is to illustrate the different formalisms we are studying, thus facilitating comparison with respect to specification of the dynamics of knowledge-based systems. In particular, we use the example to examine the representation of *states* of the reasoning process; the representation of *elementary transitions* between states and the representation of *control* over the execution of transitions. On the one hand, the example should be nontrivial to allow a good illustration of the above points. On the other hand, it should not be too complex; it is meant just as an illustration. We will use a simplified variant of the *Propose & Revise* problem solving method for the task `parametric design` as our running example. A full description of this task would take up too much space, so we will give a rather informal description. Also, we will not fully specify this example for all formalisms; rather we will focus on the interesting parts.

During the discussion of the example, we use *stores* to represent the state of the reasoning process. A store can be thought of as a placeholder for information (or knowledge). We will use *inference actions* to represent the elementary transitions between states. An inference action is an action that takes information from a store, reasons with it, and outputs the result to another store. We use *tasks* to represent composed transitions. A task is meant to perform some functionality, usually more complex than what can be performed by a single inference action. We will use a procedural language for defining control over the execution of transitions within tasks. Our style is influenced by the KADS-I [37] and CommonKADS [38] projects but it can be easily translated into the terminology of most of the other approaches in knowledge engineering (see [39]).

Our example consists of solving a design problem (of artifacts, but also for example of schedules). The design problem is viewed as a parametric design problem, i.e., the design artifact is described by a set of parameters. A design is an assignment of values to parameters. If some parameters do not have a value yet, the design is called *partial*. Otherwise, it is called *complete*. The design process must determine a value for each of them fulfilling certain requirements (described as constraints on the joint

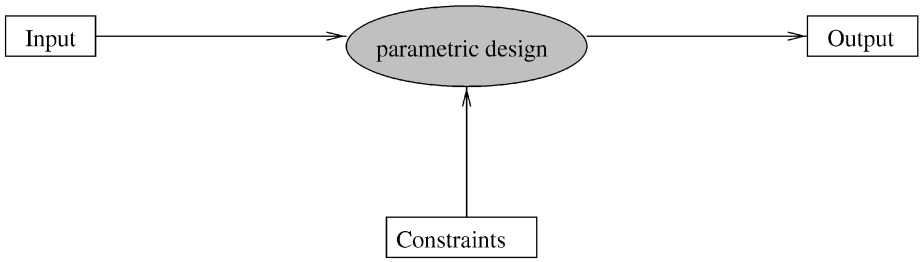

Fig. 2. Input and output of the task `parametric design`.

assignment of values to the parameters). Examples of this type of task are: Sisyphus-I (see [40]), where employees are assigned to places, ECAI '92 workshop example (see [10]): a simple scheduling task where activities have to be assigned to time slots, and the Sisyphus-II or VT-task [34], where an elevator is configured by choosing components and assigning values.

The central task is to find values for parameters, fulfilling constraints. The user is allowed to give some parameters already a value from the start. This task will be called `parametric design` and is depicted in Fig. 2.

In Fig. 2, the task `parametric design` is depicted as an oval. It is shaded to show that it is a composed task that should be further decomposed. The task `parametric design` gets its input from two stores (represented by rectangles in Fig. 2): `Input` and `Constraints`. The store `Input` contains values for parameters given by the user beforehand (it may be empty). The store `Constraints` contains the "hard" requirements on the (partial) design. The task `parametric design` must find a complete design (an assignment of values to parameters) that extends the partial design given by the user in `Input` and fulfills the requirements of `Constraints`. If it finds such a design, it will pass this to the store `Output`.

Formally, we assume that we have a number of parameters, $p_1, \ldots, p_n$, and, for each parameter p_i, a set of possible values D_i for this parameter. A partial design is a function that assigns a value from D_i to p_i, for some parameters p_i. A complete design assigns a value to each parameter. We will provide an informal functional specification of the task `parametric design` by listing the requirements on a design output by this task. We are aware that there are much richer and more complex ways for defining the design task and design problem solving. However, the purpose of our description of `parametric design` and *Propose & Revise* is to illustrate the different formalization approaches and not to provide a rich and detailed picture of design problem solving.

PR1. The initial partial design given by the user (in `Input`) may not be modified. That is, the final assignment must be an extension of the initial assignment.

PR2. The design must be complete: Each parameter must have a value.

PR3. The design must be correct, i.e., no constraints may be violated.

This functional specification of the task `parametric design` does not provide any information on how to implement this task. Moreover, `parametric design` is, in principle, an intractable task, so we will generally want to further refine such tasks in the sense that additional, possibly heuristic knowledge is applied to arrive at an acceptable and efficient approximation of the original task [41]. Therefore, a problem solving method which provides information on how to implement an efficient approximation has to be chosen. The problem solving method chosen in this paper is *Propose & Revise*. The central idea behind *Propose & Revise* is that repeatedly, values are proposed for parameters, treating each parameter in succession. After a value has been proposed, the partial design is tested to see whether any constraints are already violated. If not, then a value for the next parameter is proposed and again tested. If a constraint is violated, we try to revise the current (partial) design by changing some values for parameters that were already assigned a value, in such a way that no constraint is violated. After this, we again propose a value for a parameter, until the design is complete. The task `parametric design` can be decomposed into six subtasks (according to the *Propose & Revise* method); see Fig. 3.

Below is a list of the six subtasks of *Propose & Revise*:

- `Init`: This task initializes the design.
- `Propose`: This task proposes a value for a parameter that has not been assigned a value before.
- `Test`: This task checks whether the current design violates any constraint.
- `Revise`: This task corrects the partial design if the previous task found any violated constraints.
- `Evaluate`: This task checks if the design is complete.
- `Copy`: This task copies the design to the `Output` store.

All of these subtasks, except for `Revise`, are elementary inference actions. The subtask `Revise` is a more complex task, which should be further decomposed. We will not give the decomposition here, but instead we will just describe its functionality.

We have already described the stores `Input`, `Constraints`, and `Output`. The store `Design` contains the current design which is a (partial) assignment of values to parameters. The store `Violations` holds the constraints that are violated by the current design. This store is updated by the `Test` inference action. The inference action `Evaluate` checks whether the current design is complete. If this is the case, it returns "true;" otherwise, it returns "false." As mentioned before, the subtask `Revise`, which takes the current design as input, alters it and outputs the altered design, is more

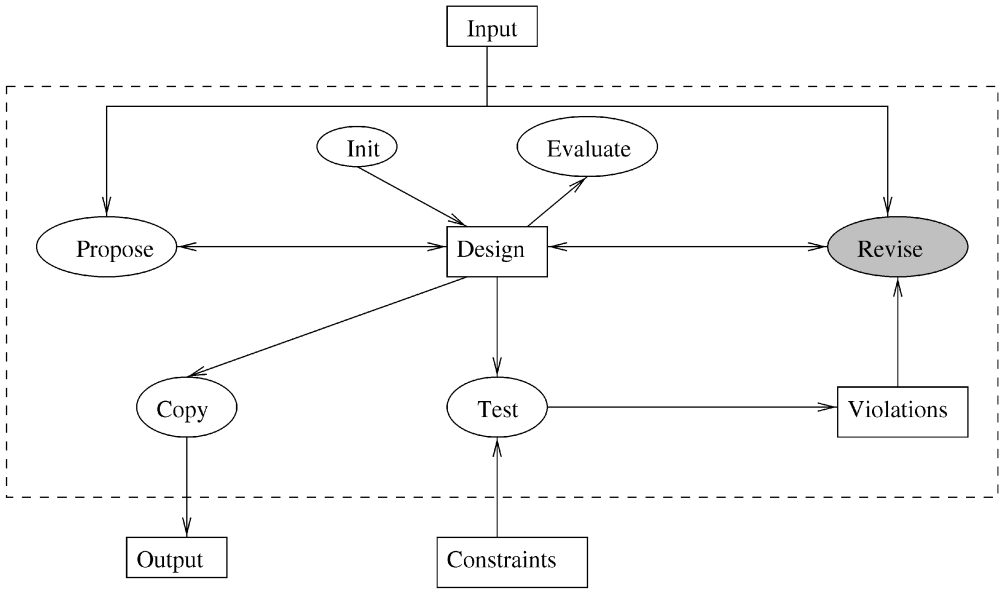

Fig. 3. Knowledge flow diagram of *Propose & Revise*.

complex and is therefore not described by an elementary inference action. Due to space restrictions, we will not give the full decomposition of this task in terms of simpler subtasks, but simply give a number of requirements on this task:

R1. The altered design is correct, i.e., no constraint is violated by it.

R2. No new parameter was assigned a value (i.e., `Revise` only alters the values of parameters that have a value).

R3. The new design respects the initial design given by the user. This means that only parameters that were not given a value by the user may be given a different value.

R4. All parameters that already have a value must still have a value in the new design.

Finally, we need to define the control between subtasks of `parametric design`. One possibility is provided in Fig. 4. After the initialization, a loop of `Propose`, `Test`, (and if necessary) `Revise` is entered until a complete, correct design has been found. In this case, it is copied to `Output`.

Parametric design based on (variants of) the *Propose & Revise* method has been studied extensively. The interested reader is directed to [42] and [43] for complete formal models of parametric design using DESIRE and KARL, respectively.

```
Init;
repeat
      Propose;
      Test;
      if (Violations ≠ ∅) then Revise endif
until Evaluate;
Copy
```

Fig. 4. Control flow of *Propose & Revise*.

4 Frameworks Studied

In this section, we will give a description of all of the frameworks to be studied. Each description will treat syntax, semantics, axiomatization, and proof calculi (if existing). This will be done for states, elementary transitions, internal specification of composed transitions, and external specification of composed transitions. The running example will be (partly) specified in all approaches. In particular, for each approach, a formalization of the functional specification (if possible) and of the problem solving method will be given. In Section 5, we will compare the different approaches using the two dimensions introduced in Section 2.

4.1 Transaction Logic

Transaction logic (or $\mathcal{TR}$, see [44], [45]) is designed especially to deal with the dynamics of database updates and logic programming. It is an extension of predicate logic meant to deal with dynamics in a clean and declarative fashion. The idea is to combine predicates that express properties of states with predicates that express properties of transitions, into a single amalgamated logic. The main elements of $\mathcal{TR}$ are transitions: queries, updates, and combinations of these. Queries do not change the database, updates do. Using a sequence operator ($\otimes$), one can combine primitive state transitions (as specified in so-called oracles) into more complex transitions.

$\mathcal{TR}$ does not make a syntactic distinction between (nonupdating) queries and actions with side effects. This is considered a feature, because it allows a uniform treatment of queries and actions. However, as indicated in Section 1, from a knowledge engineering perspective, this blurring of the dynamics/statics distinction is not desirable. It is possible, of course, to give descriptive names to actions: The action which inserts a fact "b" into the database could be called *ins_b*. Transaction logic offers a clean treatment of the assert and retract operators in Prolog. Indeed, a semantics is given for the interaction between logical

operators and updates (e.g., $\neg assert(X)$) that is lacking in other work.

4.1.1 Syntax

It is important to appreciate the distinction between *transitions* and *transactions* in $\mathcal{TR}$. Transitions are the elementary steps that update one state into another, whereas transactions are complex compositions of these elementary transitions. One should think of transaction formulae as formulae that are true for sequences of states. Syntactically, however, there is no distinction between state formulae, transition formulae, and transaction formulae.

Transaction formulae are used to combine (simpler) transactions into more complex ones. All classical logical connectives can be used to define complex transactions, by stating the logical relationships that must hold. One special connective, $\otimes$, expressing sequence is added to specify dynamics. So, the dynamic meaning of the formula $\phi \otimes \psi$ is that first ϕ and then ψ holds.

Other dynamic operators are defined in terms of this operator. An example of such an operator, which will be used in the running example, is $\Rightarrow$. The formula $\phi \Rightarrow \psi$ means that whenever ϕ is true, ψ must be true immediately thereafter. Formally, $\phi \Rightarrow \psi \equiv \neg(\phi \otimes \neg\psi)$.

An example is a transaction formula that defines a transaction *design* as an *initialize* transaction followed by a *propose&revise* transaction (both possibly defined somewhere else).

$$design \leftarrow initialize \otimes propose\&revise.$$

The strange thing is that this formula does not show syntactically that it defines a dynamic transaction, even though $\otimes$ is used. If both *initialize* and *propose&revise* are ordinary database predicates (which may be true or not in a database state), then *design* is just a derived predicate which may be true or not in the database state. If both operands of a $\otimes$-operators are static, then $\otimes$ behaves as classic conjunction. Contrary to what the name suggests, transaction formulae can also express static properties, like "a constraint is violated if the values assigned to the n parameters are illegal:"

$$violation(C) \leftarrow constraint(C, V_1, \ldots, V_n) \\ \wedge design(P_1, V_1) \wedge \ldots \wedge design(P_n, V_n).$$

In a specification language, one may want to show syntactically which formulae describes states and which correspond to proper transactions. We will use the special predicate **state** which is defined to be true on all states (or, to be precise, on sequences of states of length 1), and false on all sequences of states of length greater than 1. For a formula ϕ, we will abbreviate $\phi \wedge \mathbf{state}$ by $\langle\phi\rangle$. This formula is only true on states where ϕ is true.

It is insightful to consider an example of a while-loop specified in transaction logic.

$$while \leftarrow (\langle test\rangle \otimes do \otimes while) \vee \langle\neg test\rangle.$$

This rule states that the transaction *while* is true if either *test* is false in the first state, or else if *test* is true, after which the sequence *do* and *while* is true. The procedural reading of this rule is: In order to perform *while*, either *test* must be false, or if it is true, then *do* must be performed, after which *while* must be performed again.

Similar to Prolog for logic, there is an executable Horn version for $\mathcal{TR}$ in which all transaction formulae have to be implications with a single atom as head and a sequence of atoms as body. A generalized Horn version, including stratified negation and negation-as-failure is also defined.

A nice feature of $\mathcal{TR}$ is that one can express formulae that constrain the possible sequences. The formula below, for instance, makes sure that *revise* is never performed twice in a row.

$$\neg(revise \otimes revise).$$

So, transaction logic combines the constraining and constructive specification styles.

The style of control in $\mathcal{TR}$ can be characterized as sequence-based because we can express properties that must hold over sequences of states of any length. An important means of control (to which the Horn version is restricted), is the definition of procedures. Iteration (like the while-loop above) can be obtained by recursion.

4.1.2 Semantics

Transaction logic describes the dynamics of database updates, which is reflected in the semantics. A set of states exists representing all possible database states, and transaction formulae are interpreted over sequences of such states. This is in fact the power of $\mathcal{TR}$: By assigning an interpretation to a sequence of states, it is possible to specify conditions on intermediate states and to use logic to specify control.

Transaction logic is parameterized by a pair of oracles $\mathcal{O}^d$ and $\mathcal{O}^t$. The domain of these oracles is a set of state identifiers. The oracles map a state identifier, respectively, a pair of state identifiers to a set of first-order formulae. Then, $\phi \in \mathcal{O}^d(\mathbf{D})$ means that ϕ is true on the database state identified by $\mathbf{D}$. Similarly, $\phi \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ means that ϕ is true on the transition from $\mathbf{D}_1$ to $\mathbf{D}_2$. The intuitive meaning of this is that ϕ is the name of a transition that can take $\mathbf{D}_1$ to $\mathbf{D}_2$.

The domains of these oracles define the set of *paths*, i.e., finite sequences of state identifiers: $\langle \mathbf{D}_1, \ldots, \mathbf{D}_k\rangle$, where $k \geq 1$. Each formula will be interpreted with respect to such paths. Formally, a model or *path structure* is a triple $\mathbf{M} = \langle U, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{path}\rangle$. U is a set called the domain of $\mathbf{M}$ and $\mathcal{I}_{\mathcal{F}}$ is an interpretation function of the function symbols over U. Together these define the class of all first-order structures denoted by $Struct(U, I_{\mathcal{F}})$ (so, this class contains all structures of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}}\rangle$, where $I_{\mathcal{P}}$ is an interpretation of predicate symbols on U). Finally, $\mathcal{I}_{path}$ is a mapping that assigns to every path a structure in $Struct(U, \mathcal{I}_{\mathcal{F}}) \cup \{\perp\}$, where $\perp$ is a special model in which all formulae are true.[4] This mapping must obey two conditions: *compliance with the data oracle*, so if $\phi \in \mathcal{O}^d(\mathbf{D})$, then $\mathcal{I}_{path}(\langle\mathbf{D}\rangle) \models_\nu \phi$ (where $\models_\nu$ denotes the classical satisfaction of a formula in a first-order structure, with respect to the variable assignment ν), and

4. The reason for including $\perp$ here is to localize inconsistency to the states wherein it occurs.

compliance with the transition oracle so if $\phi \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, then $\mathcal{I}_{path}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle) \models_\nu \phi$. The $\mathbf{D}_i$ are just state identifiers. The interpretation function assigns a meaning (semantics), which consists of first-order structures with domain U, to these identifiers (and, in fact, to every sequence of identifiers). The oracles constrain the possible interpretation by stating which formulae must at least be true in the meaning assigned to these identifiers (and pairs of identifiers).

Transaction formulae are interpreted with respect to a sequence of states and a variable assignment. An atomic statement $P(t_1, \ldots, t_n)$ is true in a model $\mathbf{M}$ with respect to path π and variable assignment ν iff $\mathcal{I}_{path}(\pi) \models_\nu P(t_1, \ldots, t_n)$. The interpretation of the standard connectives is defined as usual. The only special case is for the $\otimes$-operator. Intuitively, $\phi \otimes \psi$ is true on a path if first ϕ and then ψ is true. Formally, $\mathcal{I}_{path}(\langle \mathbf{D}_0, \ldots, \mathbf{D}_n \rangle) \models_\nu \phi \otimes \psi$ iff it is possible to split the path $\langle \mathbf{D}_0, \ldots, \mathbf{D}_n \rangle$ into two paths, $\langle \mathbf{D}_0, \ldots, \mathbf{D}_i \rangle$ and $\langle \mathbf{D}_i, \ldots, \mathbf{D}_n \rangle$ such that ϕ is true in $\mathbf{M}$ with respect to the first path, and ψ is true in $\mathbf{M}$ with respect to the second. A transaction formula is true in a model if it is true in the model with respect to every possible path. A set of transaction formulae (a program, $\mathbf{P}$) is true in a model if all formulae are true in the model.

