Specification of Dynamics for Knowledge-Based Systems

Pascal van Eck[1], Joeri Engelfriet[1], Dieter Fensel[2], Frank van Harmelen[1], Yde Venema[3], and Mark Willems[4]

[1] Vrije Universiteit Amsterdam, Faculty of Mathematics and Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Tel: +31 20 4447730, fax: +31 20 4447653. Email: {patveck,joeri,frankh}@cs.vu.nl
[2] Institut AIFB, University of Karlsruhe, D-76128 Karlsruhe, Germany. Email: dfe@aifb.uni-karlsruhe.de
[3] Institute for Logic, Language and Computation, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands. Email: yde@wins.uva.nl
[4] Bolesian B.V., Steenovenweg 19, 5708 HN Helmond, The Netherlands. Email: mark@bolesian.nl

Abstract. During the last years, a number of formal specification languages for knowledge-based systems have been developed. Characteristic 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 means to specify a complex and large amount of knowledge and they have to provide means to specify the dynamic reasoning behaviour of a knowledge-based system. This paper will focus on the second aspect, which is an issue considered to be unsolved. For this purpose, we have surveyed existing approaches in related areas of research. We have taken approaches for the specification of information systems (i.e., Language for Conceptual Modelling and TROLL), approaches for the specification of database updates and the dynamics of logic programs (Transaction Logic and Dynamic Database Logic), and the approach of Abstract State Machines.

1 Introduction

Over the last years a number of formal specification languages have been developed for describing *knowledge-based systems* (KBSs). Examples are DESIRE [19]; KARL [8]; $\mathrm{K_{BS}SF}$ [27]; $(\mathrm{ML})^2$ [15]; MLPM [9] and TFL [24]. 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 [31, 11, 7].[1]

[1] See also ftp://swi.psy.uva.nl/pub/keml/keml.html at the World Wide Web.

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. [23]). 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 *non-functional* 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 behaviour 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 afore-mentioned languages use a number of formalisms to describe the dynamics of a KBS: DESIRE uses a meta-logic to specify control of inferences of the object logic, $(ML)^2$ and MLPM apply dynamic logic ([14]), KARL integrates ideas of logic programming with dynamic logic, and TFL uses process algebra in the style of [1]. 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, 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 Modelling (LCM, [32]) and TROLL ([17]) as examples from the information systems area. Both languages provide means to express the dynamics of complex systems.
– Transaction Logic ([3]), (Propositional) Dynamic Database Logic (PDDL, [30] and DDL [29]) as examples for database update languages provide means to express dynamic changes of databases.
– Abstract State Machines ([13]) 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. [33]), which provide means for a functional specification of a system, and model-based approaches like Z [28] and the Vienna Development Method - Standard Language (VDM-SL) [16], 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. 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. We were also not so much looking for full-fledged specification approaches but we were searching for extensions of logical languages adapted for the purpose of specifying dynamics. 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 [7]. Finally, one may miss specification approaches like LOTOS [2] that are designed for the specification of interactive, distributed and concurrent systems with real-time aspects. Because 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.

The paper is organized as follows. First, in Section 2 we introduce two dimensions we distinguish to structure our analysis. In Section 3, we introduce the different approaches we have studied. A comparison based on an example worked out in all approaches has been carried out. Section 4 sketches this example and presents the most important issues of the formalization of the example in all approaches. The interested reader is referred to the long version [6] for a detailed presentation of the example and the formalization of it in all approaches. Section 5 provides a short comparison between the formalisms according to our dimensions of analysis, and conclusions.

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.

The behaviour 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 behaviour of a KBS. We will look at the possibility of specifying *how* the reasoning process achieves its results. This is called the *internal specification*. The description of *what* the reasoning process has to derive is called the *external specification*. One must be able to relate the internal specification of a reasoning process with the goal that should be achieved by it. This introduces two requirements: modelling primitives are required that describe the desired functionality of a KBS (i.e., its external specification) and a proof system must be provided that enables to relate the internal and external descriptions 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). Which modelling primitives does the language offer to describe a state, elementary transitions, etc? Second, we examine the semantics of the language. A formal semantics serves two purposes: it enables the definition of a precise meaning of language expressions and it enables proofs of statements over language expressions. These proofs can be formalized and semi-automatic proof support can be provided if a proof system based on a formal semantics has been developed. Therefore, in the third place we look at such proof systems and operationalization. Operationalization of the logic is required for prototyping, which is based on operational semantics. Prototyping or partial evaluation could provide restricted but still very useful support for the validation of specifications.