The idea behind entailment in $\mathcal{TR}$ is that we can evaluate a transaction ϕ with respect to a sequence of databases on the basis of the specification of dynamics (transitions and transactions). Given a program $\mathcal{P}$ we say that ϕ is executionally entailed,

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \phi \tag{1}$$

if and only if, for every model $\mathbf{M}$ of $\mathbf{P}$, we have that $\mathbf{M}, \langle \mathbf{D}_0, \ldots, \mathbf{D}_n \rangle \models \phi$. The statement

$$\mathbf{P}, \mathbf{D}_0 - - - \models \phi$$

expresses the fact that there exists a sequence $\mathbf{D}_0, \ldots, \mathbf{D}_n$ (for some n) such that (1) holds. This statement is especially useful, because it reflects the situation where the user asks to execute a transaction ϕ from a database $\mathbf{D}_0$ and the result is a sequence of states, or in other words, an execution of the program $\mathbf{P}$.

Although the semantics may seem very natural, the possibility of interpreting both (intuitively) static and dynamic formulae with respect to sequences *of any length* gives rise to a number of subtleties one only discovers after very careful studying. To give an example, it may be the case that $\mathbf{P}, \mathbf{D}_0 - - - \models \phi$ and $\mathbf{P}, \mathbf{D}_0 - - - \models \neg\phi$. The reason is that there may be more possible routes through the states, some of which satisfy ϕ and some of which satisfy $\neg\phi$. Note that it is not possible that both $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \phi$ and $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \neg\phi$.

4.1.3 Proof Calculi

The authors describe a proof procedure for the Horn version of $\mathcal{TR}$, called *serial-Horn* $\mathcal{TR}$. However, a constraining approach to specification often requires non-Horn queries. Work on a more general logic of state change with a sound and complete proof system has been announced in [46]. Translating statements of $\mathcal{TR}$ into this logic would yield an indirect proof procedure for transaction logic.

In the Horn version, the program must consist of *serial-horn rules*, that is, formulae of the form $a_0 \leftarrow a_1 \otimes a_2 \otimes \ldots \otimes a_n$, where all a_i are atomic formulae. A query is an existentially quantified serial conjunction, e.g., an expression of the form $(\exists X)(a_1 \otimes \ldots \otimes a_n)$. The data oracle must be a so-called "Generalized Horn Oracle." Finally, the program must be "independent" of the data oracle, which means that predicate symbols that occur in the rule heads must not occur in rule bodies in the data oracle.

Given these restrictions, an inference system is defined that consists of sequents of the form

$$\mathbf{P}, \mathbf{D}_1 - - - \vdash \mathrm{q},$$

where q is a serial-horn query. Such a sequent should be generated if and only if there is a sequence of states $\mathbf{D}_1 \ldots \mathbf{D}_n$ such that $\mathbf{P}, \mathbf{D}_1, \ldots, \mathbf{D}_n \models q$. The inference system consists of three rules that are: apply a transaction definition, query the data oracle, and perform an update (from the transition oracle). Deduction basically amounts to executing these rules. When reasoning in a forward direction (or bottom-up) we start with an initial database, and systematically update it to form a final database.

4.1.4 Operationalization

Execution of a $\mathcal{TR}$ program means finding a sequence of states that satisfies the query, given the program and the oracles (where we might be most interested in the last state in this sequence: the result of performing the query). Actually, it is possible, in principle, to read off such a sequence from a proof, but this is difficult since there is much freedom in a proof. The order in which the sequents appear in a proof is not fixed and sequents may appear in a proof which are not needed. *Executional deduction* is a restricted version of the inference procedure which does not allow this freedom. Formally, the order of sequents $seq_1, \ldots, seq_n$ is restricted by assuming that seq_i is obtained *only* from the previous sequent seq_{i-1}. Furthermore, the first sequent must be of the form $\mathbf{P}, \mathbf{D} \vdash ()$ and the last sequent must be of the form $\mathbf{P}, \mathbf{D}' - - - \vdash (\exists X)\phi$. By reading off the databases in the sequents in the reverse order (from the last sequent to the first), and only when the rule dealing with a database update has been used, we get the desired execution path. This restricted version is also sound and complete.

This executional deduction corresponds to the operational reading of Horn rules, where (in the forward direction) one either *applies* a transaction definition (and proceeds to prove the body), or one *queries* the database (by testing the truth of a (static) atom), or one *updates* the database (by applying an elementary transition). Thus, a transaction formula, in Horn form, is read declaratively as "the head is true or the body is false," and operationally as "to execute the head it is sufficient to execute the body." For instance, the while-loop expression given above is read operationally as "when we execute the while, either the test succeeds and is followed by do and another loop, or the test fails."

4.1.5 Running Examples

The conventions used in the $\mathcal{TR}$ formalization of *Propose & Revise* will also be adopted in the other formalizations, whenever appropriate.

Introduction. To specify *Propose & Revise* in transaction logic, we first need to determine which functions and (basic) predicates we will use to represent a state. For every store, we will define a predicate:

$$input(P, V),$$
$$output(P, V), \text{ and}$$
$$constraint(C, V_1, \ldots, V_n),$$

where P is a parameter name, V and $V_1, \ldots, V_n$ are values and C is a constraint name. The meaning of $constraint(C, V_1, \ldots, V_n)$ is that assigning $V_1, \ldots, V_n$ to the parameters $p_1, \ldots, p_n$ is not allowed, and this is part of the constraint C. Note that this formulation of a constraint requires a value for each parameter, even if only some of the parameters are constrained.

These predicates are all meant to be state predicates, which means that in all formulae we write, we should use $\langle output(P, V)\rangle$ (or $output(P, V) \wedge \mathbf{state}$). In order to avoid cluttering of (different sorts of) parentheses, we will omit these parentheses ($\langle\ \rangle$), but the reader should insert them around all static predicates. For a negated predicate, these parentheses must include the negation: $\neg\, output(P, V)$ should be read as $\langle \neg\, output(P, V)\rangle$. The formula $\neg\langle output(P, V)\rangle$ is equivalent to $\neg output(P, V) \vee \neg\mathbf{state}$, which is true on all sequences of length greater than 1, clearly not the intended meaning.

In principle, since every parameter has at most one value, we might have wanted to use a function (e.g., $input(P) = V$). Unfortunately, this is not possible in transaction logic, since functions are static, which means that they cannot change value between states. As the relations *input* and *output* are functional, we have to add an axiom to the program stating this:

$$(\forall P, V_1, V_2)[output(P, V_1) \wedge output(P, V_2) \rightarrow eq(V_1, V_2)].$$

Because equality is not a built-in operator in $\mathcal{TR}$, we must use $eq(V_1, V_2)$ as equality predicate, and moreover add all axioms of equality (reflexivity, symmetry, transitivity) to the program. These axioms are straightforward but necessary to get the intended models.

Conceptually, not every parameter needs to have a value in $input$. This would mean that the relation $input$ should be a partial function. However, as a constraint refers to all parameters, we have to make sure that every parameter has a value. Therefore, we introduce a special value, $undef$, where $input(P, undef)$ means that the parameter P does not have an input value. We will use constants p_i (with $1 \leq i \leq n$ to denote the set of n parameters. The condition that every parameter has an input and output value (possibly $undef$) is now formulated as:

$$\bigwedge_{i=1}^{n}(\exists V)(input(p_i, V))$$
$$\bigwedge_{i=1}^{n}(\exists V)(output(p_i, V)).$$

We have made one simplification in the above discussion: the variables in the formulae should be typed (e.g., V should be of type *value*). Transaction logic is not typed, but we can easily introduce unary predicates to describe such types. A predicate $value(v)$ for instance could denote that its argument is a value. The above formulae quantifiers should be changed to incorporate typing. For example, $(\exists V)(\ldots)$ should be replaced with $(\exists V)(value(V) \wedge \ldots)$. We have not done this in order to keep the formulae readable.

Functional Specification of `parametric design`. The `parametric design` task will be specified in transaction logic by a complex transaction we will call $parametric\_design$. First, we will give the functional requirements for this task, which will be specified as transaction formulae in the program.

First of all, the input may not be modified by $parametric\_design$ (PR1):

$$\bigwedge_{i=1}^{n}(\forall V)[(input(p_i, V) \otimes parametric\_design) \Rightarrow output(p_i, V)].$$

This means that if a parameter has some input value (in some state) and we perform $parametric\_design$, then afterwards the parameter should have the same value in the output.

The second requirement (PR2) is that the design must be complete:

$$parametric\_design \Rightarrow \bigwedge_{i=1}^{n} \neg output(p_i, undef).$$

To complete the specification, we should add the last requirement, stating that no constraint is violated (PR3):

$$parametric\_design \Rightarrow (\forall V_1, \ldots, V_n)\Big(\bigwedge_{i=1}^{n} output(p_i, V_i) \rightarrow \neg\exists C\ constraint(C, V_1, \ldots, V_n)\Big).$$

It is possible to leave the specification as it is at this point and keep the implementation of $parametric\_design$ out of the program. In this case, the program contains only these three rules, which constrain the possible models. In general, many sequences of states may satisfy this functional specification, and $parametric\_design$ may be true on sequences of different length.

In $\mathcal{TR}$, the problem solving method *Propose & Revise*, which implements the specification given above, can be given in two ways: within $\mathcal{TR}$, as treated in the next section, or outside $\mathcal{TR}$, using the oracles. Such an outside implementation will define a state oracle, which defines a state for each instantiation of $input$ and $constraint$ without $output$, as well as corresponding states with the $output$ computed. We define the transition oracle by:

$$parametric\_design \in \mathcal{O}^t(\mathbf{D}, \mathbf{D}')$$

for all pairs $\mathbf{D}$, $\mathbf{D}'$ that "do" $parametric\_design$. We could then prove that this implementation satisfies the functional requirements, for example:

$$\mathbf{P}, \mathbf{D} - - - \vdash \mathbf{arc} \otimes parametric\_design \otimes \neg \left[\bigwedge_{i=1}^{n} \neg output(p_i, undef) \right].$$

The standard predicate **arc** (that is true only on elementary transitions, i.e., paths of length 2), is useful to test all possible starting states. This query will only succeed if there is a sequence of states such that jumping to a random other state and then performing $parametric\_design$ violates the requirement. If no such path is found the programmer can be satisfied that the implementation works correctly. Of course, these queries depend on the existence of a general proof theory (not given in [46]) that can handle non-Horn requirements.

Conceptually, what we are doing above is to implement $parametric\_design$ outside the program (in the oracles). It is a feature of $\mathcal{TR}$ that it allows such external implementation, while at the same time allowing the functionality thereof to be verified.

Decomposition and Control of Propose and Revise. Above, all dynamics were implemented by the oracles and, thus, the dynamics in a sense remains hidden. In this section, we will specify the running example inside $\mathcal{TR}$, by implementing different elementary transitions in the transition oracle, namely, for $init$, $propose$, $test$, $revise$, $evaluate$, and $copy$, instead of $parametric\_design$. The relationship between $parametric\_design$ and these elementary transitions, i.e., the decomposition and control, is then specified by the transaction program, which we will give below. As might be expected, the decomposition of $parametric\_design$ is a while-loop.

$$\begin{aligned} parametric\_design &\leftarrow init \otimes pr\text{-}loop \otimes copy \\ pr\text{-}loop &\leftarrow propose \otimes \\ &\quad test \otimes \\ &\quad (\exists C violated(C) \Rightarrow revise) \otimes \\ &\quad (evaluate \otimes \\ &\quad evaluate\text{-}complete \vee \\ &\quad (\neg evaluate\text{-}complete \otimes pr\text{-}loop)). \end{aligned}$$

The predicates that are used in the program above are all defined either in other rules of the program or by the oracles. For instance, $test$ is a transition that deletes all expressions of the form $violated(C)$ and replaces them by the new set of violations. Also, $evaluate\text{-}complete$ is defined by the data oracle; we assume that in every state, it is either true or false, i.e., for every $\mathbf{D}$, we either have $evaluate\text{-}complete \in \mathcal{O}^d(\mathbf{D})$ or $\neg evaluate\text{-}complete \in \mathcal{O}^d(\mathbf{D})$. And, again, the reader should insert the $\langle\ \rangle$ parentheses around it, as we only want it to be interpreted on states. By just looking at the above rules, it is not possible to decide whether $test$ is an elementary transition, a static query, or a transaction that is in turn decomposed. From a modeling perspective, this kind of mixing of statics and dynamics is not a transparent way of specification: The specification obscures the difference between (static) declarative domain knowledge and (dynamic) methods used to reason about the domain. As we mentioned before, by descriptively naming transactions and static queries, part of this problem can be alleviated.

Particularly insightful is the expression stating that $revise$ must happen immediately after finding out that there are violations. What we want is an expression that is true on a sequence of length (at least) two, if there are violations, but in a single state, if there are no violations. This is why we use the double arrow in $\exists C\ violated(C) \Rightarrow revise$.

Having refined the initial (functional) specification, we would like to be able to prove higher level requirements (PR1 through PR3) *assuming* that lower level transitions are implemented correctly. We would like to test the decomposition and control without having implemented the lower levels (yet). What we want is to deduce that the decomposed program together with new requirements (or, functional specification) for each of the new transactions (see below) satisfies the higher-level requirements.

The `revise` Task. The decomposition in the previous paragraphs describing the decomposition and control of propose and revise uses a transaction, called $revise$, which should change occurrences of $design(P, V)$, i.e., the currently proposed values of the parameters, in order to fix the violations. We will make the assumption that the predicates $input$, $output$, and $design$ are state formulae, i.e., that they do not occur in the transition oracle, and that the state oracle contains each predicate, or its negation, for every state. As we did for $parametric\_design$, we can add formulae to the program stating requirements on this transition:

- The altered design is correct (R1):

$$revise \Rightarrow (\forall V_1, \ldots V_n) \Big(\bigwedge_{i=1}^{n} design(p_i, V_i) \rightarrow \neg \exists C constraint(C, V_1, \ldots, V_n) \Big).$$

- The task $revise$ does not propose values (R2):

$$\bigwedge_{i=1}^{n} (design(p_i, undef) \otimes revise) \Rightarrow design(p_i, undef).$$

- The input is respected (R3):

$$\bigwedge_{i=1}^{n} (\forall V)(input(p_i, V) \otimes revise) \Rightarrow design(p_i, V).$$

Finally, it is not allowed to set a parameter value to $undef$ if it had a value unequal $undef$ before (R4):

$$\bigwedge_{i=1}^{n} \neg \Big[(design(p_i, undef) \otimes revise) \Rightarrow \neg design(p_i, undef) \Big].$$

The `evaluate` Task. For $evaluate$, the following requirement is formulated:

$$evaluate \Rightarrow \left(evaluate\text{-}complete \leftrightarrow \bigwedge_{i=1}^{n} \neg design(p_i, undef) \right).$$

This solution clearly shows that if we want to make a distinction between statics and dynamics, we are forced to separate the predicates. The state predicate *evaluate-complete* is used to test the state and the dynamic predicate *evaluate* is used to control the reasoning process.

4.1.6 Conclusions

A nice thing about $\mathcal{TR}$ is that one can state both constraints and decompositions in the transaction program. The former constrain the possible models, whereas the decomposition (and the oracles) construct the implementation. Thus, $\mathcal{TR}$ nicely integrates constraining dynamics with constructive dynamics.

The use of oracles and a program allows successive refinement. At any moment during refinement, there will be a set of elementary transitions which can be implemented using the oracles, or using a functional specification. It is very easy to refine a specification by replacing the functional specification by decomposition rules in the program.

Transaction logic has both a declarative and an operational proof system for a Horn fragment. Work is in progress to obtain a proof system for the general case, by translating $\mathcal{TR}$ into a more general logic of state change.

One of the striking aspects in $\mathcal{TR}$ is that there is no division of static and dynamic predicates. This is considered an advantage by the authors as dynamics and statics are treated uniformly. However, most people studying dynamic phenomena in knowledge-based systems agree that the declarative knowledge about the domain should be clearly separated from the specification of the methods used to reason about the domain. It is possible to make a distinction between static (query) and dynamic (action) predicates by using a different naming scheme.

Some other disadvantages from the viewpoint of knowledge-based systems are also due to the fact that $\mathcal{TR}$ is geared toward specification of object-oriented databases. Control is scattered: Elementary transitions are defined in the transition oracle, composed transactions are defined in the program, and also (complex) queries may impose constraints on the dynamic behavior. The dataflow is not clear and states are sequence-based and not modularized.

4.2 Dynamic Database Logic

In this section, we discuss Dynamic Database Logic, as proposed in [22] and [21]. Our treatment is divided in two parts: We first discuss the propositional language PDDL [21], followed by a discussion of the first-order language DDL [22].

Both PDDL and DDL are intended as logics to reason about state and state change, particularly in database applications. As the names suggest, these logics are based on dynamic logic [15]. Before we go on to discuss these languages in detail, we briefly outline the relation between various languages based on dynamic logic. All these languages characterize states by truth values of predicates holding in a state. The languages differ in the way they characterize transitions between states. In the original proposal for PDL (Propositional Dynamic Logic) from Harel, transitions are only characterized implicitly. No operational definitions can be given for how to compute transitions. They can only be characterized by defining the properties of the original and final state of a transition: a formula such as $p \rightarrow [\alpha]q$ partially characterizes the transition α by stating that if α is executed in a state where p holds, and if α terminates, the result will be a state where q holds.