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

As mentioned in the previous subsection, we distinguish two styles for the specification of composed transitions: external and internal. The former specifies a system as a black box in terms of its externally visible behaviour. 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 description styles appear in

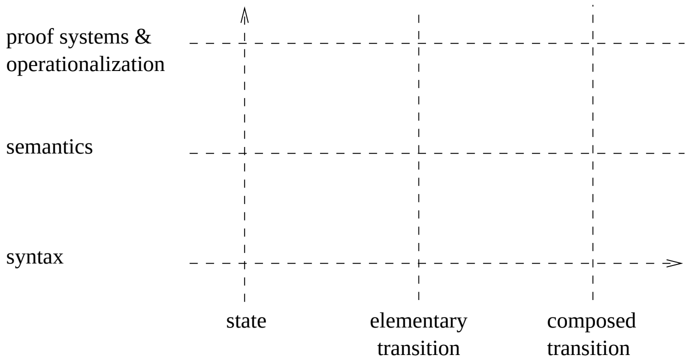

Fig. 1. The two dimensions of our analysis

specification languages for KBSs: external descriptions may appear at the lowest and at the highest level of specification of a KBS, while internal specifications relate the description at the lowest and highest levels.

The elementary inferences of a KBS as well as its overall functionality should be describable in an external style, as the internal details of an elementary inference are regarded as implementational aspects. (A specification should not enforce any commitments to its algorithmic realization.) The overall functionality of a KBS, that is, the goals it can reach, should be describable independent from the way they are achieved. Actually, the equivalence of the functional specification of the goals (or task) and the specification of the reasoning process of the KBS 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 should 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 following we discuss these different concepts of a specification in more detail.

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 local states) and (3) how an individual state is represented;

Not each specification approach in software or knowledge engineering provides the explicit notion of a state (either global or local). An alternative point of view would be an *event-based* philosophy useful to specify parallel processes (compare [22]). TFL uses processes as elementary modelling primitives that are further characterized by abstract data types in the style of process algebra. 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).

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 ([18])), 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).

Elementary Transitions. Elementary transitions should be describable without enforcing any commitments to their algorithmic realization. A pure 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 modelling decision that its internal details should not be represented. An important criterion for specification approaches for KBSs is therefore the granularity of the elementary transitions they provide.

Composed Transitions. One can distinguish non-constructive and constructive manners to specify control over state transitions. A *non-constructive* 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 non-determinism which again leads to several possibilities for the actual control.

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); (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.

Syntax. Each of the three concepts of a specification is represented by a part of the syntax of a specification framework. A spectrum of flavours 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, possibly using one or two grammar rules.

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. To support rigid proofs of specification properties, such a semantics should be formal. The semantics should be intuitive and relatively easy to understand so users are able to precisely comprehend what a specification means.

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. The 'operationalization' of a specification framework is meant to refer to the possibilities and techniques for such automatic prototype generation.

3 Languages

In this section, we will give a very brief description of all of the frameworks we have studied. The reader interested in more detail can either consult the original works, or read the longer version of this paper [6]. In the longer version, we describe an example of a knowledge-based system which has a non-trivial control of reasoning. This example was taken from the Sisyphus project ([20]), which was an extensive comparative exercise in the KBS community. This example has been (partly) specified in all frameworks, in order to make a realistic comparison between the languages. A specification of the top-level of the system is given, together with a refined version of one of the parts of the system (to test the possibility of external and internal specifications). The results in this paper are partly based on our experience with the example, and again, the interested reader should consult the longer version. We will now list and describe the frameworks studied.

Dynamic Database Logic ((P)DDL)

PDDL is a propositional logic for describing state and state change in deductive databases. It is based on Dynamic Logic, with both passive and active updates. In a passive update $\mathcal{I}p$ or $\mathcal{D}p$, a single proposition p is inserted into or deleted from the database. In active updates $\mathcal{I}^H p$ or $\mathcal{D}^H p$, after insertion or deletion of p, the database is closed under the rules of the (definite) logic program H, leading to further insert or delete operations on the database. The propositions in a database are divided into base-predicates and derived predicates. The base-predicates can be directly inserted into or deleted from the database. The value of the derived predicates follows from the rules of the given logic program. These predicates cannot occur as arguments to update operations. Complex transitions can be formed from these elementary transitions using the Dynamic Logic operators for sequence, choice, iteration and test.

The first-order variant (DDL) allows conditional (bulk) insertion and deletion. Conditional insertion is written $\&\{x_1, \ldots x_n\}\mathcal{I}p(t_1, \ldots t_n)\ \mathit{where}\ \phi$, which means that $p(t_1, \ldots, t_n)$ is set to true for all values of $x_1, \ldots x_n$ that make ϕ true, and similarly for conditional deletion. Complex transitions are again formed from primitive transitions by sequence, test, iteration and choice, plus an additional operator called conditional choice: $+(x_1, \ldots, x_n)\alpha\ where\ \phi$ executes α for one of the possible value assignments to $(x_1, \ldots, x_n)$ which makes ϕ true.

The semantics are like those of Dynamic Logic (Kripke models with relations for the programs), with special interpretations for the operators (the $\mathcal{I}$ operator should cause insertion for example). A proof system and an operational semantics is provided. The proof systems of DDL and PDDL are only complete for full Kripke structures (i.e., structures that contain a world for all possible valuations).

Transaction Logic ($\mathcal{TR}$)

$\mathcal{TR}$, like (P)DDL, is also a logic of state and state change in databases. In contrast to DDL, the atomic actions are a parameter of the logic: they are to be described in a transition oracle which sanctions the transition from a state to another for each elementary transition. The only dynamic operator is sequence ($\otimes$). A formula like $\phi \otimes \psi$ intuitively means that first ϕ must hold, and after that ψ must hold. Other dynamic operators are defined in terms of this operator. An example of such an operator, which will be used in the example, is $\Rightarrow$. The formula $\phi \Rightarrow \psi$ means that whenever ϕ is true, ψ must be true immediately thereafter. Formally $\phi \Rightarrow \psi \equiv \neg(\phi \otimes \neg\psi)$.

Semantically, formulae are interpreted over *sequences* of database states, called *paths* (in contrast to DDL, where the meaning of a program is a binary relation on states). Atomic statements representing updates and database facts are evaluated by the transition oracle, respectively the data oracle. The usual first-order connectives and quantifiers have their standard interpretation. A formula $\phi \otimes \psi$ is true on a path if that path can be split into two paths such that ϕ is true on the first, and ψ is true on the second. The behaviour of the database is described by formulae which constrain the allowed sequences of states. Together, these formulae are called a *program*. Often, they are in the form of Prolog-like rules. Entailment allows the deduction of properties of the program, given an initial state. These properties can describe the final state of the path (the state the database is in after execution of the program in the initial state), but are not limited to this: one can express properties of entire paths (starting at the initial state).

A proof system for a Horn-like fragment of the language is provided. By placing further restrictions on the proofs in this system, a form of *executional entailment* is obtained. Proving a query corresponds to the execution of the program.

Abstract State Machines (ASM)

The Abstract State Machine (ASM, previously called Evolving Algebras) approach originally was an attempt to provide operational semantics to programs and programming languages by improving on Turing's Thesis (Turing Machines are *too* low-level). With ASMs, it is possible to specify algorithms at any level of abstraction, and use successive refinement to investigate properties of the algorithm (like correctness) at any of these levels.

The basic concept of ASMs is simple: an ASM specification consists of rules for updating algebras. An algebra consists of a set together with functions on that set. The rules are (basically) of the form `if` ϕ `then` R, where ϕ is a condition on algebras, and R is a set of updates of the form `f(t) := s`. The intuitive meaning is that if the *current* algebra satisfies ϕ, then it can be updated to a new algebra, where the value of the function `f` in the argument `t` is `s` (and the other updates in R have been performed as well). A run of an ASM is a sequence of algebras generated by repeatedly firing all the rules in the current algebra.

There are many extensions of this simple form of rules, including bulk updates and indeterministic choice. However, there are no further constructs for determining the flow of control (dynamics), like loops or temporal operators. It is possible to use (nullary) functions as control variables. External functions are functions in the algebra which can not be updated by the ASM (but can be 'read') but which can change during a run. These functions are used to model for example the input to an ASM.

The ASM approach does not come equipped with a (fixed) proof system. Properties of evolving algebras 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. Operationalization of ASMs is relatively straightforward. Basically, one just needs a mechanism that continually fires the applicable rules.

Troll/OSL

TROLL is a language for the specification of object-oriented information systems. It provides a very rich syntax, aimed at a user-friendly way of specification. The basic structuring mechanism in TROLL is the template, which is a generic description of possible objects in terms of attributes and events of these objects. A template is not the same as a class. A class is regarded as a collection of objects described by the same template together with an identification mechanism for instances. For the specification of attributes and events, four basic languages are defined: a data language, consisting of terms in a sorted first-order logic, a state formulae language, which uses the terms from the data language, a temporal language for describing temporal constraints over state formulae and a pattern language for ordering events using process algebra operators. The state formulae language contains, for each event term e from the data language, a predicate **occurs**(e), which is true in a state where e is about to happen.

A template thus defines an object's local signature (consisting of the attributes) and local life cycle (admissible behaviour in terms of commitments, constraints, obligations and event effects). A TROLL specification consists of a number of such definitions of local object aspects together with a number of relationship definitions at a global level, such as specialization and interactions.