In the first-order version of dynamic logic (DL), transitions can be described explicitly: A state is characterized by the values of a set of variables. Consequently, in DL, operational definitions can be given for how to compute transitions: An elementary transition is defined by assigning a value to one of these variables (e.g., $x := 1$). Composed transitions can be defined by composing elementary transitions via sequence, iteration, and choice operators. Both the propositional language PDDL and the first-order language DDL follow DL by providing explicit definitions of transitions. However, whereas in DL, transitions can only be defined in terms of assignments to variables, PDDL and DDL define transitions in terms of truth-assignments to propositions (PDDL) or predicates (DDL). In this respect, DDL and PDDL are closely related to the more recent proposals for the language MLPM [8], which allows the explicit definition of transitions by assigning values to truth of predicates, and of MLCM [47], which also allows for the assignment of function values. We now turn to the discussion of PDDL and DDL, respectively.

4.2.1 The Propositional Language PDDL

PDDL features both passive updates and active updates (i.e., updates which trigger derivation rules). These atomic update actions can be combined by a regular language, as in PDL. In [22, p. 103], the authors claim a number of achievements for PDDL, the most important of which are the development of a semantics based on Kripke structures, a sound and complete proof system for "full" structures (i.e., structures containing a world for all possible valuations), and a Plotkin-style operational semantics.

Syntax. In PDDL, a database is described using a *database schema* (S, C, R), consisting of a signature S, a set of database constraints C, and a set of derivation rules R. The signature S defines the propositional symbols that can be used to formulate the derivation rules in R, which are program clauses as in logic programming, and to formulate the constraints in C. Predicates are divided in derived predicates (all predicates that occur as a head in some derivation rule) and base predicates (all other predicates).

In PDDL, a *database state* of a certain database schema is a set of sets of literals. A set of literals stands for the conjunction of its elements and is called a conjunction set. A set of such conjunction sets stands for the disjunction of its elements.

Update programs describe how to get from one database state to another. Update programs are regular expressions composed of *atomic updates*. PDDL defines two types of atomic updates: passive and active updates. For any atom p, the passive updates are $\mathcal{I}p$ (intuitively: "set p to true") and $\mathcal{D}p$ (intuitively: "set p to false"). For any atom p and set of

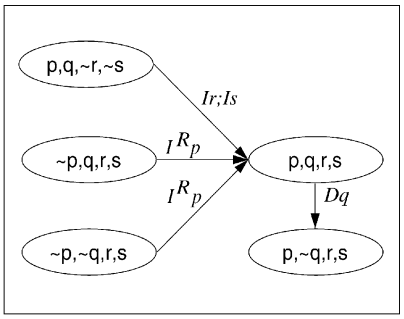

Fig. 5. Example PDDL model structure.

derivation rules R, the active updates are $\mathcal{I}^R p$ and $\mathcal{D}^R p$. $\mathcal{I}^R p$ sets p to true and then recomputes all the derived predicates of R (those occurring in the heads of clauses of R, where R is a definite logic program). Similarly, $\mathcal{D}^R p$ sets p to false and then recomputes all the derived predicates of R. These active updates are only defined for the base predicates (those predicates not occuring in the head of any clause in R). For the computation of derived predicates, the minimal Herbrand model for logic programs is used.

Update programs (composed transitions) can be formed from these atomic updates (elementary transitions) in the usual way (à la PDL) by sequence, choice, iteration, and test. For example, the transition $\mathcal{I}^R q; p?; \mathcal{I}r$ first sets q to true and derives new values for all the derived predicates, then tests whether p holds, and, if so, sets r to true. Formulae in PDDL are defined as in Propositional Dynamic Logic, containing modal formulae of the form $[\alpha]p$, where p is a formula, and α is any transition consisting of $\mathcal{I}$, $\mathcal{D}$, $\mathcal{I}^R$, and $\mathcal{D}^R$ and the composition operations. (The constraints C in a database schema (S, C, R) consist of these formulae.)

Declarative Semantics. The declarative semantics of PDDL is based on the notion of *possible worlds*. A possible world is defined as a truth assignment to all the predicates. The truth of modal formulae is determined by an accessibility relation between these worlds in the usual way, i.e., standard models in DL (the accessibility relations are determined by the set of possible worlds). Thus, the set of possible worlds plus the accessibility relation form a Kripke structure as usual.

As an example, consider the database schema with signature S, constraints C, and derivation rules R with

$$S = (p, q, r, s)$$
$$C = \{[\mathcal{I}^R p]q, [\mathcal{I}^R r]s, r \rightarrow s\}$$
$$R = \{p \rightarrow q\}$$

which states that the database contains formulae over p, q, r, s, and that q must hold after active insertion of p, s must hold after active insertion of r, and whenever r holds, s must hold. Note that the constraint $[\mathcal{I}^R p]q$ is obeyed automatically because of the derivation rule $p \rightarrow q$. The constraint $[\mathcal{I}^R r]s$ is not obeyed automatically because $r \rightarrow s$ is only a constraint, not a derivation rule. (Example taken from [21, p. 33].)

Fig. 5 is an example model of this database, in which some links between possible worlds are shown.

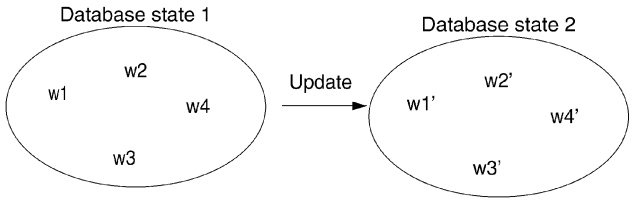

Fig. 6. Database states and worlds in PDDL.

Proof Calculus. Spruit et al. [21] presents a proof calculus for PDDL which is sound and complete for full Kripke structures, i.e., Kripke structures in which a world exists for every possible combination of truth values for all atoms in the signature. For example, the structure in Fig. 5 is *not* full, since (among others) there is no world for $\neg p, \neg q, \neg r, \neg s$. Spruit et al. [21] also show that the proof system is sound and complete for structures in which all given constraints are satisfied and no world with a valuation different from the valuations of the worlds already in the structure can be added without violating some constraint. The proof system for PDDL is built by taking an existing proof system for PDL and adding axioms that describe the behavior of the atomic update actions. For example, the axioms for passive insertion read:

$[\mathcal{I}p]p$	insertion axiom
$q \rightarrow [\mathcal{I}p]q$	positive frame axiom
$\neg q \rightarrow [\mathcal{I}p]\neg q$	negative frame axiom
$\langle \mathcal{I}p \rangle true$	successor existence
	(Note : full structures only!).

Similar axioms hold for active insertion and the deletion operators.

Operational Semantics. Whereas the declarative semantics states the required accessibility relation between individual worlds, as given above, the operational semantics provides the required relations between entire database states (see Fig. 6). Remember that in PDDL, a *database state* of a certain database schema is a set of sets of literals. A set of literals stands for the conjunction of its elements and is called a conjunction set. A set of such conjunction sets stands for the disjunction of its elements.

This operational semantics is given in the form of transition rules between database states, such as:

$$\langle \mathcal{I}p, \sigma \rangle \rightarrow \{(c \setminus \{\neg p\}) \cup \{p\} \mid c \in \sigma\}$$

which states that via $\mathcal{I}p$, we move from a database state σ to a database state where all worlds in σ are changed by enforcing p. There is a form of equivalence between the declarative and the operational semantics which states that an action α results in the transition from a database state σ to a database state σ' *iff* for all worlds $w' \in \sigma'$, there is a world $w \in \sigma$ such that $(w, w') \in m(\alpha)$, where $m(\alpha)$ is the function that assigns to each update program α its interpretation as an accessibility relation on possible worlds.

4.2.2 The First-Order Case

Spruit et al. [22] and [48] present DDL, a first-order generalization of PDDL which has as additional aim the

possibility of object creation and parallel updates. In this section, syntax and semantics of DDL will be presented.

Syntax. In DDL, the atomic updates are of the form "$\&\{x_1,\ldots,x_n\}\mathcal{I}p(t_1,\ldots,t_n)$ where ϕ" (called conditional insertion), with the intended meaning that $p(t_1,\ldots,t_n)$ is set to true for all values of $x_1,\ldots,x_n$ that make ϕ true. Similarly, "$\&\{x_1,\ldots,x_n\}\mathcal{D}p(t_1,\ldots,t_n)$ where ϕ" (called conditional deletion) sets $p(t_1,\ldots,t_n)$ to false for all values of $x_1,\ldots,x_n$ that make ϕ true. These two operations can be combined into the update operator $\mathcal{U}$, which "copies" the truth values of ϕ to p: "$\&\{x_1,\ldots x_n\}\mathcal{U}p(t_1,\ldots,t_n)$ where ϕ" sets $p(t_1,\ldots,t_n)$ to true for all values of $x_1,\ldots,x_n$, where ϕ is true and to false where ϕ is false. This operator can be defined in terms of insertion and deletion as follows:

$$\begin{aligned}&\{x_1,\ldots,x_n\}\mathcal{U}p(t_1,\ldots,t_n) \text{ where } \phi\\ &\equiv \&\{x_1,\ldots x_n\}\mathcal{I}p(t_1,\ldots,t_n) \text{ where } \phi;\\ &\quad \&\{x_1,\ldots x_n\}\mathcal{D}p(t_1,\ldots,t_n) \text{ where } \neg\phi.\end{aligned}$$

Because of this definition in terms of $\mathcal{I}$ and $\mathcal{D}$, the $\mathcal{U}$ operator does not affect any of the semantic properties of the language.

Conceptually, these are *parallel* updates, which update multiple instances of $p(t_1,\ldots,t_n)$ at once. In contrast, the expression "$+(x_n,\ldots,x_n)\alpha$, where ϕ" executes α for one of the possible values of $x_1,\ldots,x_n$ which make ϕ true. As before, composed transitions are formed from elementary transitions by sequence, test, iteration, and choice.

The combination of parallel updates and conditional choice make DDL very close to the more recently proposed language MLPM [8], with the parallel updates corresponding to MLPM's λ operator, and the conditional choice corresponding to MLPM's ϵ operator. Notice that the active updates of PDDL are no longer present in DDL. Presumably, the parallel updates are considered a sufficiently powerful replacement.

Although the semantics of DDL assumes a (single) domain for the interpretation of constants (DDL does not have functions), which is the same for all possible worlds, object creation can nevertheless be modeled by introducing a special existence predicate E. Object creation can then be expressed as:

$$+\{x\}\mathcal{I}E(x) \text{ where } \neg E(x).$$

Semantics. The semantics of DDL is similar to that of PDDL, with possible worlds defined as truth assignments to predicate symbols, and accessibility relations between the possible worlds for each of the elementary and composed transitions. The ontology of possible worlds as truth assignments to predicates should be contrasted with the semantics of classical dynamic logic, where possible worlds are instead variable assignments, and where the interpretation of predicate symbols is fixed across all worlds.

Proof Calculus. As with PDDL, an axiomatization for DDL is built from a standard axiomatization for dynamic logic, extended with axioms for DDL's new operators. Again, this axiomatization is sound and complete only for full structures (and domains have to be finite). A small example shows how to use DDL for a correctness proof of a particular update program α that copies the extension of a unary predicate q to another unary predicate p. The specification for this program is given as

$$\begin{aligned}SPEC \;\equiv\; &\forall y(q(y)\leftrightarrow[\alpha]q(y))\wedge\\ &[\alpha]\forall y(q(y)\leftrightarrow p(y))\end{aligned}$$

(i.e., the extension of q remains unchanged by execution of α, and after the execution of α, p has the same extension as q). If we define the program α as:

$$\alpha \equiv \&\{x\}\mathcal{D}p(x) \text{ where true}; (\&\{x\}\mathcal{I}p(x) \text{ where } q(x)),$$

then we can prove that $SPEC$ holds for the above definition of α. (Example from [22, p. 114].)

4.2.3 Running Example

The discussion of the running example in DDL starts with some general remarks on the basic structure of the example in DDL and on how information on types is represented in DDL.

Remarks (Basic Idea). The basic idea of the specification is to treat all input and output roles of an inference action as predicates. Inference actions can then be formalized as *update programs*, which make assignments to their output roles.

For example, an inference action $union$, with input roles $in_1(X_1), in_2(X_2)$ and output role $out(Y)$ can be specified as assigning "true" to all instances of the output role $out(Y)$ for which Y is the union of the two input roles:

$$\begin{aligned}union \equiv\; &\&\{Y\}\ \mathcal{I}\ out(Y)\\ &\textbf{where } \exists X_1,X_2 : i_1(X_1)\wedge i_2(X_2)\wedge\\ &\qquad \forall x(x\in Y\leftrightarrow x\in X_1\vee x\in X_2).\end{aligned}$$

All this is very close to some existing literature on formalization of KBS, such as [49] and the simplified version thereof in [50].

In general, for an inference action T, there will be a functional specification $T_{spec}(X_1,\ldots,X_n,Y)$ of its input/output-relation. We then need an implementation T_{impl} of the inference action such that: if $X_1,\ldots,X_n$ are values for the input roles, and the functional relation holds between $X_1,\ldots,X_n$ and Y, then after execution of T_{impl}, Y is a value of the output role:

$$\begin{aligned}&in_1(X_1)\wedge\ldots\wedge in_n(X_n)\wedge\\ &T_{spec}(X_1,\ldots,X_n,Y)\leftrightarrow\ \langle T_{impl}\rangle out_1(Y).\end{aligned}\tag{2}$$

A trivial implementation of T_{impl} would be:

$$\begin{aligned}T_{impl} \equiv\; &\&\{Y\}\ \mathcal{U}\ out_1(Y)\\ &\textbf{where } in_1(X_1)\wedge\ldots\wedge in_n(X_n)\wedge T_{spec}(X_1,\ldots,X_n,Y)\end{aligned}\tag{3}$$

because (2) would follow immediately. Of course, instead of this, we would rather want an implementation which realized T_{spec} by implementing it by program expressions, instead of simply enforcing it as a postcondition as in the above.

Types. To simplify the presentation of the running example in DDL, we treat DDL as if it were a typed logic. This can be simply encoded by introducing unary

$$
\begin{array}{l}
parametric\_design(V_1, ..., V_n) \\
\leftrightarrow \\
\bigwedge_{i=1}^{n}(\quad output(p_i, V_i) \wedge & \% \ V_1, ..., V_n \text{ are output parameters} \\
\quad \neg\exists V' : (output(p_i, V') \wedge V_i \neq V') \wedge & \% \text{ their values are unique,} \\
\quad V_i \neq undef \wedge & \% \text{ unequal to undef (Req. PR2),} \\
\quad (input(p_i, V_i) \vee & \% \text{ and not overriding the} \\
\quad\ input(p_i, undef))) & \% \text{ user input (Req. PR1)} \\
\quad \wedge \quad \neg\exists Viol : & \% \text{ no constraint may be} \\
\quad\ constraints(Viol, V_1, ..., V_n)) & \% \text{ violated (Req. PR3)}
\end{array}
$$

Fig. 7. Functional specification of the task `parametric design` in DDL.

$$
\begin{array}{l}
\% \text{ don't make parameters undefined (Req. R4):} \\
design(P, V) \wedge V \neq undef \rightarrow [revise] design(P, V') \rightarrow V' \neq undef \\
\% \text{ don't revise undefined parameters (Req. R2):} \\
design(P, undef) \rightarrow [revise] design(P, undef) \\
\% \text{ don't revise user-defined parameters (Req. R3):} \\
input(P, V) \rightarrow [revise] input(P, V) \wedge design(P, V) \\
\% \text{ the new values no longer cause a violation (Req. R1):} \\
[revise] \neg\exists Viol' : \exists V_1, ..., V_{j-1}, V_{j+1}, ..., V_n : \wedge_{i \neq j} design(p_i, V_i)
\end{array}
$$

Fig. 8. Functional specification of the *Revise* inference action in DDL.

$$
\begin{array}{ll}
P\&R \equiv & \% \text{ Clear the roles for output and intermediate designs, and} \\
& \% \text{ initialise the input:} \\
& init; \\
& \quad \textbf{repeat} \quad propose; \\
& \qquad\qquad test \\
& \qquad\qquad\ \textbf{if} \quad \exists V : violations(V) \\
& \qquad\qquad\ \textbf{then} \quad revise \\
& \qquad\qquad\ \textbf{endif} \\
& \quad \textbf{until} \quad evaluate; \\
& \% \text{ and finish by copying the results to the output role:} \\
& \&\{P, V\} \ \mathcal{I} \ output(P, V) \ \textbf{where} \ design(P, V)
\end{array}
$$

Fig. 9. Implementation of *Propose & Revise* in DDL.

predicates for all the types in the standard fashion. We will leave these types implicit in the variable names. Thus, a formula like

$$\forall P \exists V : output(P, V)$$

should be read as

$$\forall P \exists V : parameter(P) \rightarrow value(V) \wedge output(P, V),$$

in order to include all the required type-restrictions. This translation is entirely mechanical and does not change the logical expressiveness of the language. It is similar to our treatment of $\mathcal{TR}$ in Section 4.1.

The `parametric design` Task. Fig. 7 presents the functional specification of the task `parametric design` expressed in DDL. In order to be able to give an implementation for this functional specification, the inference actions have to be specified. We will only give a functional specification of the inference action *revise*.

The *Revise* Inference Action. Fig. 8 presents the functional specification of the inference action *revise*. As a conjunction, these four conditions together form the formula T_{spec} from (2) for the inference step *revise*.

Implementation of *Propose & Revise*. Using the programs for the smaller inference actions, we can specify the overall program for *Propose & Revise* by almost literally transcribing the control flow from Fig. 4 in Section 3. The overall program for *Propose & Revise* is presented in Fig. 9. Note that although constructs like "repeat...until" and "if...then...endif" (used in the specification in Fig. 9) are formally not part of the syntax of DDL, they can easily be defined. Notice the special use of the inference action *evaluate*: It functions as a predicate (in this as the stop-condition for the loop) and does not do any updates. This corresponds to the fact that in the knowledge flow diagram (Fig. 3) evaluate has no outgoing arrows. Its sole purpose is to control the loop, not to produce any output.

Concluding. Notice the striking resemblance between the above control-loop and the description of the control loop in our example in Fig. 4, Section 3. We suspect that this is largely due to the fact that DDL is based on Dynamic Logic, which was also the basis for the KBS specification languages KARL [3] and $(\mathrm{ML})^2$ [7]. This meant that the

specification of the running example in Section 3 is already written largely "PDDL-style," and very little further changes remained to be made.

4.2.4 Discussion

There is a remarkable similarity between PDDL and the language MLPM [8], which was independently developed specifically for specifying the control of KBS. This similarity was also apparent from the similarity between the informal description of our example in Section 3 and the PDDL formalization given above. This suggests closer links between the control of deductive databases on the one hand and knowledge bases on the other hand than have been investigated until now. The most important difference between PDDL and MLPM lies in the active updates $\mathcal{I}^H p$ and $\mathcal{D}^H p$ which are present in PDDL, but are lacking from MLPM.

Finally, the origin of PDDL as a database specification is apparent in the restriction that only literals can be added or deleted from a database state. This is perhaps a reasonable restriction for databases, but for knowledge-bases one would want to add and remove more complex constraints such as $p \rightarrow q$, etc.

4.3 Abstract State Machines[5]

The Abstract State Machines (ASM) approach, in the original form as proposed by Gurevich in 1988, was an attempt to provide operational semantics to programs and programming languages by improving on Turing's Thesis.[6] The problem with Turing Machines is that they are low-level, so that the description of algorithms other than toy examples is almost infeasible. The Abstract State Machines approach, however, makes it possible to specify algorithms at any level of abstraction. Using successive refinement, it is possible to investigate properties of the algorithm (like correctness) at any of these levels. In this fashion, one can give operational semantics to programming languages, architectures, protocols, etc., and this has been done extensively (see references in [23], [51] and the special ASM issue of the JUCS [52]). Although not the primary goal, the ASM approach is now often used as a *specification* formalism.

4.3.1 Basic Syntax and Semantics

We will briefly describe the notion of an abstract state machine, restricting ourselves to the bare essentials (in various work on ASMs, various definitions are given and many extensions of the basic framework exist). A signature Σ is a finite collection of function names, each with an associated arity. A (static) algebra $\mathcal{A}$ of signature Σ is a set (called the *superuniverse*) together with interpretations of the function names on this set. An abstract state machine consists of a signature and a *program*, which is a set of *transition rules*. These rules describe how transitions between (static) algebras of signature Σ can occur. Basically, a transition rule is an expression of the form $f(\bar{t}) := t_0$, where f is a function symbol, $\bar{t}$ is a tuple of terms (in the signature Σ) of length the arity of f, and t_0 is another term. Such a rule is fired in an algebra $\mathcal{S}$ by evaluating the terms $\bar{t}$ and t_0 (to, say, the tuple $\bar{a}$ and a in the superuniverse of $\mathcal{S}$) and by setting the value of f in $\bar{a}$ to a, obtaining a new algebra $\mathcal{S}'$. A so-called *sequence*[7] of such rules can be fired by firing them all simultaneously, provided this can be done consistently (this means there should not be two rules which try to update the same function in the same argument to two different values—we will give an example later on). A run of an abstract state machine is a sequence $(\mathcal{S}_0, \mathcal{S}_1, \ldots)$ (possibly finite) of static algebras of signature Σ such that $\mathcal{S}_{i+1}$ is the result of firing the rules in $\mathcal{S}_i$. In addition, an ASM may contain a static algebra of signature Σ (the initial state) and a set of static algebras of signature Σ, the final states. In that case, a run must begin with the initial state, and none of the states, with the exception of the last (if there is one) is allowed to be a final state. In a finite run, the last state must be a final one. The simultaneous transitions are somewhat similar to *rewrite logic* [53], which serves as a semantics for the Maude specification language [54] and is proposed, e.g., in [55], as an alternative semantics for temporal logic based specification languages for object-oriented systems, such as TROLL.

In a ASM specification, in principle, only the transitions are defined. This specifies possible runs of the abstract state machine, which, as stated above, consist of consecutive algebras constrained by the transitions defined in the specification.

4.3.2 Simple Extensions

Of course, this basic framework is very simple, making proofs about (general) properties of abstract state machines simple. However, in applications one would like to have a richer language for describing algebras and transitions between them, so for the ease of "programming," a number of further elements are introduced. Each signature is assumed to contain nullary function names *true*, *false*, and names for the usual Boolean operations. Relations can be introduced by identifying them with their characteristic functions. Each signature must also contain the equality sign. Unary relations can be used to define *universes* (sorts). The nullary function name $undef$ is used to be able to introduce partial functions. Boolean terms can be built by means of Boolean operations on relation terms. This allows *guarded transitions*, of the form

$$\textbf{if } g \textbf{ then } R,$$

where g is a Boolean term and R is a set of transition rules. Universes can be expanded by taking elements of the special universe *Reserve*, which occurs in every signature.

Example. The following expressions are two simple transition rules, where the second one is guarded:

$$\mathrm{age}(\texttt{Bert}) := 0,$$

$$\textbf{if } \mathrm{age}(\texttt{Ernie}) \leq 150 \textbf{ then } \mathrm{age}(\texttt{Ernie}) := \mathrm{age}(\texttt{Ernie}) + 1.$$

5. Abstract State Machines were formerly called *Evolving Algebras*.

6. A lot of information about Abstract State Machines can be found at http://www.eecs.umich.edu/gasm and http://www.uni-paderborn.de/cs/asm.html.

7. The word *sequence* is misleading here. Gurevich proposes *block*. *Set* would also have been a more appropriate term.

Given a static algebra, the rules are fired as follows: The first rule sets the age of `Bert`, which is a constant, to zero in the new algebra. To fire the second rule, first the constant `Ernie` is evaluated in the current algebra, then age(`Ernie`) is evaluated (again in the current algebra). If the resulting value is less than or equal to 150, age(`Ernie`) will be one higher in the new algebra. A conflict might arise here. If `Bert` and `Ernie` evaluate to the same element in the current algebra, the rules are inconsistent, and the new algebra is equal to the current one.

4.3.3 Further Extensions

The basic paradigm of Abstract State Machines has been extended to handle nondeterminism through the *Choice* construct which chooses an element from a universe nondeterministically. Also, *variables* can be allowed in rules. A rule with a variable (which may occur in the guard) is executed by firing the rule for all possible instantiations of the variable in parallel. One last extension we will mention here are *distributed* ASMs, in which a finite number of agents are allowed, each with their own *module*. Many other extensions exist, too many to mention here.

Interaction with the external world is handled by introducing *external* functions. These functions are not defined or updated in the ASM, but whenever it needs the value of an external function, it can ask the external world. Typically, this is input provided by the user, or it models interaction with other components of a system. An external function can be "implemented" by another ASM.

External functions are also used as a mechanism for abstraction: Any functionalism one does not wish to specify can be hidden in an external function. This functionalism can be expressed as constraints on the external function (using general mathematical terms). When refining a specification, such a function can be made internal and can be manipulated within the ASM.

4.3.4 Proofs and Operationalization

The ASM approach does not come equipped with a (fixed) proof calculus. Properties of abstract state machines can be proved informally, using standard mathematical techniques. Mathematical proofs can of course always be verified (should one desire) by any proof checker for first-order logic. This liberal view is certainly satisfactory when viewing ASMs as improved Turing Machines—the latter do not have a standard logic with proof checkers either. Using the ASM approach as a specification mechanism, however, this position has some drawbacks. If automatic verification of properties is desirable (and many users of formal specification languages feel this is the case), then one would like to have tools that can automatically translate a specification into some (fixed) logical formalism and prove properties of the specification, expressible in this same formalism. Indeed, work is being done on formal proof systems for ASMs, using various proof tools. For instance, [56] use the Karlsruhe Interactive Verifier (KIV, a proof assistant), while [57] use PVS and [58] uses SMV, which are both model checkers.

Operationalization of ASMs is relatively straightforward. Basically, one just needs a mechanism that continually fires the applicable rules. An abstract machine for ASMs is given in [59], and [60] gives a very simple interpreter. An example of a concrete language for ASM specifications which can be executed is DASL ([61]), which also includes polymorphic types and equational specifications. At the University of Paderborn, work is done on a specification and design environment.[8] At the German National Research Center for Information Technology, a compiler for ASMs has been developed.[9] Another interpreter is developed at the University of Michigan.[10]

4.3.5 Running Example

Since there is no notion of subroutine or procedure in ASMs, the entire example will be one long specification. (In some interpreters, there are ways to use subroutines.) It is possible to perform successive refinement using external functions. This allows the hiding of functionality in functions which are specified outside the ASM paradigm. In order to show the structure of the full specification, we will not use these functions.

In the specification of the signature of an ASM, there is no structuring mechanism, so there is just one big signature. We omit a complete formal description of the signature, describing only the parts of the input and the design. Since there are only functions in an algebra, we have to describe predicates using Boolean functions. In ASMs, there are, in principle, no sorts or universes; they have to be simulated using unary predicates, which have to be described using a function. In later documents on ASMs, universes are mentioned, but without a formal syntax to describe them. Therefore, we will make no distinction between universes (which are subsets of the superuniverse), and their characteristic functions. The part of the signature for the input consists of a unary function `Is_value` which should be given Boolean values only: It denotes whether the argument is of the right sort. Furthermore, it contains a binary function `Input`, which should also only have Boolean values. The signature contains an extra binary function `Design` with Boolean values.

The structure of the specification is given in Fig. 10. We use expressions like <`Initialize`> to denote a set of rules which specifies the behavior of the ASM when it is initializing. In the final specification, corresponding rules should be inserted here.

In the specification of an ASM, there should also be a way of identifying whether we are in the initial state. In various documents on ASMs, this is handled differently. One of the approaches is the following: There is a special Boolean function constant `Start` which is true in the starting state only. The rules that should only fire in the initial state can use this constant. We have followed this approach in our specification.

As is the case for Turing Machines, in the ASM approach, there are no explicit programming constructs for loops and subroutines. In the specification of control, one can only use guards in the transition rules to make sure they fire only when needed. We use constants to keep track of what we

8. http://www.uni-paderborn.de/cs/asm/ASMToolPage/asm-workbench.html.
9. http://www.first.gmd.de/~ma/aslan.
10. ftp://ftp.eecs.umich.edu/groups/gasm/interp2.tar.gz.

```
if Start then
        Var v ranges over Is_value
                Output(p_1,v) := false
                    ....
                Output(p_n,v) := false
                Design(p_1,v) := false
                    ....
                Design(p_n,v) := false
        Mode := initializing
endif
<Initialise>
<Propose>
<Test>
if (Mode = mainloop & Doing = check_if_revise_needed) then
        if ((exists v in Is_violation) Violations(v) = true) then
                Doing := revising
                Revise_mode := begin_revise
        else
                Doing := evaluating
        endif
endif
<Revise>
<Evaluate>
<Copy>
```

Fig. 10. Structure of the ASM specification of *Propose & Revise*.

```
if (Mode = mainloop & Doing = testing) then
        Var viol ranges over Is_violation
                if  (exists v1, ..., vn in Is_value)
                   (Design(p_1,vn) = true &  ...  & Design(p_n,vn) = true
                    & Constraints(viol,v1, ..., vn) = true) then
                        Violations(viol) := true
                endif
        Doing := check_if_revise_needed
endif
```

Fig. 11. ASM specification of the test inference action.

are doing, and use these constants in the guards of transition rules. The first constant is `Mode` to keep track of where we are in the main loop of *Propose & Revise*. Its possible values are `initializing`, `mainloop`, and `copying`. A second control variable, `Doing`, is used for control inside the main loop. Its possible values are `proposing`, `testing`, `check_if_revise_needed`, and `revising`. These control variables are to be used and updated by the rules belonging to `Initialize`, `Propose`, `Test`, `Revise`, and `Copy`. So, for instance, all rules for `<Propose>` should be of the form:

```
if (Mode = mainloop & Doing = proposing & conditions
        then updates
endif
```

One of these rules should set `Doing` to `testing` if the proposing phase is finished. We will show what this looks like for `Evaluate`.

Evaluate. The rules for `Evaluate` should only fire if the control variables indicate that we are evaluating. They should test whether the design is complete, and, if so, set the `Mode` to `copying`.

```
if (Mode = mainloop & Doing = evaluating) then
        if Design(p_1, undef) = true or ... or
          Design(p_n, undef) = true then
                Mode := proposing
        else
                Mode := copying
        endif
endif
```

Test. The rule for `Test` presented in Fig. 11 illustrates the variable mechanism. The variable `viol` is instantiated, in parallel, to every element in the algebra for which `Is_violation` holds and, for this value, the rule inside the `Var` construct is fired.

We will not give the functional specification of `Revise` since this can not be done *within* the ASM framework. It is possible to define an external function for `Revise`. Such a function can be described using any specification

mechanism, including a logical description as used in the requirements R1 through R4 in Section 3.

4.3.6 Discussion

Summarizing, a state in an abstract state machine is an algebra, admitting very rich structures. Such a state can be described by Boolean expressions built up from relations (or actually their characteristic functions). A transition between two states is induced by the firing of appropriate transition rules of a very simple form. The control over a run of an ASM is simple: As long as the current state is not a final state, all applicable rules are fired simultaneously (a guarded transition is only applicable if the guard—the Boolean term—evaluates to true). If a conflict between rules (e.g., the updating of a function value to two distinct values) arises, the algebra is not updated at all: The next algebra in the run is the same as the current one (the ASM is in an infinite "loop"). Thus, the control is local, that is, we can only specify the control over elementary transitions, not over entire runs of the system. The rules give a constructive way to calculate the next state (algebra). Although in this fashion a declarative description of *what* to do *when* is given, it is in general not trivial to explicitly describe control over runs (similar to the case of Turing Machines). As the control is implicit in the rules, the ASM approach does not seem to be the first choice for the specification of control. This should not come as a surprise, since initially, specification was not the main goal of abstract state machines. Also, there is no formal proof system for specifications of abstract state machines. Proofs of correctness are informal (but, possibly, rigid; see [62]). Although tools for running abstract state machines exist which allow "observation and experimentation" for establishing correctness ([23]), a formal proof system would allow automated proofs for use in verification and validation.

Given the rich structure of an algebra, it is certainly possible to describe knowledge (Boolean operators are present in any algebra) and to define transition rules for reasoning, but the control over reasoning remains implicit, or has to be made explicit by the designer using control parameters (as we have done in the running example). Also, the main means for functional specification and abstraction is provided by external functions. Such external functions can only be specified *outside* the ASM framework (using an (in)formal mathematical definition, a programming language or a specification formalism like Z [63]). There are, however, some other means for functional specification and abstraction. It is common in ASMs, for example, to define a concrete internal function which works over some abstract set, and then in later revisions to replace the abstract set with more concrete sets upon which functions in the ASM can operate.

4.4 TROLL and OSL

The specification language TROLL is aimed at the specification of object-oriented information systems. TROLL is based on many-sorted first-order temporal logic, and its semantics is defined by translating TROLL constructs to *Object Specification Logic* [64]. There are various versions of the language: TROLL [19], TROLL*light* [65] and the version described in Jungclaus' thesis [66], on which our discussion is based.

4.4.1 Syntax of TROLL

TROLL provides a very rich syntax, aimed at a user-friendly way of specifying object-oriented systems. In the current section, we will first briefly describe some key ideas of object-oriented specification. After that, we will describe the syntax of TROLL. Due to space restrictions, only a few key ideas of the TROLL syntax will be touched upon here. Together with the pieces of example TROLL code given in Section 4.4.4 below, these ideas should give a flavor of TROLL specifications.

The general idea in object-oriented systems is to model the universe of discourse as a collection of objects, each of which encapsulates a local state and behavior, and can be distinguished using some unique identifier that is immutable during its lifetime. The (description of) the global state is thus distributed over these objects. The global behavior of the modeled system as a whole emerges from the combined local behavior of its constituent objects, governed by interaction relationships. Some additional structuring concepts in object-oriented systems are (taken from [66]):

Classification. For the purpose of abstraction, objects with the same characteristics and behavior are grouped in classes.

Specialization. A specialization is a more detailed view on the same conceptual entity (represented by an object or a class of objects). Specialization is a static notion, except for roles:

Roles. During their lifetime, objects can play different roles, which are temporary specializations.

Aggregation. Objects can be joined to form composite objects called aggregations.

At a bird's-eye view, a state in TROLL is described by a (structured) collection of values of the *attributes* (i.e., the observable properties) of objects. Elementary state transitions are caused by the occurrence of *events*, which can be combined to form composed transitions, like in $\mathcal{TR}$ (see Section 4.1). Before describing the high-level structuring concepts from TROLL, we will first introduce the four basic languages on which TROLL specifications are built. These basic languages are:

Data language. This language describes the possible attribute values in the universe of discourse. The data language consists of terms in a sorted first-order language. There are special terms for events, called event terms (an example will be given below).

State formulae language. Object states are described by first-order formulae over the terms of the data language. There is a special predicate $\mathbf{occurs}(e(t_1, \ldots, t_n))$, defined on event terms $e(t_1, \ldots, t_n)$ intended to mean that the event e is about to occur.

Temporal language. The temporal basic language, which consists of a past tense part and a future tense part, expresses temporal relationships between state formulae. For example (if `m` is of sort `money_type`), then the expression ($\mathbf{occurs}(\texttt{withdraw(m)})$ **and** $\texttt{balance} = \texttt{m}$)

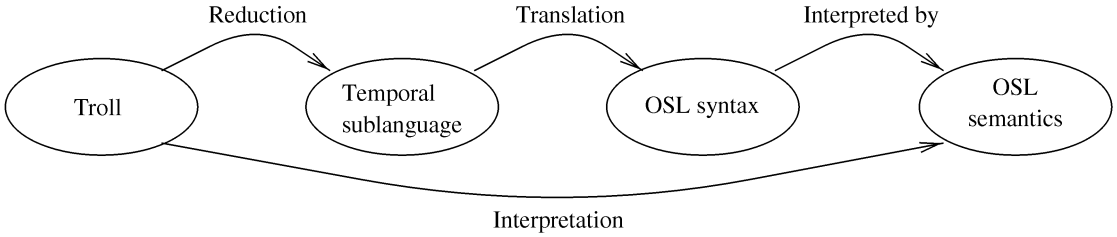

Fig. 12. The semantics of TROLL.

implies (**next** `balance` $= 0$) states that if in the current state a withdrawal event occurs (withdrawing an amount `m`), and the balance equals `m`, then in the next state the balance is zero.