The semantics of TROLL are obtained via a translation into Object Specification Logic (OSL, [26]), a temporal logic for reasoning about objects. Also in OSL, there is a local logic for reasoning about local object aspects and there is a global logic (incorporating the local logic) for reasoning about object interactions. OSL is equipped with a proof system, which enables reasoning about TROLL specifications by first translating them into OSL. An execution mechanism is provided for a fragment of TROLL (lacking the temporal language), called TROLL*light*.

Language for Conceptual Modeling (LCM)

LCM is developed 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. The basic language of LCM is equational logic (for specifying abstract data types).

The signature of the equational language has separate parts for value types, classes and events. The event-signature must contain at least one sort *EVENTS* referring to actions that can be performed on states. The particular set of operations for this signature is not fixed, but one should have some kind of process algebra in mind, like ACP [1] or CCS [21]. There is one minimum condition on the event-signature, namely that the sort *EVENTS* has a binary communication operator. This communication operator can be used to indicate which local events (pertaining to one object only) may be composed to one global event.

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) equations (the machinery of dynamic logic is not yet used here). 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*. The third and last type of axiom is that of a *precondition axiom*. Such an axiom must be of the form $\langle e \rangle \mathit{true} \rightarrow \psi$, where e 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.

LCM has a proof system. In order to capture the intended models in which the effects of the events is minimal, one needs to write down frame axioms explicitly.

4 The Running Example

In a longer version of this paper [6], we have applied the various languages to a single example, thus facilitating comparison with respect to specification of the dynamics of knowledge-based systems. In particular, we have used 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. A full description of this example 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). A transition takes information from a store, reasons with it, and outputs the result to another store. We use *tasks* to represent complex transitions. A procedural language will be used for defining control over the execution of transitions.

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 central task is to find values for parameters, fulfilling certain constraints on the values of the parameters. The user is allowed to give some parameters already a value from the start. An informal functional specification of this task, which is called Parametric Design and which serves as the running example in this paper, is given by the following three requirements: (i) the initial values given by the user may not be modified; (ii) the final design must be complete and (iii) in the final design, no constraint is 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 [10]. 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 and Revise. The central idea behind Propose and 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 constraint is already violated. If not, then a value for another 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. According to the Propose and Revise method, the task Parametric Design can be decomposed into six subtasks (see Fig. 2):

- `Init`: this task initializes the design, based on parameter values that are possibly given by the user in the `Input` store.
- `Propose`: this task proposes a value for a parameter that has not been assigned a value before. It updates accordingly the current design in the store `Design`.

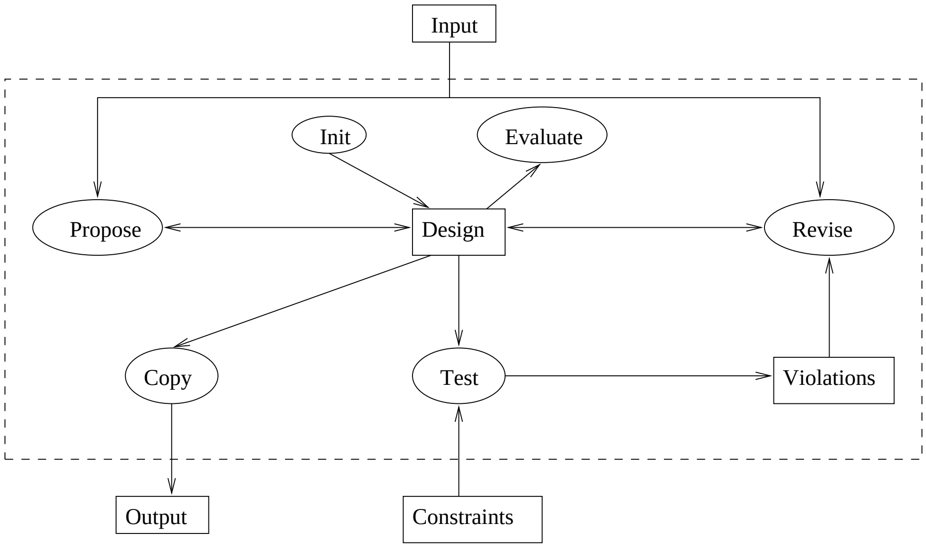

Fig. 2. Knowledge flow diagram of Propose and Revise. Boxes with round corners represent tasks, rectangles represent information stores

- `Test`: this task checks whether the current design (in the store `Design`) violates any constraints (from the input store `Constraints`) and outputs any such violated constraints to the store `Violations`.
- `Revise`: this task corrects the partial design in `Design` if the previous task stored any violated constraints in `Violations`. `Revise` is not allowed to alter any parameter values that were specified by the user in the `Input` store.
- `Evaluate`: this task checks if the current design in the store `Design` is complete.
- `Copy`: this task copies the design from `Design` to the `Output` store.