Pattern language. The pattern language is used for ordering event terms of a set, in a way resembling process algebra formalisms like CCS [67] and ACP [17]. There are three operators: sequential composition (->), choice (**select** ... **end select** and **case** ... **esac**) and recursion.

We are now ready to introduce the basic structuring mechanism of TROLL: templates. A template is a generic description of an object, providing a local signature declaration, which declares the attributes and event symbols, and a local specification, which specifies the admissible behavior and attribute values for instances of the template. The syntax for the local specification is very rich: Using the basic languages, it is possible to specify the effects of event occurrences, static or dynamic constraints on the values of attributes, commitments and obligations to perform certain events, and patterns describing an instance's behavior in a constructive way. All this is illustrated by our running example below.

A template is not the same as a class. A class is regarded as a collection of objects described by the same template. In the TROLL syntax, a class definition specifies this collection, or, in other words, it describes the potential extensions of the class. The difference between a class and a template is that a class adds to a template a system for identifying instances of that template.

A TROLL specification consists of a number of class definitions and specializations, role definitions, relationships, and interfaces based on the class definitions. These additional concepts are not discussed here. However, we will end this discussion of the syntax of TROLL with a few remarks needed to read the examples.

Event effects. In the local specification part of a template, there are several ways to specify the effect of an event occurrence. Here, we discuss two ways, using the following TROLL code:

```
effects
    [event_id] attr = value;
constraints
    occurs(event_id) implies next attr = value;
```

An expression like the one in the first two lines can be used to relate the occurrence of an event to the value of an attribute. The expression given should be read as: if the event `event_id` occurs in the current state, then in the next state the value of `attr` equals `value`. The same effect can be reached using a dynamic constraint, illustrated in the last two lines. In general, the keyword **constraints** is followed by a temporal formula.

Object interaction. So far, only local behavior has been discussed. Relations between event occurrences in different objects can be specified using interactions as follows:

```
interactions
    object1.event_id1 ≫ object2.event_id2;
```

The expression `object1.event_id1` ≫ `object2.event_id2` denotes event calling, and means that whenever `event_id1` in `object1` occurs, the `event_id2` in `object2` occurs.

4.4.2 Semantics of TROLL

The semantics of TROLL is based on *Object Specification Logic* (OSL, see [64]), which is a temporal logic for reasoning about objects. In Fig. 12, the relation between TROLL, OSL, and their semantics is depicted. On the left hand side, the TROLL syntax and its sublanguages are shown. The formal semantics of TROLL is given in [66] as a translation to OSL formulae. This corresponds to the arrow from TROLL to OSL in Fig. 12. This arrow departs from TROLL's temporal language, as that language is closest to OSL, and many parts of TROLL are actually first translated to the temporal sublanguage, which in turn is trivially[11] translated to OSL.

OSL itself is interpreted over sequences of states, as explained in the rest of this section. Thus, these sequences of states serve as the semantics of OSL. Indirectly, they are also the semantics of TROLL.

Object Specification Logic. OSL consists of two levels: a local and a global level. The local level is concerned with the definition of the local state and behavior of an object (the description of which is called an *aspect* of an object), specified by aspect templates in a local specification language. At the global level, the different aspects are related, forming specializations and aggregations. In this section, we will adopt the notation and terminology used in [66], which differs from the notation in [64]. First, we present the local specification language, used for specifying object aspects. After that, the semantics of the local level is given. We then proceed to discuss morphisms, which

11. Not completely trivial, as OSL only contains future tense temporal operators. The past tense part of TROLL is translated to future tense OSL formulae relative to the beginning of time.

connect the local and the global level. Finally, we discuss the global level and its semantics.

Local Syntax of OSL. At the local level, states and transitions are described using many-sorted first-order local temporal predicate languages. These languages are defined using (local) signatures $\Theta = (\mathcal{IS}, \alpha, ATT, EVT)$. A local signature consists of a partially ordered set[12] $\mathcal{IS}$ of indentifier sorts, a distinguished sort $\alpha \in \mathcal{IS}$ (the local sort), which is used for indentifying instances of the local specification, a set of attribute symbols ATT and a set of event symbols EVT. Based on a signature Θ, a set of predicates $\mathcal{P}(\Theta)$ is defined, consisting of predicates $\triangleright e(x_1, \ldots, x_n)$ and $\odot e(x_1, \ldots, x_n)$ for every event symbol e and $\triangleright a(x_1, \ldots, x_n)$ for every attribute symbol a. The intended meaning of the predicates $\triangleright e(x_1, \ldots, x_n)$ and $\triangleright a(x_1, \ldots, x_n)$ is that event e is enabled (i.e., it *can* occur (it does not have to occur) in the current state), respectively, that a certain value for attribute a is observable. The predicate $\odot e(x_1, \ldots, x_n)$ indicates that event e occurs in the current state. From these predicates, a local specification language is defined, with the usual logical connectives as well as three temporal operators: $\bigcirc$ ("next"), $\Box$ ("always"), and $\diamond$ ("sometime"). In this local language, predicates are localized by prefixing them with a variable with as sort the sort α. A *local specification* is a set of formulae (the local axioms) in this language. State transitions are described by axioms that define admissible behavior by relating current and future states.

Local Semantics of OSL. Also, semantically local and global states are distinguished. Here, we first introduce the local interpretation of formulae. We assume the existence of a fixed data universe (an algebra of data types), providing for each sort s a carrier set $A(s)$. Formulae in the local specification language are interpreted over local interpretation structures that consist of a family of carrier sets for the sorts used and a sequence of states $(\sigma_k)_{k \in \mathbb{N}}$ (discrete linear time). States are sets of predicates that describe which observations are possible, which events are enabled and which events actually occur in that state. Formally, states are elements of the set of all possible local states

$$\Pi = \{p(x_1, \ldots, x_n) \mid p \in \mathcal{P}(\Theta), x_i \in A(s_i) \text{ for } i = 1, \ldots, n\}.$$

A sequence of states has additional semantic constraints, among which is the following frame assumption: Only the occurrence of an event can change the set of observables and enabled events (i.e., only an event occurrence can change a state). Another constraint is that attributes are functional in OSL: if $\triangleright a(x) \in \sigma$ and $\triangleright a(x') \in \sigma$ for some state σ and for some $a \in ATT$, then $x = x'$. As a consequence, attributes are interpreted functionally, as usual.

Template Morphism Syntax: Bridge between Local and Global Level. As stated before, objects can be composed to form complex objects. In OSL, this is specified using *template morphisms* to compose complex signatures from which the complex objects are generated. A template morphism $\sigma : \Theta \to \Theta'$ between two signatures $\Theta = (\mathcal{IS}, \alpha, ATT, EVT)$ and $\Theta' = (\mathcal{IS}', \alpha', ATT', EVT')$, is a tuple $(\sigma_{IS}, \omega_\alpha, \sigma_{ATT}, \sigma_{EVT})$, with σ_{IS} a mapping of identifier sorts, σ_{ATT} and σ_{EVT} mappings for the attribute and event symbols, respectively, and $\omega_\alpha : \alpha' \to \sigma_{IS}(\alpha)$ an operator used to distinguish kinds of morphisms. There are two kinds of morphisms: *inclusion* of Θ in Θ' and *injection* of Θ in Θ'. With the former, additional signature elements can be added to Θ, making inclusion morphisms suitable for modeling specialization. With the latter, Θ can be incorporated in a more complex signature, making it suitable to model aggregations of objects in composed objects. Using template morphisms, different local signatures can be combined. The resulting signatures are used to compose formulae in the global language.

Global Language and Semantics of OSL. The global language is based on a signature that consists of all local signatures, generated by inclusion and injection template morphisms. The language defined over this signature consists of formulae of the form $(\varphi \Rightarrow \psi)$, where φ and ψ are local-language interaction formulae over different signatures. Using this language, relations between (the behavior of) objects is described. In a formula $\varphi \Rightarrow \psi$, the occurrence of events described in φ implies the simultaneous occurrence of the events described in ψ ("event calling"). Formulae in the global language are interpreted over composed interpretation structures that are generated from the local interpretation structures.

Intuition Behind OSL. Consider a global language formula $\varphi \Rightarrow \psi$. Both φ and ψ are formulae of the global signature, containing subformulae that refer to different objects in the system. At the global level, with formulae like $\varphi \Rightarrow \psi$, it is only possible to formulate expressions like "if something happens in the life of an object referred to in φ, then something else has to happen in the life of an object referred to in ψ at the same moment." For such an expression to have any meaning, it is necessary to specify what the lives of both objects look like. This is what is done by the full temporal logic subformulae of φ and ψ.

4.4.3 Proof Calculi and Operationalization of TROLL and OSL

In [64], a sound axiom system for OSL is given. (Completeness has not been proven yet.) Experiments using an automated theorem prover for reasoning about specifications have been performed. By translating TROLL specifications to OSL, it is, in principle, possible to reason about TROLL specifications as well.

Execution of TROLL specifications is, in general, not possible. Execution of a TROLL specification would amount to finding a model that satisfies the specification, but, because TROLL is an extension of full first-order logic, no algorithm exists that finds such a model for an arbitrary TROLL specification. To provide operationalization, a restricted version of TROLL with an operational semantics has been defined. This version of the language restricts the arbitrary first-order temporal constraints available in TROLL. To enable formal proof techniques, a more restricted version, called TROLL*light* [65], has been developed. This language is much less declarative: It lacks the temporal language and instead uses an operational formalism resembling the pattern language of TROLL.

12. Actually, $\mathcal{IS}$ is the Cartesian category freely generated from the partial order. We will present OSL here without reference to the category structure of the sorts. In [66], OSL is also presented without reference to the category structure of the sorts.

```
class Parameter_class
   template
      attributes
         Name:  name_type [key];
         Value:  value_type;
      events
         set_value(V: value_type);
      effects
          [set_value(V: value_type)] Value=V;
end class Parameter_class;
```

Fig. 13. TROLL specification of the class `Parameter_class`.

4.4.4 Running Example

To specify the running example *Propose & Revise* in TROLL, the problem has to be modeled as an object-oriented system first. This can be done in (at least) two ways, depending on which parts of *Propose & Revise* are modeled as the most important objects:

Inference actions as active objects. With this approach, the inference actions are modeled by separate objects that cooperate with objects or data structures that model the stores. When using this approach for modeling *Propose & Revise*, the main (active) objects would be a Proposer and a Reviser. Both would operate on a (passive) design object.

Stores as active objects. With this approach, stores are modeled by objects that have methods corresponding to the inference actions that use them as input or output stores. The inference actions themselves are thus modeled (only) as methods. When using this approach for modeling *Propose & Revise*, the main objects are a design object, which is able to e.g., revise and evaluate itself, and an active object containing violated constraints.

In this paper, the second approach is taken because this results in a clearer specification: there are less objects, parameter passing is minimal, and the specification has a more object-oriented spirit (data and operations are grouped together). The main objects are: an (active) design object, which is able to initialize, propose, revise, and evaluate itself, and a violations object. These objects work together as components of a third object: `Parametric_design_task`, which handles I/O and controls the overall behavior.

In the rest of this section, first, three basic classes are specified. After that, a first-level decomposition of *Propose & Revise* is given, corresponding to its functional specification. This decomposition is refined into a second-level decomposition, in order to specify the control flow, as indicated in Fig. 4 in Section 3.

The Basic Classes. We begin by defining, in Fig. 13, a class for parameter objects.[13] Parameters have two attributes: a name (which serves as the key identifying parameter obejcts in a collection, indicated by [**key**]), and a value. There is one event (i.e., an operation on a parameter): `set_value`, which sets the value attribute to a certain value.

13. The class definitions in this paper are not complete: for example, birth and death events are deliberately omitted.

```
class Design_model_class
   template
      components
         Parameters:  SET(Parameter_class);
      constraints
            -- p_1 ... p_n are the only parameters
            ∧_{i=1}^{n}((exists P:|Parameter_class|)
               (P in Parameters_COMP_IDs and
               P.Name=p_i)) and
               card(Parameters_COMP_IDs)=n;
end class Design_model_class
```

Fig. 14. TROLL specification of the class `Design_model_class`.

Objects belonging to class `Parameter_class` are aggregated in a composed object of class `Design_model_class`, which is defined in Fig. 14. The scope of bound logic variables is indicated as follows in TROLL: in (**exists** `V` : $sort\_id$) (φ), variable `V` is bound in the formula φ delimited by the parentheses.

Instances of the class `Design_model_class` are composed objects, having a set of instances of class `Parameter_class` (i.e., parameter objects) as their components. This is specified by the construct

$$\textbf{components}\ name : \textbf{SET}(class\_name);.$$

In TROLL, this construct implicitly defines an attribute $name$`_COMP_IDs`, taking values of sort **set**($|\ class\_name\ |$). (Note: $|\ class\_name\ |$ denotes the identifier sort associated with class $class\_name$. It is used for referencing instances of the class.) In the first-level decomposition below, two instance of `Design_model_class` will be used, serving as input store and output store, respectively. Moreover, in the second-level decomposition, a subclass of `Design_model_class` is defined. This subclass imposes more constraints and events on a set of parameters, thus modeling a design object as a special kind of a set of parameters, having more specialized behavior. We will now define the class of constraint objects in Fig. 15.

Instances of the class `Constraint_class` are simple (i.e., noncomposed) objects having two attributes: a constraint name and a set of tuples consisting of `n` values each. As an example constraint, consider

$$(\texttt{not\_all\_fours\_or\_eights},\quad \{(4,4,\ldots,4),\ (8,8,\ldots,8)\})$$

stating that there is a constraint, called

`not_all_fours_or_eights`,

forbidding that either $p_1 = 4$ and $p_2 = 4$, etc., or that $p_1 = 8$ and $p_2 = 8$, etc.

First-level Decomposition. The first-level decomposition of the running example is presented in Fig. 16. In the specification of the running example in Fig. 16, the parametric design task is modeled as a composed object consisting of objects belonging to the class `Design_model_class` (input

```
class Constraint_class
   template
      attributes
         Name:  name_type [key];
         Values:  set(tuple(value_type,...,value_type));
         -- the elements of the set are n-tuples
end class Constraint_class
```

Fig. 15. TROLL specification of the class `Constraint_class`.

```
class Parametric_design_task
   template
      components
         Input:  Design_model_class;
         Output:  Design_model_class;
         Constraints:  SET(Constraint_class);
      events
         propose_and_revise;
      constraints
         -- Do not modify the user input by the design in output (Req. PR1):
         ∧_{i=1}^{n}((p_i in Input_ID.Parameters_COMP_IDs and Input_ID.Parameters(p_i).Value=V
            and not V=undef and occurs(propose_and_revise)) implies
               (next (Output_ID.Parameters(p_i).Value=V)));
         -- All parameters in output have a value (Req. PR2):
         occurs(propose_and_revise) implies (next
            (∧_{i=1}^{n}(not Output_ID.Parameters(p_i).Value=undef)));
         -- No constraint is violated by value assignments in output (Req. PR3):
         occurs(propose_and_revise) implies (next (not (exists C:|Constraint_class|)
            (C in Constraints_COMP_IDs and
               ((exists V_1,...,V_n:  value_type)(∧_{i=1}^{n}(Output_ID.Parameters(p_i).Value=V_i)
                  and tuple(V_1,...,V_n) in C.Values)))));
end class Parametric_design_task
```

Fig. 16. First-level decomposition of *Propose & Revise* in TROLL.

and output) and a set of constraints. Note that in TROLL definitions, free variables are implicitly universally quantified. (In our examples, variables are always denoted by single capital letters.)

Second-level Decomposition. In the first-level decomposition given above, the `parametric design` task was completely specified in terms of input and output. There was no design object representing the store "design." In the refinement of that specification, however, this will not be the case. The init, evaluate, propose, and evaluate inference actions are modeled as events that can occur in the life of a design object. In the second-level decomposition, there is also a store "Violations" (see Fig. 4), which is modeled as a separate class in TROLL, given in Fig. 17. The only attribute of "Violations" is a set of constraint identifiers.

The Design object is an instance of the class `Design_class`, presented in Fig. 18, which is a special kind of a design model (set of parameters), namely, a design model subject to the constraints imposed on the design task. To represent this relationship between `Design_class` and `Design_model_class`, the former is specified as a subclass of the latter as indicated in Fig. 18.

We are now ready to refine the specification given in Fig. 16, using the `Violations_class` and `Design_class` classes as extra components of the object. (Note that the components `Input`, `Output`, and `Constraints` are as in the specification in Fig. 16.) In the refined specification, which is given in Fig. 19, $->$ is the sequential composition of processes. All-capital identifiers are process identifiers in the pattern language.