It remains to define the control between the subtasks of `parametric design`. One possibility is as follows:

Init;
repeat
    Propose;
    Test;
    if ($Violations \neq \emptyset$) **then** Revise **endif**
until Evaluate;
Copy

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 and Revise method has been studied extensively. The interested reader is directed to [4] and [25] for complete formal models of parametric design using DESIRE and KARL, respectively. In the rest of this section, some interesting ideas used in the formalizations of the running example in the long version ([6]) are presented. In particular, for each approach a formalization of the functional specification (if possible) and of the problem solving method will be given. Some conventions and principles are used in more than one formalization. Therefore, the presentations of the formalizations are grouped. Moreover, the conventions used in the $\mathcal{TR}$ formalization of Propose and Revise will also be adopted in the other formalizations, whenever appropriate.

4.1 $\mathcal{TR}$ and (P)DDL

The basic idea of the specifications using $\mathcal{TR}$ and (P)DDL (actually, DDL is used) is to treat all input and output roles of an inference action as predicates. Thus, for every store, we will define a predicate:

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

where P is a parameter name, V and $V_1, ..., V_n$ are values and C is a constraint name. The meaning of $\textit{constraint}(C, V_1, \ldots, V_n)$ 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.

To simplify the presentation of the running example in $\mathcal{TR}$ and DDL, we treat these logics as if they were typed. This can be simply encoded by introducing unary predicates for all the types in the standard fashion. We will leave these types implicit in the variable names. Thus, a formula like

$$\forall P \exists V : \textit{output}(P, V)$$

should be read as

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

in order to include all the required type-restrictions.

The Parametric Design task will be specified in $\mathcal{TR}$ by a complex transaction which we will call *parametric_design*. First we will give the functional requirements for this task, which will be specified as transaction formulas in the program. Please note that apart form *parametric_design*, all predicates are meant to be state predicates (predicates that are only true on paths of length one), which means that in all formulas we write, we should use e.g. $\textit{output}(P, V) \wedge \textbf{state}$, where **state** is a special predicate that is only true on paths of length one. To avoid cluttering up the presentation with occurrences of **state**, we omit them, but the reader should insert them for all static predicates.

The Parametric Design task is specified as follows. First of all, the input may not be modified by *parametric_design* (Requirement (i)):

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

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.

Requirement (ii) 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 requirement (iii), stating that after parametric design no constraint is violated:

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

In $\mathcal{TR}$, the problem solving method Propose and Revise, which implements the specification given above, can be given in two ways: within $\mathcal{TR}$, as treated below, 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. The transition oracle will implement this correspondence by stating:

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

for all pairs **D, D'** that 'do' *parametric_design*. We could then prove that this implementation satisfies the functional 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.

An implementation of Propose and Revise within $\mathcal{TR}$ amounts to 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 following transaction program.

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

The predicates that are used in the program above are all defined either in other rules of the program or by the oracles.

In the formalization of Parametric Design using DDL, a slightly different representation is used for the stores Input and Output. Like in the $\mathcal{TR}$ formalization, the stores are represented by predicates.

Instead of specifying tasks and inference actions as transactions as in $\mathcal{TR}$, they are formalized in DDL as *update programs* which make assignments to their output roles. For the task Parametric Design, both a functional specification $parametric\_design(V_1, ..., V_n)$ and an implementation $P\&R$ (an update program) are given, which we shall discuss now.

The functional specification of the task Parametric Design is expressed in DDL as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{parametric\_design(V_1, ..., V_n)} \\
\leftrightarrow & \\
\bigwedge_{i=1}^{n}(\ output(p_i, V_i) \wedge & \texttt{\% } V_1, ..., V_n \texttt{ are output parameters} \\
\quad \neg\exists V' : (output(p_i, V') \wedge V_i \neq V') \wedge & \texttt{\% their values are unique,} \\
\quad V_i \neq undef \wedge & \texttt{\% unequal to undef (Req. (ii)),} \\
\quad (input(p_i, V_i) \vee & \texttt{\% and not overriding the} \\
\quad\ \ input(p_i, undef))) & \texttt{\% user input (Req. (i))} \\
\ \wedge \neg\exists Viol : & \texttt{\% no constraint may be} \\
\quad\ \ constraints(Viol, V_1, ..., V_n)) & \texttt{\% violated (Req. (iii))}
\end{array}
$$

The implementation of Propose and Revise is modelled by the update program $P\&R$, which is defined as follows (Note that although constructs like 'repeat...until' and 'if...then...endif', used in the specification below, are formally not part of the syntax of DDL, they can easily be defined):

$$
\begin{array}{lll}
P\&R \equiv & \multicolumn{2}{l}{\texttt{\% Clear the roles for output and intermediate designs, and}} \\
& \multicolumn{2}{l}{\texttt{\% initialise the input:}} \\
& \multicolumn{2}{l}{init;} \\
& \textbf{repeat} & propose; \\
& & \texttt{\% empty the violations before recomputing them:} \\
& & \&\{V\}\ \mathcal{D}\ violations(V); \\
& & test \\
& & \textbf{if} \quad \exists V : violations(V) \\
& & \textbf{then}\ revise \\
& & \textbf{endif} \\
& \textbf{until} & evaluate; \\
& \multicolumn{2}{l}{\texttt{\% and finish by copying the results to the output role:}} \\
& \multicolumn{2}{l}{\&\{P, V\}\ \mathcal{I}\ output(P, V)\ \textbf{where}\ design(P, V)}
\end{array}
$$

DDL then allows to express the resulting proof obligation that the implementation $P\&R$ satisfies the functional specification $parametric\_design(V_1, ..., V_n)$. If $\mathcal{V}$ is the conjunction of all given input values $input(p_i, V_i)$, and $\mathcal{C}$ is the conjunction of all given constraints $constraint(C, V_1, ..., V_n)$, then the following expresses the correctness of the implementation with respect to the functional specification:

$$\mathcal{V} \wedge \mathcal{C} \rightarrow [P\&R]parametric\_design(p_1, ..., p_n, V_1, ..., V_n)$$

A rather unpleasant feature of the formalization of Propose and Revise in (P)DDL is the need for explicit emptying and copying of data-stores. One would perhaps expect to be able to hide such procedural details from a specification.

4.2 ASM

The ASM approach does not allow the specification of the functional requirements (i), (ii) and (iii) within an ASM specification. Therefore, in this subsection, only the implementation of Propose and Revise expressed in the ASM formalism is given. 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.) To structure the presentation of this specification, the following convention is used: expressions like `<Initialise>` denote a set of rules which specifies the behaviour of the ASM when it is initializing. In the final specification, corresponding rules should be inserted here.

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 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 and 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 `Initialise`, `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. The ASM specification of Propose and Revise is as follows:

```
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>
```

Of course, sets of rules have to be provided for `<Initialise>`, `<Propose>`, etc.

4.3 TROLL and LCM

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

Inference actions as active objects With this approach, the inference actions are modelled by separate objects that co-operate with objects or data structures that model the stores. When using this approach for modelling Propose and 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 modelled by objects that have methods corresponding to the inference actions that use them as input or output stores. The inference actions themselves are thus modelled (only) as methods. When using this approach for modelling Propose and 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 both in the formalization of the running example using TROLL and using LCM, 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 initialise, propose, revise and evaluate itself, and a violations object. These objects work together as components of a third object: `Parametric_design_task`. This object does not correspond to a store. However, it is necessary to have this object as a representation of the overall system.

In the formalization of Propose and Revise using TROLL, first some classes are defined to represent the input, output and constraints stores. Objects of these classes are then used as components of the composed `Parametric_design_task` object defined by the following class definition. The problem solving method Propose and Revise is modelled as an event in the life of a `Parametric_design_task` object itself, subject to the constraints specified in the class definition.

```
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 user input by design in output (Req. (i)):
      ∧_{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. (ii)):
      occurs(propose_and_revise) implies (next
        (∧_{i=1}^{n}(not Output_ID.Parameters(p_i).Value=undef)));
      -- No constraint is violated by values
      -- in output (Req. (iii)):
      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
```

As is the case with the formalizations of Propose and Revise using $\mathcal{TR}$ and (P)DDL, we seek to further specify the behaviour of Propose and Revise by giving a constructive description of its dynamics. In TROLL, this is done by defining a new class that represents the store 'design' (this class is a subclass of the class of which the objects `Input` and `Output` are instances). The inference actions Init, Evaluate, Propose, Test and Revise are modelled as events that happen in the life of a Design object. A *new* specification of class `Parametric_design_task` is then given, with a Design object as one of its components. The most important part of this new class definition is the following constructive specification of the behaviour of a `Parametric_design_task` object:

```
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
```

The structure of the LCM specification of Propose and Revise resembles the structure of the TROLL specification to a great extent. Again, classes are defined to represent the store Input, Output and Constraints. As an example, consider the definition of class $PARAM\_SPACE$, which is a collection of parameter values. This class has an event which enables us to initialise all parameter values as undefined. We partition this class in two subclasses, one to represent the input and output stores, and one to represent the current design, on which we will define the design process.