4.4.5 Concluding Remarks

In principle, the TROLL idea of having a user-oriented, rich syntax on top of the terse syntax of OSL is nice. The structuring concepts introduced by this extra layer greatly extend the modeling power of OSL, which is shown by the examples given in [66]. However, it appeared to be very difficult to specify the running example in TROLL. The resulting specification suffers from very complex and long logic formulae. We think that, to a great extent, this is due to the following problems: In the first place, the language has a quite verbose syntax. In the second place, it is not easy to quantify over collections of objects. Such quantification has to be coded by the user, using (implicitly defined) attributes like `Input_ID.Parameters_COMP_IDs`. In the third place, there are often many ways to specify requirements, which makes

```
class Violations_class
   template
      attributes
         Current_violations:  set(|Constraint_class|);
      events
         -- The following events occur if the set of constraints is empty or not,
         -- respectively.  (However, their specification is not given.)
         violations_empty; violations_not_empty;
      constraints
         --The specifications of the events would have been given here.
end class Violations_class
```

Fig. 17. TROLL specification of the class `Violations_class`.

```
class Design_class
   specializing Design_model_class;
      template
         events
            init;      -- these are the inference actions from Fig. 3; their func-
            evaluate;  -- tional specifications are not given except for 'revise'
            propose(Input:  |Design_model_class|);
            test(Viol:  |Violations_class|, Constraints:  set(|Constraint_class|));
            revise(Viol:  |Violations_class|, Input:  |Design_model_class|);
            -- The following events occur after evaluate, indicating
            -- if design is complete:
            evaluate_complete; evaluate_partial;
         constraints
            -- Revise repairs all constraint violations (Req. R1):
            occurs(revise(Viol, Input)) implies (next (not (exists C:|Constraint_class|)
               (C in Viol.Current_violations and
                  ((exists V_1,...,V_n:  value_type)(∧_{i=1}^{n}(Parameters(p_i).Value=V_i) and
                     tuple(V_1,...,V_n) in C.Values)))));
            -- Revise does not make a propose (Req. R2):
            ∧_{i=1}^{n}((Parameters(p_i).Value=undef and occurs(revise(Viol, Input))) implies
               (next Parameters(p_i).Value=undef));
            -- Revise gives no new values for input (Req. R3):
            ∧_{i=1}^{n}(Input.Parameters(p_i).Value=V_i and not V_i=undef and
               occurs(revise(Viol, Input))) implies (next (Parameters(p_i).Value=V_i));
            -- Don't make parameters undefined (Req. R4):
            ∧_{i=1}^{n}(((not Parameters(p_i).Value=undef) and
               occurs(revise(Viol, Input))) implies
                  (next not Parameters(p_i).Value=undef));
end class Design_class;
```

Fig. 18. TROLL specification of the class `Design_class`.

it difficult to choose the best way. Another weakness is that there is no way to express the relation between a functional specification and its implementation: It is not possible to express this refinement relationship within the language.

As discussed above, TROLL provides two ways to specify control: by constraints that define admissible behavior using a temporal language and by constructive pattern language expressions. On the one hand, both mechanisms are useful in the specification of knowledge-based systems, which we consider as an important feature of a specification formalism. On the other hand, the pattern language is not very expressive. Because the pattern languages only has sequential composition, choice and recursion, specifying iteration results in abusing process variables as a kind of line numbers and targets for goto. Although composition, choice, and recursion are powerful enough to express constructs like a while-loop, the syntax of TROLL does not actually provide explicit looping constructs.

A final remark concerns proof obligations in the formalization of the running example. As claimed before, the specification in Fig. 19 implements the one in Fig. 16. By claiming this, we are given the obligation to prove that the properties for `propose_and_revise` as given in the first-level decomposition hold for the behavior specified in the

```
class Parametric_design_task
   template
      components
         Input:  Design_model_class;
         Output:  Design_model_class;
         Constraints:  SET(Constraint_class);
         Design:  Design_class;
         Violations:  Violations_class;
      events
         take_input; give_output;
      patterns
         take_input -> Design.init -> GO_ON;
         GO_ON is Design.propose(Input_ID)
            -> Design.test(Violations_ID, Constraints_COMP_IDs)
               -> select
                  Violations.violations_empty -> EVALUATE
               or  Violations.violations_not_empty ->
                              Design.revise(Violations_ID, Input_ID) -> EVALUATE
               end select
         EVALUATE is Design.evaluate
            -> select
                Design.evaluate_complete -> give_output
             or  Design.evaluate_partial -> GO_ON
             end select;
end class Parametric_design_task
```

Fig. 19. Second-level decomposition of *Propose & Revise* in TROLL.

second-level decomposition. However, TROLL has no facilities to express this proof obligation. (There is even no way to state the relationship between the specifications in Fig. 16 and Fig. 19.) A newer version of the language [68], which allows manipulation of formulae on a metalevel, claims to have addressed this problem.

4.5 Language for Conceptual Modeling

Wieringa's language LCM (Language for Conceptual Modeling) was designed as a tool for the conceptual analysis of object-oriented databases. The aim is to develop a theory of dynamic objects, and to provide a logic for specifying such objects and for reasoning about them. To this aim, equational logic is used for the specification of static algebras and for a dynamic algebra of events. A (basic version) of dynamic logic is used for the specification of the interaction of the two. We refer to Section 4.4 for a brief introduction about object-oriented concepts. Our exposition is based on the papers [69] and [18].

4.5.1 Syntax

The basic language of LCM is order-sorted equational logic. The formulae of this language are then used as the atomic formulae of a dynamic logic.

A specification in LCM has three parts, here to be called the *value block*, the *object class and relationship block*, and the *service block*. The first part contains common datatypes and some built-in operators on them (like the natural numbers with addition). In addition to this, the user may specify Abstract Data Types (ADT's) using the order-sorted equational language. The second part contains definitions of object classes, and of composite object classes (called *relationships*). The third part is supposed to contain at least one sort $EVENTS$ referring to actions that can be performed on states. Wieringa does not fix a particular set of operations for this signature, but one should have some kind of process algebra in mind, like ACP ([17]) or CCS ([67]). There is one minimum condition on the signature, namely, that the sort $EVENTS$ has a binary communication operator.

Of the *object class and relationship block*, we only discuss the object class part. It is assumed that, for each object class to be defined, the value block must define an identifier sort of the same name—this identifier sort provides all object identifiers of the objects of that class. The most important part of this block is the declaration, for each class, of attributes, predicates and events. Attributes and predicates refer to the aspects of objects that are subject to modification; typical examples are age or $address$. The events are functions with codomain $EVENTS$, and may have several argument sorts; the event applicable to the instances of a class are declared in its *events section*.

A *specification* in LCM consists of the three blocks described above, together with a number of axioms. For the common datatypes in the value block (natural numbers, sets, strings, etc.), one has some standard specifications in mind, and the same applies to the third block (the events)—we already mentioned the examples ACP and CCS. Concerning the object specifications, each object class contains a list of transaction decompositions; here, the communication operator of the sort $EVENTS$ can be used

to indicate which local events (pertaining to one object only) may be composed to one global event.

Finally, to specify the system's behavior, one may use axioms written in a basic version of dynamic logic. The attributes of the class objects can be subject to *static integrity constraints* to be expressed in the form of (conditional) statements, such as equations or inequalities (the machinery of dynamic logic is not yet used here). A typical example is "$age(p) \leq 150$." Second, *effect axioms* are of the form $\phi \rightarrow [e]\psi$, where ϕ and ψ are finite conjunctions of equations, and e is a term of sort $EVENTS$. A typical example here is the formula

$$(children(p) = cc) \rightarrow [addchild(p, c)](children(p) = cc \cup \{c\}).$$

The third and last type of axiom is that of a *precondition axiom*. Such an axiom must be of the form $\langle e \rangle true \rightarrow \psi$, where α and ψ are as in the previous case. The meaning of this axiom is that if we are in a state where there is a possible execution of e that terminates, then currently, ψ is true.

4.5.2 Semantics

The semantics of a specification consists of three parts (following the three parts of a specification). First of all, the meaning of the value block is given by an algebra, which should be an *initial model* of the value block specification (in an initial model, all the elements of the domain are denoted by some term, and two elements are the same exactly when their corresponding terms are provably equal). Based on this initial model, possible worlds are formed. These possible worlds have the initial model as their domain, and differ only in the interpretation of the attributes and predicates. The worlds should be models of the object class block specification.

Besides this static part, a model for the specification also contains an algebra $\mathcal{E}$ for the event part of the specification. The only constraint on this algebra is that it is a model for the event specification; thus, for instance, it should contain a universe of events interpreting the terms of sort $EVENTS$. Finally, a model contains a function ρ that relates the static and the event part by mapping terms of the sort $EVENTS$ to binary relations (or functions) on the set of possible worlds—if $(w, w') \in \rho(e)$, then the intuitive meaning is that the event e causes a transition from world w to world w'. The map ρ should satisfy some natural conditions concerning the $\mathcal{E}$-algebra; for instance, terms that denote identical elements in the $\mathcal{E}$-algebra should denote identical relations as well.

It is hard to describe the nature of the transitions in the semantics in general, since the event signature is not fixed beforehand, and the effect of events on a state has to be described by explicit axioms. This flexibility also has the effect that the power to construct new events from old ones is determined by the expressiveness of the event signature. The flexibility may (and should) of course be limited for reasons of efficiency and/or mathematical transparency.

Intended models for some special kind of specifications are described in which the effect of each event pertains to one object only, and, thus, has a minimal effect on the state as a whole. This kind of restricted semantics can also be described by explicitly listing some frame axioms.

4.5.3 Proof Calculus and Control

Wieringa defines a proof calculus for which he claims soundness and completeness with respect to the semantics described above. However, in order to handle the intended models in which the effects of the events is minimal, one needs to write down frame axioms explicitly.

Basically, Wieringa's approach to control is pair-based and not constructive; that is to say, the effect and precondition axioms only refer to what is going on now and after the execution of one event. They provide no more than a limited picture of the state as it is after the execution of one $EVENT$-term. One needs explicit frame axioms to gain sufficient control of the new state.

On the other hand, by ingeniously combining the precondition axioms with the static constraints, more is possible then it seems at first sight. Also, since the signature of the dynamic algebra is not fixed beforehand, the formalism offers a lot of flexibility, perhaps allowing some opportunities to specify global constraints in a very rich dynamic language.

4.5.4 Running Example

To specify the running example in LCM, which is developed for modeling object-oriented databases, first the running example had to be modeled as an object-oriented system, as was the case with TROLL. Again, it was chosen to model the stores as active objects. The reader is referred to Section 4.4.4 for a discussion of this choice. Our example in LCM is organized as follows: We first define the basic datatypes needed to represent parameter-values, and parameter-violations (the value types $BOOL$, $VALUE_i$, and $VIOLATION$). We then introduce an object class $PARAM\_SPACE$ to represent a set of parameters. This class is partitioned in two subclasses, a subclass $INPUT\_OUTPUT$ to represent the input and output design and a more complex subclass $CURRENT\_DESIGN$ to represent the current design.

All this leads to two alternative ways of specifying the *Propose & Revise* method: one as a specification in which only the pre and postconditions on input and output are specified, and one which "opens this up" and specifies the substeps of this method as the life-cycle of the current-design object.

The basic datatypes that are needed to model the running example can be modeled in a straightforward way. First of all, we need a datatype for Boolean values, and a datatype for the value types of each parameter. Each such value type must include at least the special value $undef$:

begin value type $BOOL$
  functions
    $true, false : BOOL$
end value type

begin value type $VALUE_i$
  functions

begin object class $PARAM\_SPACE$
  attributes
    $p_1 : VALUE_1$
    ...
    $p_n : VALUE_n$
  events
    set_undef
  axioms
    $\forall P : PARAM\_SPACE[set\_undef(P)] \bigwedge_{i=1}^{n} P.p_i = undef_i$
  partitioned by
    $INPUT\_OUTPUT, CURRENT\_DESIGN$
end object class

begin object class $INPUT\_OUTPUT$
end object class

begin value type $INPUT\_OUTPUT$
  functions
    $input, output : INPUT\_OUTPUT$
end value type

Fig. 20. LCM specification of object class $PARAM\_SPACE$.

begin object class $P\&R$
  axioms
    % Do not modify the user input by the design in output (Req. PR1):
    $\bigwedge_{i=1}^{n}(input.p_i = v_i \rightarrow [P\&R]output.p_i = v_i)$
    % All parameters in output have a value (Req. PR2):
    $\bigwedge_{i=1}^{n}[P\&R]\neg(output.p_i = undef_i)$
    % No constraint is violated by value assignments in output (Req. PR3):
    $[P\&R]constr\_viol(output.p_1, ..., output.p_n) = \emptyset$
end object class

Fig. 21. LCM specification of *Propose & Revise*.

$undef_i : VALUE_i$
end value type

Next, we declare the $VIOLATION$-type, together with two elementary functions on sets of violations. The third function $constr\_viol$ determines for a tuple of values (i.e., a full design) which set of constraints has been violated (possibly the empty set if no constraints are violated).

begin value type $VIOLATION$
  functions
    $\emptyset$: **set of** $VIOLATION$
    $elem\_of : VIOLATION \times$
      set of $VIOLATION \rightarrow BOOL$
    $constr\_viol : VALUE_1 \times \ldots \times VALUE_n \rightarrow$
      set of $VIOLATION$
end value type

We now introduce, in Fig. 20, the basic notion of a parameter-space, which is a collection of parameter values. This class has an event which enables us to initialize all parameter values as undefined. We partition this class in two subclasses, one to represent the input and output designs, and one to represent the current design, on which we will define the design process.

We are now in a position to give, in Fig. 21, the specification of the *Propose & Revise* method solely in terms of the input-output conditions. The object-class given in Fig. 21 does nothing more than specifying the overall properties that we want to enforce on the problem solving method, and correspond directly to Requirements PR1, PR2, and PR3.

An alternative way of specifying *Propose & Revise* is to describe the steps that together make up the computation of this method. This specification is done by specifying the life-cycle of the class $CURRENT\_DESIGN$, of which the object $current$ is the only instance. The specification is presented in Fig. 22.

The attributes $p_1, \ldots, p_n$ represent the values for the n different parameters. The predicate $violating$ determines if a design violates any constraints. All the inference actions are specified as events that can happen to a design, and these events are characterized by postconditions axioms. The overall control flow over these events is specified in the life-cycle of the design object-class. It is unclear to us whether (and, if so, how) the relation between the specification of the class $P\&R$ and the detailed specification in the class $CURRENT\_DESIGN$ can be specified in LCM.

```
begin value type CURRENT_DESIGN
    functions
        current : CURRENT_DESIGN
end value type

begin object class DESIGN
    predicates
        violating
    events
        test
        evaluate
        revise
        propose
        init
        copy
    axioms
        % the new value no longer causes a violation (Req. R1):
        [revise]constr_viol(current.p_1, ..., current.p_n) = ∅
        % don't revise undefined parameters (Req. R2):
        ∧_{i=1}^{n}(current.p_i = undef_i → [revise]current.p_i = undef_i)
        % don't revise parameters defined in the input (Req. R3):
        ∧_{i=1}^{n}(input.p_i = v_i ∧ v_i ≠ undef_i → [revise]current.p_i = v_i)
        % and don't make parameters undefined (Req. R4):
        ∧_{i=1}^{n}(current.p_i ≠ undef_i → [revise]current.p_i ≠ undef_i)
    lifecycle
        ∀d : DESIGN : DESIGN(d) =
        INPUT_OUTPUT.set_undef(output);
        set_undef(d);
        init(d);
         repeat   propose(d);
                  violating(d) := false;
                  test(d);
                  if violating(d) then revise(d) endif
         until    evaluate
        copy(d);
end object class
```

Fig. 22. Alternative LCM specification of *Propose & Revise*.

4.5.5 Discussion

It is important to notice that in all of the above we did not use any of the features of LCM which were used to motivate the language. We are only using the process algebra in a trivial way to string operations together in sequences and loops, and we are not using it to model operations as a decomposition of communicating processes. We are also using only very few of the object-oriented features of the languages. All the real work happens in one class, with one big object which brings all the elements of the specification together. From this, we may conclude that our example is not aimed very well at displaying these distinguishing features of LCM.

The nice thing about Wieringa's approach is its (conceptual) neatness; in particular, the syntactic separation of the static and the dynamic part seems quite elegant. The fact that the dynamic signature is not fixed beforehand makes the framework very flexible. The price that one has to pay for this is the frame axioms that are needed to fully describe the effect of events. Finally, the communication algebra that Wieringa uses in the paper we studied provides a neat way of localizing the dynamics of an object to that object.

5 Comparison and Conclusions

In this section, we briefly compare the different formalisms using our two dimensions of analysis and then discuss a number of implications for the specification of (in particular, control of) knowledge-based systems.

5.1 A Short Comparison

We will give a brief overview of the frameworks in terms of the concepts and aspects of specification mentioned in Section 2.

5.1.1 States

With the exception of PDDL, where a state is a propositional valuation, a state is either an algebra (ASM and LCM)

TABLE 2
Overview of State Description Syntax and Semantics

	Syntax	Semantics
(P)DDL	PDDL: propositional formulae DDL: first-order predicate formulae	PDDL: (set of) propositional valuations DDL: (set of) first-order structures
$\mathcal{TR}$	First-order predicate formulae	First-order structure
ASM	Equational formulae	Algebra
TROLL	Sorted first-order predicate formulae (used for attribute declaration part of object templates)	First-order structure (Templates are translated into OSL sorted first-order formulae that denote first-order structures)
LCM	Sorted equational formulae (used for Value type and object class declarations)	Algebra

or a first-order structure (DDL, $\mathcal{TR}$, and TROLL/OSL). Syntactically, algebras are described in equational logic, while first-order structures are described in first-order predicate logic. In TROLL and LCM, the language is sorted, in the other frameworks it is unsorted. In PDDL, a state is described in propositional logic. DDL and PDDL have an operational semantics in which a state is a *set* of first-order structures (DDL) or a *set* of propositional valuations (PDDL). One last point is whether the interpretation of function symbols is fixed over all states, or whether it may vary. In ASM and LCM (in which there are only functions), functions are of course allowed to vary over states. In LCM, only the attribute functions and Boolean functions (which play the role of predicates) are allowed to vary; functions specified in the data value block (addition on the integers, for instance) must be the same in all states. In DDL, there are no function symbols, only constants, which should be the same in all states. In both TROLL and $\mathcal{TR}$ functions are not allowed to vary (although varying functions can be simulated by varying functional relations). Table 2 summarizes syntax and semantics of the specification of states.