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

We are now in a position to give specification of the Propose and Revise method solely in terms of the input-output conditions. The following object class does nothing more than specifying the overall properties that we want to enforce on the problem solving method, and corresponds directly to requirements (i), (ii) and (iii). This class has the same function as the class `Parametric_design_class` in the TROLL specification.

begin object class $P\&R$

  axioms

    $\bigwedge_{i=1}^{n}(input.p_i = v_i \rightarrow [P\&R]output.p_i = v_i)$ (Req. (i))

    $\bigwedge_{i=1}^{n}[P\&R]\neg(output.p_i = undef_i)$ (Req. (ii))

    $[P\&R]constr\_viol(output.p_1, ..., output.p_n) = \emptyset$ (Req. (iii))

end object class

Again, we seek to further specify the behaviour of Propose and Revise by giving a constructive description of its dynamics. In LCM, for technical reasons this is done slightly different compared to TROLL. Like in TROLL, a new class is defined that represents the store 'design', and the inference actions Init, Evaluate, Propose, Test and Revise are modelled as events that happen in the life of a Design object. The dynamics of Propose and Revise are then defined by an LCM life cycle definition for this new class as follows:

lifecycle

$\forall d : DESIGN : DESIGN(d) =$
$INPUT\_OUTPUT.set\_undef(output);$
$set\_undef(d);$
$init(d);$
repeat $propose(d);$
  $violating(d) := false;$
  $test(d);$
  if $violating(d)$ **then** $revise(d)$ **endif**
until $evaluate$
$copy(d);$

5 Comparison and Conclusions

In this section we will 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 the introduction.

States. With the exception of PDDL, where a state is a propositional valuation, a state is either an algebra (ASM and LCM) 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. Table 1 summarizes syntax and semantics of the specification of states.

Table 1. 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

Elementary Transitions. With respect to the specification of elementary transitions, two approaches can be distinguished: user-defined and pre-defined, 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 **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 behaviour 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 pre-defined, fixed set of elementary transitions, which resemble the assignment statement in programming languages. In (P)DDL, there are two predefined elementary transitions, and there is no possibility for the user to define additional ones. These predefined

Table 2. 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)

transitions are $\mathcal{I}^H p$ (set p to true) and $\mathcal{D}^H p$ (set p to false) 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 2 summarizes syntax and semantics of the specification of elementary transitions.

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 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. 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. 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 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.

Table 3. 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	Transitions can be guarded, bulk updates and choice between transitions is expressible. Sequencing or iteration is not expressible	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

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 3 summarizes syntax and semantics of the specification of composed transitions.

Proof Systems and Operationalization. The third aspect identified in Section 2.2, proof systems and operationalization, is summarized in Table 4.

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.

Table 4. 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
TROLL	TROLL is translated to OSL; there is a Hilbert-style proof system for OSL	Provided for TROLL*light*, a restricted version lacking the temporal language
LCM	Equational logic	

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. 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.

Modularity. 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 modelling 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.

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, 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.

The languages differ in their treatment of intermediate states that might occur during a transition from an initial to a terminal state. In DDL, as in dynamic logic on which DDL is based, there is no representation of any intermediate states of a program execution: any execution is represented as a pair of initial and terminal states (step-based control specification). Similar properties hold for the other languages, with the exception of TROLL and $\mathcal{TR}$. In these languages, the execution of a program is represented as a sequence of intermediate states (sequence-based control specification). As explained in Section 2.1, this has important consequences for the representation of the reasoning process in a KBS.

A final point concerns the treatment of non-terminating processes. Such non-terminating processes might occur in the specification of knowledge-based systems for process control and monitoring. TROLL, LCM and ASM can all deal with such non-terminating processes. Although it is of course possible to specify non-terminating processes in (P)DDL and $\mathcal{TR}$, it is not possible to derive any useful properties of such programs because in (P)DDL and $\mathcal{TR}$, non-terminating processes have trivial (empty) semantics.

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 behaviour 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 post-conditions 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 [5].

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 pre-defined 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 explicitly). 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.

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.

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, as well as ASM nowadays aim at being a full grown specification language, while formalisms like ASM, $\mathcal{TR}$ and (P)DDL aim instead at providing a framework (a logical framework, in the case of $\mathcal{TR}$) that should be used as the foundation of a specification, rather then being a specification language themselves.

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, and not just an ordered pair of begin- and end-state of a program. In $\mathcal{TR}$ and Troll this is possible using formulae that constrain the set of possible sequences.

It might, however, be possible to overcome the apparent shortcoming of the interpretation of composed transitions as ordered pairs of states suggested here. States can be seen as an abstraction mechanism that define equivalence classes of sequences of states, or histories. For example, in (P)DDL, any two histories that have the same begin- 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, if 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 point 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.

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 [12, Ch.4]: "The modelling 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 very 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

We are grateful to E. Börger, M. Kifer, G. Saake and R. Wieringa for their comments on an earlier version of this paper.

References

1. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.
2. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.
3. A.J. Bonner and M. Kifer. Transaction logic programming. In *Proceedings of the Tenth International Conference on Logic Programming (ICLP)*, pages 257–279, Budapest, Hungary, 1993. MIT Press.
4. F. Brazier, P. van Langen, J. Treur, N. Wijngaards, and M. Willems. Modelling an elevator design task in DESIRE: the VT example. *International Journal of Human-Computer Studies, Special Issue on Sisyphus-VT (A.Th. Schreiber and W.P. Birmingham, Eds.)*, 44(3–4):469–520, 1996.
5. G. Denker, J. Ramos, C. Caleiro, and A. Sernadas. A linear temporal logic approach to objects with transactions. In Michael Johnson, editor, *Algebraic Methodology and Software Technology: 6th International Conference, AMAST '97*, volume 1349 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, 1997.
6. P. van Eck, J. Engelfriet, D. Fensel, F. van Harmelen, Y. Venema, and M. Willems. A survey of languages for specifying dynamics: A knowledge engineering perspective. Technical Report IR-447, Vrije Universiteit Amsterdam, Faculty of Mathematics and Computer Science, 1998.
7. D. Fensel. Formal specification languages in knowledge and software engineering. *The Knowledge Engineering Review*, 10(4), 1995.
8. D. Fensel. *The Knowledge Acquisition and Representation Language KARL*. Kluwer Academic Publ., Boston, 1995.
9. D. Fensel and R. Groenboom. MLPM: Defing a semantics and axiomatization for specifying the reasoning process of knowledge–based systems. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 1996.
10. D. Fensel and R. Straatman. The essense of problem-solving-methods: Making assumptions for gaining efficiency. *Journal of Human Computer Studies*, 1998. (to appear).
11. D. Fensel and F. van Harmelen. A comparison of languages which operationalize and formalize KADS models of expertise. *The Knowledge Engineering Review*, 9(2), 1994.
12. D. Gabbay. *What is a Logical System?*, volume 4 of *Studies in Logic and Computation*. Oxford University Clarendon Press, 1994.
13. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.
14. D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II: extensions of Classical Logic*, pages 497–604. Reidel, Dordrecht, The Netherlands, 1984.
15. F. van Harmelen and J. Balder. $(\mathrm{ML})^2$: A formal language for KADS conceptual models. *Knowledge Acquisition*, 4(1), 1992.

16. C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 2nd edition, 1990.
17. R. Jungclaus, G. Saake, Th. Hartmann, and C. Sernadas. TROLL—a language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.
18. F. Kroeger. *Temporal Logic of Programs.* Springer–Verlag, Berlin, 87.
19. I. van Langevelde, A. Philipsen, and J. Treur. Formal specification of compositional architectures. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, August 1992.
20. M. Linster (ed.). Special issue on the Sisyphus 91/92 models. International Journal of Man-Machine Studies 40:2, 1994.
21. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
22. R. Milner. *Communication and Concurrency.* Prentice Hall Int., New York, 1989.
23. B. Nebel. Artificial intelligence: A computational perspective. In G. Brewka, editor, *Principals of Knowledge Representation*, Studies in Logic, Language and Information, pages 237–266. CSLI Publications, 1996.
24. C. Pierret-Golbreich and X. Talon. TFL: An algebraic language to specify the dynamic behaviour of knowledge-based systems. *The Knowledge Engineering Review*, 11(3):253–280, 1996.
25. K. Poeck, D. Fensel, D. Landes, and J. Angele. Combining KARL and CRLM for designing vertical transportation systems. *International Journal of Human-Computer Studies, Special Issue on Sisyphus-VT (A.Th. Schreiber and W.P. Birmingham, Eds.)*, 44(3–4):435–467, 1996.
26. A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, October 1995.
27. J. W. Spee and L. in 't Veld. The semantics of $\mathrm{K_{BS}SF}$: A language for KBS design. *Knowledge Acquisition*, 6, 1994.
28. J. M. Spivey. *The Z Notation. A Reference Manual.* Prentice Hall, New York, 2nd edition edition, 1992.
29. P. Spruit, R. Wieringa, and J.-J. Meyer. Dynamic database logic: the first-order case. In V.W. Lipeck and B. Thalheim, editors, *Fourth International Workshop on Foundations of Models and Languages for Data and Objects*, pages 102–120. Springer-Verlag, 1993.
30. P. Spruit, R. Wieringa, and J.-J. Meyer. Axiomatization, declarative semantics and operational semantics of passive and active updates in logic databases. *Journal of Logic and Computation*, 5(1), 1995.
31. J. Treur and Th. Wetter, editors. *Formal Specification of Complex Reasoning Systems.* Ellis Horwood, New York, 1993.
32. R. J. Wieringa. LCM and MCM: Specification of a control system using dynamic logic and process algebra. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes Computer Science*, pages 333–355. Springer-Verlag, 1995.
33. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers, 1990.