5.1.2 Elementary Transitions

With respect to the specification of elementary transitions, two approaches can be distinguished: user-defined and predefined, fixed elementary transitions. In TROLL and LCM, the user defines a set of elementary transitions (i.e., specifies their names) and describes their effects using effect and precondition axioms. For instance, in TROLL, the user defines for each object class a set of events, which are the elementary transitions from one point in time of a TROLL model to the next. Associated with each event e is a predicate $\mathbf{occurs}(e)$, which is true in a time point t iff event e occurs in time point t, leading to a new state at time point $t+1$. Using this predicate, the user describes the intended behavior of e. In LCM, the user also defines a set of events for each object class. For each event e, the user can define effect axioms of the form $\phi \rightarrow [e]\psi$ and precondition axioms of the form $\langle e\rangle true \rightarrow \psi$. The events denote binary relations over states. On the other hand, in (P)DDL and ASM, there is only a predefined, fixed set of elementary transitions, which resemble the assignment statement in programming languages. In (P)DDL, there are two parameterized predefined elementary transitions, and there is no possibility for the user to define additional ones. These predefined transitions are $\mathcal{I}^H p$ (set p to true) and $\mathcal{D}^H p$ (set p to false), which update the database state according to a logic program H after setting p to true or false, respectively, and their variants $\mathcal{I}p$ and $\mathcal{D}p$, which just insert p into or delete p from a database state. Semantically, $\mathcal{I}p$ and $\mathcal{D}p$ are relations that link pairs of states (m, n), where $m = n$ for all predicates but p. In ASM, there is only one type of elementary transition, namely, function updates expressed as $f(t) := s$, which links two algebras A and A' that only differ in the values for $f(t)$. Like DDL, there are parallel updates and choice. The $\mathcal{TR}$ approach is in-between these two approaches: as in TROLL and LCM, the user defines a set of elementary transitions, but unlike in TROLL and LCM, it is possible to constructively define their effect in a transition oracle. Semantically, in $\mathcal{TR}$, an elementary transition is a relation between database states, where the transition oracle defines which pairs of database states are related. In $\mathcal{TR}$, it is also possible to describe the effect of an elementary transition without explicitly defining that transition in the transition oracle. Table 3 summarizes syntax and semantics of the specification of elementary transitions.

5.1.3 Composed Transitions

In ASM, there are several possibilities to specify composed transitions, such as adding guards to transition rules, specifying bulk transitions that fire a number of transitions at the same time, and specifying choice. However, there is no possibility to explicitly specify sequential composition or iteration. For the other frameworks, two approaches can be distinguished. In TROLL and $\mathcal{TR}$, elementary transitions can be composed using sequencing, iteration, and choice, using the syntax of the pattern language in TROLL, and $\otimes$ and $\rightarrow$ in $\mathcal{TR}$. In both frameworks, the composed transitions thus formed are interpreted over sequences of states. In LCM, elementary transitions can be composed using a syntax derived from process algebra, which also amounts to having sequencing, iteration, and choice for composition. In (P)DDL, this can be done using sequencing, iteration, bulk updates. and choice. However, unlike in TROLL and $\mathcal{TR}$, a composed transition is not interpreted over a sequence of states, but as a relation between pairs of states: the state at the beginning of the composed transition and the final state of the composed transition, as in

TABLE 3
Overview of Syntax and Semantics of Elementary Transitions

	Syntax	Semantics
(P)DDL	Fixed: database updates $\mathcal{I}^H p$ and $\mathcal{D}^H p$ (active) and $\mathcal{I}p$ and $\mathcal{D}p$ (passive)	Relation between states (m, n) where m and n differ at p
$\mathcal{TR}$	User-defined: set of names, *possibly* with effects described in transition oracle and program	Relation between states
ASM	Fixed: function updates $f(t) := s$	Relation between states (algebras) (m, n) where m and n only differ at $f(t)$
TROLL	User-defined: user specifies set of event names and effects using effect and precondition axioms	Axioms are translated to OSL, where they denote relations between states at time point t and $t + 1$
LCM	User-defined: user specifies set of event names and effects using effect and precondition axioms	Relation between states (algebras)

TABLE 4
Overview of Syntax and Semantics of Composed Transitions

	Syntax	Semantics
(P)DDL	Constructive: sequence (;), iteration (*) and test (?) as in Dynamic Logic Constraining: not possible	Relation between begin state and end state
$\mathcal{TR}$	Constructive and constraining: first-order formulae with special operator for sequence ($\otimes$)	Formulae are interpreted over sequences of states
ASM	Constructive: Transitions can be guarded, bulk updates and choice between transitions is expressible. Sequencing or iteration is not expressible Constraining: not possible	As for elementary transitions: guarding, choice and bulk updates are not concerned with sequences of states or with begin/end states
TROLL	Constructive: pattern language expressions with operators from process algebra Constraining: temporal language expressions	Pattern language and temporal language translated to OSL and interpreted over temporal sequences of states
LCM	Constructive: expressions in user-defined process algebra Constraining: not possible	Relation between begin state and end state

Dynamic Logic. The transition relation associated with a composed transition is of the same kind as the transition relation associated with an elementary transition in LCM and (P)DDL, and no intermediate states are accessible in the semantics, so, it is impossible to express constraints on intermediate states.

There is another important difference between TROLL and $\mathcal{TR}$ on the one hand, and LCM and (P)DDL on the other hand. In (P)DDL and LCM, specifying control in composed transitions in a constructive way ("programming" with sequencing, choice, and iteration) is the only possibility. However, in TROLL and $\mathcal{TR}$, control can also be specified by constraining the set of possible runs of a system, e.g., in TROLL, control over runs of the system can also be specified by expressing constraints using temporal logic. Table 4 summarizes syntax and semantics of the specification of composed transitions.

5.1.4 Proof Systems and Operationalization

The five approaches use various proof systems. Both PDDL and DDL are equipped with a Hilbert-style proof system. In both cases, the proof system is sound and complete for a subclass of all structures that form the semantic domain of PDDL. For $\mathcal{TR}$, there is a Gentzen-style proof system for the Horn version of $\mathcal{TR}$. A more general proof system is announced in [46]. The ASM approach deliberately does not provide any specific proof system. Instead, the ASM approach is kept as simple as possible to allow the use of standard mathematical techniques for proving properties of specifications. Several authors have experimented with using various proof tools for ASMs. For TROLL, there is no direct proof system. However, as the semantics of TROLL is defined in terms of OSL, for which a proof system exist, it is possible to prove properties of a TROLL specification via a

TABLE 5
Overview of Proof Systems and Operationalization

	Proof system	Operationalization
(P)DDL	Hilbert-style proof system	Operational semantics in terms of transition rules
$\mathcal{TR}$	Gentzen-style proof system for the 'serial-Horn' fragment of $\mathcal{TR}$	Operational semantics based on 'executional deduction', serial-Horn fragment only
ASM	No fixed proof system. General mathematical techniques are applicable	An interpreter for ASMs is straightforward, and several already exist
TROLL	TROLL is translated to OSL; there is a Hilbert-style proof system for OSL	Provided for a restricted version of TROLL lacking full quantification, and for TROLL*light*
LCM	Equational logic	

translation to OSL. LCM uses a proof system based on equational reasoning.

Most approaches support operationalization. For PDDL, an operational semantics in terms of state transitions is provided. This operational semantics is proven to be equivalent with the declarative semantics. For $\mathcal{TR}$, the operational semantics is restricted to the so-called serial Horn version of $\mathcal{TR}$. This semantics is based on a restricted form of deduction which restricts some freedom in proofs. Operationalization of ASM specifications is straightforward, and indeed a number of ASM interpreters have been created. For TROLL, there is an operational semantics for a version of the language in which some restrictions are imposed on first-order temporal formulae. A more restricted operational version, TROLL*light*, has been created to support automatic proof techniques. Table 5 summarizes proof systems and operationalization of the five approaches.

5.2 Conclusions

In this second part of the concluding section, we will make a number of observations that are relevant for future users of the specification languages discussed above, and for future designers of KBS specification languages, in particular, as far as the choice of specification language features for control is concerned.

5.2.1 Constructive or Constraining Specifications

In all of the languages discussed in this paper, the constructive style of specification is supported. Examples of this are the program expressions in DDL, or the communicating algebra expressions in LCM. In contrast with the widely supported constructive style of specification, only TROLL and $\mathcal{TR}$ support the constraining style of specification. (ASM allows constraints on domains and on external functions to be expressed in general mathematical terms instead of in the ASM language itself.) We think that for the specification of control of the reasoning process of a KBS, both styles are valuable. It would be especially useful to be able to combine both styles in one specification, as is possible in $\mathcal{TR}$ and TROLL.

5.2.2 Global or Local Control

The languages differ in the extent to which control must be specified globally, for an entire system, or locally, separately for individual modules of a system. In particular, DDL and $\mathcal{TR}$ only allow a single, global control specification, while TROLL and LCM allow the specification of control that is local for individual modules. Because the arguments in favor of either approach resemble very much the arguments in favor or against object-oriented programming, we will not go into any detail here, but refer to that discussion, with the proviso that we are concerned here with notions of modularity and encapsulation, and not so much with inheritance and message passing. Besides such general software engineering arguments in favor of object-oriented techniques, knowledge modeling has particular use for such techniques: Frames have a long tradition in knowledge representation and are a precursor of object-oriented techniques. Dealing with mutually inconsistent subsets of knowledge is a particular example of the use of localized specifications.

5.2.3 Control Vocabulary

With "control vocabulary," we mean the possibilities (in a technical sense) that the language gives us to construct composed transitions from more primitive ones. Here, the news seems to be that there is relatively little news: there is a standard repertoire of dynamic type constructors that every language designer has been choosing from. This repertoire usually contains sequential compositions and often one or more from the following: iteration, choice, or parallelism (with or without communication).

Two languages take a rather different approach however, namely, LCM and ASM. The designers of LCM suggest the use of some form of process algebra for their dynamic signature, but make no strong commitment to any particular choice, and LCM should perhaps be viewed as parameterized over this choice. In the case of ASM, it seems that there is no possibility at all to include any control vocabulary in the language: ASM provides only its elementary transitions (the algebra updates). It provides neither a fixed vocabulary for building composed transitions, nor does it seem parameterized over any choice for such a vocabulary. In practice, auxiliary mechanisms such as macros (textual substitutions) are used to augment the control vocabulary.

A final point concerns the treatment of nonterminating processes. Such nonterminating processes might occur in the specification of knowledge-based systems for process

control and monitoring. TROLL, LCM, and ASM can all deal with such nonterminating processes. Although it is of course possible to specify nonterminating processes in (P)DDL and $\mathcal{TR}$, it is not possible to derive any useful properties of such programs because in (P)DDL and $\mathcal{TR}$, nonterminating processes have trivial (empty) semantics.

5.2.4 Refinement

It is commonly accepted in software engineering that a desirable feature of any specification language is to have the possibility of refinement. By this, we mean the ability to specify program components in terms of their external properties (i.e., a functional specification, sometimes called a "black box" specification), and only later unfold this black box specification into more detailed components, and so on recursively.

In the context of specification languages, a necessary condition for the possibility of refining is the presence of names for actions: one needs to be able to name a transition which is atomic on the current level (i.e., a "black box" specification), but which is perhaps a complex of transitions on a finer level. Without such names for actions, one cannot give an abstract characterization of transitions. Of course, such an abstract characterization (in terms of preconditions, postconditions, etc.) should be possible in the framework to allow refinement later on.

It is not immediately clear how the languages discussed above behave in this respect. DDL clearly does not allow refinement (names referring to composed actions simply do not exist in DDL), while LCM does (at least, if we choose the signature of the process algebra sort rich enough). The external functions of ASM give us the means to make black box specifications. However, it is not possible *within* the ASM framework to specify the behavior of such black boxes, which by implication also precludes the possibility of proving within the ASM framework that a given implementation (refinement) of a black box satisfies the specifications. The designers of the ASM framework prefer to use general mathematical techniques for treating refinement. The simple mathematical structure of the ASM framework makes this feasible.

Although the transaction base from $\mathcal{TR}$ resembles the external functions of ASM, $\mathcal{TR}$ is stronger than ASM in this respect: The transaction base can be used to model black-box transitions, but, unlike the external functions in ASM, the transitions of $\mathcal{TR}$ can be specified by means of pre and postconditions within $\mathcal{TR}$ itself. Furthermore, it is possible to later provide an implementation of a transaction in $\mathcal{TR}$, and to prove that this implementation is indeed a correct refinement of the functional specification.

In TROLL, it seems that there is *almost* the possibility to say that one specification refines the other. TROLL enables both constraining specification (based on atomic transition), but also constructive specification of composed transitions (in terms of more detailed atomic transitions). What is lacking is syntactic support to relate such a constructive specification to an atomic transition, so it cannot be expressed that this more detailed specification is a refinement of the atomic transition. Semantical considerations of the relationship between transactions and their refinement are investigated in detail in [70].

Finally, desirable as the presence of names for composed actions may be, there is a price to be paid for having the option of black box specifications. A black box specification of a transition usually only states which things change, with the assumption that all other things remain the same. It should not be necessary for the user to explicitly specify what is left unaffected by the transition. The problem of how to avoid statements of what remains the same (the frame axioms) has proven to be very difficult. This so-called frame problem is the price that has to be paid.

In languages with only predefined transactions (like in DDL), the designers of the language have specified the required frame-axioms. For languages with user-defined atomic transactions, there is no way out for the user but to write down the frame axioms explicitly (although they can sometimes be generated automatically). For the purposes of execution, the frame problem can be circumvented by an implementation of the primitive transactions outside the logic. However, the languages we are dealing with are meant to *specify* systems, and the price for such externally implemented primitive transactions has to be paid at verification time. For verification purposes, we would want the primitive transactions to be specified in the logic, which then brings back the frame problem.

5.2.5 Proofs

Since the languages discussed in this paper are intended as tools to formally specify software systems, we would expect them to be equipped with proof systems which enables us to prove that a specification exhibits certain properties. Of the languages discussed, only $\mathcal{TR}$ and (P)DDL pay extensive attention to a proof system. TROLL has to rely on its translation to OSL in order to use the proof system of OSL, while ASM relies on general mathematical reasoning, without a formal proof system. LCM has a proof system based on equational logic.

5.2.6 Syntactic Variety

There is a large variety in the amount of syntactic distinctions which are made by the various languages. On the one hand, languages like TROLL and LCM provide a rich variety of syntactic distinctions, presumably to improve ease of use by human users, while on the other hand, approaches like (P)DDL, ASM, and $\mathcal{TR}$ provide a much more terse and uniform syntax. This issue is related with the different goals which the different proposals are aiming at. Syntactically rich languages like TROLL and LCM aim at being a full blown specification language, while formalisms like $\mathcal{TR}$ and (P)DDL aim in the first place at formalizing the notion of database updates, rather than being a specification language themselves. ASM was originally designed as a foundational framework for computation, although it is used as a specification language as well.

5.2.7 States as Histories

In three of the languages discussed in this paper (ASM, LCM, and (P)DDL), a composed transition is interpreted as an ordered pair of states (begin state and end state). However, for the types of properties that we might want to verify of our systems using the logics discussed in this paper, this interpretation of composed transitions is not

sufficient. For many purposes, an interpretation as a sequence of (intermediate) states is required. For example, many safety critical applications require proofs of properties such as "action α is never done," or "α is never done twice" or "α is never done twice in a row," or "action α is never followed by action β." To prove such properties, we must consider sequences of intermediate states (as in $\mathcal{TR}$ and TROLL), and not just an ordered pair of begin- and end-state of a program (as in (P)DDL).

Using such sequences of intermediate states (also called: histories) as the basis for a formalization of program behavior is indeed more general than merely considering the begin state and end state of a program. In particular, states can be defined as equivalence classes of histories (e.g., an end state corresponds with the set of all histories that terminate in this state). In this way, abstraction mechanisms can be defined that distinguish more or less details among histories, as desired. For example, in (P)DDL, any two histories that have the same begin state and end state are equivalent. Other possibilities are to regard two histories as equivalent when they are composed of the same sets of states, but perhaps in a different order, or only to regard them as equivalent when they are identical sequences of states (this is the option taken in $\mathcal{TR}$). For example, assume that we are interested in the values that a particular variable v takes during the course of a computation (as is often the case in safety-critical applications):

- If we are only interested in the final value of v, then all histories can be identified with their final state;
- If we are interested in all intermediate values of v, but not in their sequence, then histories can be treated as sets of states;
- If we are interested in the sequence of values for v, then histories must be treated as sequences of states.

It is an open issue whether the grain size of such distinctions between different histories should be a fixed aspect of the logic (with (P)DDL and $\mathcal{TR}$ representing opposite choices in this respect), or whether such a grain size should be definable in the language of the logic, for instance, by expressing equality axioms among histories.

5.2.8 Transitions as Semantical Concepts

In most languages, transitions are available in the language (e.g., a procedure in (P)DDL corresponds to a transition, as does an event in LCM), but, semantically, they are derivates of states. In such languages, a transition is an ordered pair of states, and no semantically separate category exists for transitions per se. Furthermore, transitions do not occur in the languages as first-class objects over which we can express predicates.

In the words of Gabbay [71, Ch. 4]:"The modeling given so far may eventually prove not radical enough. After all, if logical dynamics is of equal importance to logical statics, then the two ought to be accorded equal ontological status. This means that transitions would come to be viewed, not as ordered pairs of states, but rather as independent basic objects in their own right." Again, it remains an open and interesting question how approaches in which transitions are first-class objects relate to the approaches discussed in this paper, in particular, with respect to the representation of histories and equivalence of histories.

5.3 Final Remarks

The original motivation of the research reported in this paper was the lack of consensus among KBS specification frameworks concerning the specification of control for KBSs. We had hoped that neighboring areas might have solved this problem, or at least have established more stable notions than what had been achieved in the KBS area.

Our investigations among non-KBS specification languages have revealed a number of constructions that could certainly be of interest for the KBS specification language community. Examples of these are the notions of constructive and constraining control specification (and, in particular, the idea to combine both of these in a single language), the idea to define transitions in terms of sequences of intermediate states instead of just the initial and terminal state of the transition, and the rich variety of semantic characterizations of the notion of state. Furthermore, these constructions are not just initial ideas, but have often reached a state of formal and conceptual maturity which make them ready to be used by other fields such as the specification of KBSs.

However, this wide variety of well worked out proposals, is at the same time a sign of much unfinished work. As in the field of KBS specification languages, the neighboring fields have not yet reached any sort of consensus on the specification of control, neither in the form of a single ideal approach, nor in the form of guidelines on when to use which type of specification.

ACKNOWLEDGMENTS

The authors are grateful to E. Börger, M. Kifer, G. Saake, and R. Wieringa for their comments on an earlier version of this paper. The anonymous reviewers of this paper provided valuable suggestions for improvements.

REFERENCES

[1] F.M.T. Brazier, B.M. Dunin-Keplicz, N.R. Jennings, and J. Treur, "DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework," *Int'l J. Cooperative Information Systems,* special issue on Formal Methods in Cooperative Information Systems: Multiagent Systems, M. Huhns and M. Singh, eds., vol. 6, no. 1, pp. 67–94, 1997.

[2] F.M.T. Brazier, J. Treur, N.J.E. Wijngaards, and M. Willems, "Temporal Semantics of Compositional Task Models and Problem Solving Methods," *Data and Knowledge Eng.,* vol. 29, no. 1, pp. 17–42, 1999.

[3] D. Fensel, *The Knowledge Acquisition and Representation Language KARL.* Boston: Kluwer Academic, 1995.

[4] D. Fensel, J. Angele, and R. Studer, "The Knowledge Acquisition and Representation Language KARL," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 4, pp. 527–550, July/Aug. 1998.

[5] L. in 't Veld, W. Jonker, and J.W. Spee, "The Specification of Complex Reasoning Tasks in $K_{BS}SF$," *Formal Specification of Complex Reasoning Systems,* J. Treur and T. Wetter, eds., pp. 233–255, 1993.

[6] J.W. Spee and L. in 't Veld, "The Semantics of $K_{BS}SF$: A Language for KBS Design," *Knowledge Acquisition,* vol. 6, 1994.

[7] F. Harmelen and J. Balder, "$(ML)^2$: A Formal Language for KADS Conceptual Models," *Knowledge Acquisition,* vol. 4, no. 1, pp. 127–161, 1992.

[8] D. Fensel and R. Groenboom, "MLPM: Defing a Semantics and Axiomatization for Specifying the Reasoning Process of Knowledge-Based Systems," *Proc. 12th European Conf. Artificial Intelligence (ECAI-96),* pp. 1123–1127, Aug. 1996.

[9] C. Pierret-Golbreich and X. Talon, "TFL: An Algebraic Language to Specify the Dynamic Behaviour of Knowledge-Based Systems," *The Knowledge Eng. Rev.,* vol. 11, no. 3, pp. 253–280, 1996.

[10] *Formal Specification of Complex Reasoning Systems,* J. Treur and T. Wetter, eds. New York: Ellis Horwood, 1993.

[11] D. Fensel and F. Harmelen, "A Comparison of Languages Which Operationalize and Formalize KADS Models of Expertise," *The Knowledge Eng. Rev.,* vol. 9, no. 2, pp. 105–146, 1994.

[12] D. Fensel, "Formal Specification Languages in Knowledge and Software Engineering," *The Knowledge Eng. Rev.,* vol. 10, no. 4, pp. 361–404, 1995.

[13] D. Fensel and R. Straatman, "The Essence of Problem Solving Methods: Making Assumptions for Efficiency Reasons," *Advances in Knowledge Acquisition,* N. Shadbolt, K. O'Hara, and G. Schreiber, eds., pp. 17–32, 1996.

[14] B. Nebel, "Artificial Intelligence: A Computational Perspective," *Principals of Knowledge Representation,* G. Brewka, ed., studies in Logic, Language, and Information, pp. 237–266, 1996.

[15] D. Harel, "Dynamic Logic," *Handbook of Philosophical Logic, Vol. II: Extensions of Classical Logic,* D. Gabbay and F. Guenthner, eds., pp. 497–604, 1984.

[16] D. Kozen and J. Tiuryn, "Logics of Programs," *Handbook of Theoretical Computer Science,* J. Leeuwen, ed., vol. B, ch. 14, pp. 789–840, 1990.

[17] J.A. Bergstra and J.W. Klop, "Algebra of Communicating Processes with Abstraction," *Theoretical Computer Science,* vol. 37, pp. 77–121, 1985.

[18] R.J. Wieringa, "LCM and MCM: Specification of a Control System using Dynamic Logic and Process Algebra," *Formal Development of Reactive Systems: Case Study Production Cell,* C. Lewerentz and T. Lindner, eds., pp. 333–355, 1995.

[19] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas, "Troll—A Language for Object-Oriented Specification of Information Systems," *ACM Trans. Information Systems,* vol. 14, pp. 175–211, Apr. 1996.

[20] A. Bonner and M. Kifer, "Transaction Logic Programming," *Proc. 10th Int'l Conf. Logic Programming (ICLP),* pp. 257–279, 1993.

[21] P. Spruit, R. Wieringa, and J.-J. Meyer, "Axiomatization, Declarative Semantics and Operational Semantics of Passive and Active Updates in Logic Databases," *J. Logic and Computation,* vol. 5, no. 1, pp. 27–50, 1995.

[22] P. Spruit, R. Wieringa, and J.-J. Meyer, "Dynamic Database Logic: The First-Order Case," *Proc. Fourth Int'l Workshop Foundations of Models and Languages for Data and Objects,* U. Lipeck and B. Thalheim, eds., pp. 102–120, 1993.

[23] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," *Specification and Validation Methods,* E. Börger, ed., 1994.

[24] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications,* vol. 1. Berlin: Springer-Verlag, 1985.

[25] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications,* vol. 2. Berlin: Springer-Verlag, 1990.

[26] M. Wirsing, "Algebraic specification," *Handbook of Theoretical Computer Science,* J. Leeuwen, ed., vol. B, ch. 13, pp. 676–778, 1990.

[27] *Algebraic System Specification and Development.* M. Bidoit, H.J. Kreowski, P. Lescane, F. Orejas, and D. Sannella, eds., 1991.

[28] J.M. Spivey, *The Z Notation. A Reference Manual,* second ed. New York: Prentice Hall, 1992.

[29] J.B. Wordsworth, *Software Development with Z.* Addison-Wesley, 1992.

[30] C.B. Jones, *Systematic Software Development Using VDM,* second ed. Prentice Hall, 1990.

[31] D. Andrews and D. Ince, *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[32] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems,* vol. 14, no. 1, pp. 25–59, 1987.

[33] S. Marcus and J. McDermott, "SALT: A Knowledge Acquisition Language for Propose and Revise Systems," *Artificial Intelligence,* vol. 39, no. 1, pp. 1–37, 1988.

[34] A.T. Schreiber and B. Birmingham, "Special Issue on the VT Sisyphus Task" *Int'l J. Human-Computer Studies (IJHCS),* 1996.

[35] R. Milner, *Communication and Concurrency.* New York: Prentice Hall Int'l, 1989.

[36] F. Kroeger, *Temporal Logic of Programs.* Berlin: Springer-Verlag, 1987.

[37] B.J. Wielinga, A.T. Schreiber, and J.A. Breuker, "KADS: A Modeling Approach to Knowledge Engineering," *Knowledge Acquisition,* vol. 4, no. 1, pp. 5–53, 1992.

[38] A.T. Schreiber, B.J. Wielinga, H. Akkermans, W.V.D. Velde, and R. de Hoog, "CommonKADS. A Comprehensive Methodology for KBS Development," *IEEE Expert,* vol. 9, no. 6, pp. 28–37, 1994.

[39] *Second Generation Expert Systems,* J.-M. David, J.-P. Krivine, and R .Simmons, eds. Berlin: Springer-Verlag, 1993.

[40] "Special Issue on the Sisyphus 91/92 Models," *Int'l J. Man-Machine Studies,* M. Linster, ed., vol. 40, no. 2, 1994.

[41] D. Fensel and R. Straatman, "The Essense of Problem-Solving-Methods: Making Assumptions for Gaining Efficiency," *Int'l J. Human Computer Studies,* vol. 48, no. 2, pp. 181–215, 1998.

[42] F.M.T. Brazier, P. Langen, J. Treur, N.J.E. Wijngaards, and M. Willems, "Modelling an Elevator Design Task in DESIRE: The VT Example," *Int'l J. Human-Computer Studies,* special issue on Sisyphus-VT, A.Th. Schreiber and W.P. Birmingham, eds., vol. 44, nos. 3-4, pp. 469–520, 1996.

[43] K. Poeck, D. Fensel, D. Landes, and J. Angele, "Combining KARL and CRLM for Designing Vertical Transportation Systems," *Int'l J. Human-Computer Studies,* special issue on Sisyphus-VT, A.Th. Schreiber and W.P. Birmingham, eds., vol. 44, nos. 3-4, pp. 435–467, 1996.

[44] A.J. Bonner and M. Kifer, "An Overview of Transaction Logic," *Theoretical Computer Science,* vol. 133, no. 2, pp. 205–265, 1994.

[45] A.J. Bonner and M. Kifer, "A Logic for Programming Database Transactions," *Logics for Databases and Information Systems,* J. Chomicki and G. Saake, eds., pp. 117–166, 1998.

[46] A. Bonner and M. Kifer, "Transaction Logic Programming (or, A Logic of Procedural and Declarative Knowledge)," Technical Report CSRI-323, Computer Systems Research Inst., Univ. of Toronto, Nov. 1995.

[47] R. Groenboom and G.R.R. de Lavalette, "Reasoning about Dynamic Features in Specification Languages," *Semantics of Specification Languages (SoSL): Proc. Int'l Workshop Semantics of Specification Languages,* D.J. Andrews, J.F. Groote, and C.A. Middelburg, eds., 1994.

[48] P. Spruit, R.J. Wieringa, and J.-J.Ch. Meyer, "Regular Database Update Logics," *Theoretical Computer Science,* in press.

[49] M. Aben, *Formal Methods in Knowledge Engineering.* PhD thesis, Univ. of Amsterdam, Faculty of Psychology, ISBN 90-5470-028-9, Feb. 1995.

[50] F. van Harmelen, "Applying Rule-Base Anomalies to KADS Inference Structures," *Decision Support Systems,* vol. 21, no. 4, pp. 271–280, 1998.

[51] E. Börger and J.K. Huggins, "Abstract State Machines 1988-1998: Commented ASM Bibliography," *EATCS Bull.,* Formal Specification Column, H. Ehrig, ed., pp. 105–127, Feb. 1998.

[52] "JUCS Special ASM Issue," *J. Universal Computer Science,* E. Börger, ed., vol. 3, nos. 4-5, 1997.

[53] J. Meseguer, "Conditional Rewriting Logic as a Unified Model of Concurrency," *Theoretical Computer Science,* vol. 96, no. 1, pp. 73–155, 1992.

[54] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of Maude," *Electronic Notes in Theoretical Computer Science,* vol. 4, 1997.

[55] G. Denker, "From Rewrite Theories to Temporal Logic Theories," *Electronic Notes in Theoretical Computer Science,* vol. 15, 1998.

[56] G. Schellhorn and W. Ahrendt, "Reasoning About Abstract State Machines: The WAM Case Study," *J. Universal Computer Science,* vol. 3, no. 4, pp. 377–413, 1997.

[57] W. Zimmerman and T. Gaul, "On the Construction of Correct Compiler Back-Ends: An ASM Approach," *J. Universal Computer Science,* vol. 3, no. 5, pp. 504–567, 1997.

[58] K. Winter, "Model Checking for Abstract State Machines," *J. Universal Computer Science,* vol. 3, no. 5, pp. 689–701, 1997.

[59] G. Castillo, I. Durdanovic, and U. Glässer, "An Evolving Algebra Abstract Machine," *Computer Science Logic,* selected papers from CSL '95, H.K. Büning, ed., pp. 191–214, 1996.

[60] B. Beckert and J. Posegga, "leanEA: A Lean Evolving Algebra Compiler," *Computer Science Logic,* selected papers from CSL '95, H. Kleine Büning, ed., pp. 64–85, 1996.

[61] A.M. Kappel, "Executable Specifications Based on Dynamic Algebras," *Proc. Fourth Int'l Conf. Logic Programming and Automated Reasoning (LPAR-93),* pp. 229–240, 1993.

[62] E. Börger, "Why Use Evolving Algebras for Hardware and Software Engineering?," *Theory and Practice of Informatics: Proc. Software Seminar (SOFSEM '95),* M. Bartosek, J. Staudek, and J. Wiedermann, eds., pp. 236–271, 1995.
[63] A.Z. Diller, *An Introduction to Formal Methods.* John Wiley & Sons, 1992.
[64] A. Sernadas, C. Sernadas, and J. Costa, "Object Specification Logic," *J. Logic and Computation,* vol. 5, pp. 603–630, Oct. 1995.
[65] S. Conrad, M. Gogolla, and R. Herzig, "TROLLlight: A Core Language for Specifying Objects," Informatik-Berichte 92-02, Technische Universität Braunschweig, 1992.
[66] R. Jungclaus, *Modeling of Dynamic Object Systems—A Logic-Based Approach,* Advanced Studies in Computer Science. Vieweg Verlag, 1993.
[67] R. Milner, "A Calculus of Communicating Systems," *Lecture Notes in Computer Science,* vol. 92. Springer-Verlag, 1980.
[68] S. Conrad, J. Ramos, G. Saake, and C. Sernadas, "Evolving Logical Specification in Information Systems," *Logics for Databases and Information Systems,* J. Chomicki and G. Saake, eds., pp. 199–228, 1998.
[69] R.J. Wieringa, "A Formalization of Objects using Equational Dynamic Logic," *Proc. Second Int'l Conf. Deductive and Object-Oriented Databases (DOOD '91),* C. Delobel, M. Kifer, and Y. Masunaga, eds., pp. 431–452, 1991.
[70] G. Denker, J. Ramos, C. Caleiro, and A. Sernadas, "A Linear Temporal Logic Approach to Objects with Transactions," *Proc. Algebraic Methodology and Software Technology: Sixth Int'l Conf., AMAST '97,* M. Johnson, ed., pp. 170–184, 1997.
[71] D. Gabbay, "What is a Logical System?" *Studies in Logic and Computation,* vol. 4, 1994.

Pascal van Eck received his diploma in computer science (MSc degree) from the Free University of Amsterdam in 1995. Since 1995, he has worked as a research assistant in the Artificial Intelligence Department at the Free University and has been writing a PhD thesis on the development of a formal, compositional semantic structure for the dynamics of multiagent systems. In 2000, he started as an assistant professor in the Information Systems Group at the University of Twente, The Netherlands. His research interests include requirements analysis and design of multiagent systems and their application in e-business, and automated negotiation between software agents.

Joeri Engelfriet studied computer science and mathematics at the Free University of Amsterdam. After receiving MSc degrees in 1993 and 1995, respectively, he started working on a PhD thesis on formal models for static and dynamic aspects of complex reasoning processes, and was awarded the PhD degree from the Free University in 1999. From 1998 to 1999, he worked as an associate professor in the AI group at the Free University, where his research interests shifted toward knowledge-based systems, agent technology, and electronic commerce. He is author of more than 25 research papers. Currently, he is employed at McKinsey & Company as management consultant.

Dieter Fensel studied mathematics, sociology and computer science in Berlin. In 1989, he joined the Institute AIFB at the University of Karlsruhe. His major subject was knowledge engineering and his PhD thesis, in 1993, was about a formal specification language for knowledge-based systems. From 1994 until 1996, he visited the group of Bob Wielinga at the SWI Department in Amsterdam. During this time, his main interests were problem-solving methods for knowledge-based systems. In 1996, he came back as a senior researcher at the Institute AIFB working finalizing his Habilitation in 1998. In 1999, he started as an associate professor at the Free University of Amsterdam. Currently, his focus is on the use of ontologies to mediate access to heterogeneous knowledge sources and to apply them in electronic commerce.

Frank van Harmelen studied mathematics and computer science in Amsterdam. In 1989, he was awarded the PhD degree from the Department of AI in Edinburgh for his research on metalevel reasoning. While in Edinburgh, he worked with Dr. Peter Jackson on Socrates, a logic-based toolkit for expert systems, and with Professor Alan Bundy on proof planning for inductive theorem proving. After his PhD research, he moved back to Amsterdam where he worked from 1990 to 1995 in the SWI Department under Professor Wielinga. He was involved in the REFLECT project on the use of reflection in expert systems, and in the KADS project, where he contributed to the development of the $(ML)^2$ language for formally specifying Knowledge-Based Systems. In 1995, he joined the AI Department at the Vrije Universiteit Amsterdam, where he holds a senior lectureship. His current interests include formal specification languages for knowledge-based systems, verification and validation of knowledge-based systems, and developing approximate notions of correctness for knowledge-based systems. He is author of a book on metalevel inference, editor of a book on knowledge-based systems, and has published more than 60 research papers.

Yde Venema received the PhD degree in 1992 from the University of Amsterdam under the supervision of Johan van Benthem. His main research interests are in various branches of modal logic, such as temporal and dynamic logic. He is the coauthor of a monograph on multidimensional modal logic and is currently preparing a text book on modal logic. Dr. Venema is working at the University of Amsterdam as a research fellow of the Royal Netherlands Academy of Arts and Sciences.

Mark Willems received the PhD degree in 1993 from the Faculty of Applied Mathematics at the University of Twente, The Netherlands, on semantic representation of natural language. From 1993 to 1995, he was assistant professor (UD) at the Vrije Universiteit Amsterdam, The Netherlands. From 1996 to 1997, he worked at Bolesian, the leading knowledge technology and management company in the Benelux. From 1997 to 1998, he worked at Cycorp, Austin, Texas, a research company specializing in ontologies and representation of common-sense knowledge. After returning to the Netherlands, he worked for Bolesian again. In 2000, he joined Quintiq, a provider of advanced planning and scheduling software, as vice president of professional services.