Specification of Dynamics for Knowledge-based Systems (extended abstract)

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

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 Evolving Algebras. This paper, which is a short version of a longer report, concentrates on the methodology of our comparison and on the conclusions we have drawn. The actual comparison between the languages has been removed from this version because of space limitations.

1 Introduction

Over the last few years a number of formal specification languages have been developed for describing *knowledge-based systems* (KBSs). Examples are DESIRE [vLPT92]; KARL [Fen95b]; $\mathrm{K_{BS}SF}$ [SitV94]; $(\mathrm{ML})^2$ [vHB92]; MLPM [FG96] and TFL [PGT96]. 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 [TW93, FvH94, Fen95a].[4]

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. [Neb96]). A specification has to describe

[1]Vrije Universiteit Amsterdam, Faculty of Mathematics and Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Email: {patveck,joeri,frankh,yde}@cs.vu.nl

[2]Institut AIFB, University of Karlsruhe, D-76128 Karlsruhe, Germany. Email: dfe@aifb.uni-karlsruhe.de

[3]Cycorp Inc., 3721 Executive Center Drive, Suite 100, Austin TX, 78731-1615, USA. Email: willems@cyc.com

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

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: t 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 ([Har84]), KARL integrates ideas of logic programming with dynamic logic, and TFL uses process algebra in the style of [BK85]. 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 axiomatization or proof calculus 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) of ([Wie95]) and TROLL ([Jun93]) as examples from the information systems area. Both languages provide means to express the dynamics of complex systems.

- Transaction Logic ([BK93]), Database Update Logic (PDDL ([SWM95]) and DDL ([SWM93])) as examples for database update languages which provide means to express dynamic changes of databases.

- Evolving Algebras ([Gur94]) from the theoretical computer science and software engineering area. 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. [Wir90]), which provide means for a functional specification of a system, and model-based approaches like Z [Spi92] and the Vienna Development Method - Standard Language (VDM-SL) [Jon90], 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 [Fen95a]. Finally, one may miss specification approaches like LOTOS [BB87] that are well-suited 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. Section 4 provides a short comparison between the formalisms according to our dimensions of analysis, and conclusions.

In this short version we concentrate on the methodology of our comparison and on the conclusions we have drawn. The actual comparison between the languages has been removed from this version because of space limitations. The interested reader is referred to the long version [vEEF+97].

2 The Two Dimensions of Our Analysis

In the analysis of the different frameworks, it will be convenient to distinguish two dimensions (see Figure 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.* It should also be possible to relate the description of the reasoning process to the declarative description of its desired effect. 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 calculus 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 an axiomatization based on a formal semantics has been developed. Therefore, we look at axiomatization and proof calculi. Restricted but still very useful support for the validation of specifications could be provided by prototyping or partial evaluation based on an operational semantics.

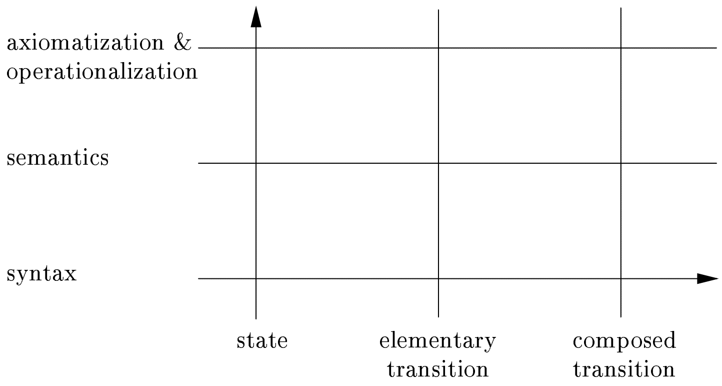

Figure 1: The two dimensions of our analysis

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 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. Note that current KBS specification languages do not provide this at all levels of specification. Actually, the equivalence of the functional specification of the goals (or task) and the external 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.

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 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 [Mil89]). 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. $(\mathrm{ML})^2$, 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 ([Kro87])), as an assignment to program variables (as in the first-order variant of Dynamic Logic), as an algebra (we will see that in Evolving Algebras), 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 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.

2.1.3 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., LCM 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 axiomatization/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 four elements 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.

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. 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.

2.2.3 Proof Calculus 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 calculus or 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. In that paper, 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 ([Lin94]), 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 logic for describing state and state change in deductive databases. It is based on Dynamic Logic, with special operators $\mathcal{I}^H p$ and $\mathcal{D}^H p$ where p is an atom, and H is a definite logic program. $\mathcal{I}^H p$ means that the fact p is inserted in the database, after which the database is closed under the rules of H, and $\mathcal{D}^H p$ means that p is deleted from the database, which is then closed under the rules of H. Apart from these operators, the language contains the Dynamic Logic operators for sequence, test and iteration. 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 calculus and an operational semantics is provided. The first-order variant (DDL) allows conditional insertion.

Transaction Logic ($\mathcal{TR}$)

$\mathcal{TR}$ is also a logic of state and state change in databases. In contrast to DDL, 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. Semantically, formulas are interpreted over sequences of database states (in contrast to DDL, where the meaning of a program is a binary relation on states). A program contains formulas which constrain the allowed sequences of states (these formulas are often in the form of Prolog-like rules). A proof system for the Horn fragment of the language is provided, and $\mathcal{TR}$ has an operational semantics.

Evolving Algebras (EA)

The basic concept of EA's is simple: an EA specification consists of rules that can (only) change the value of a function in a particular argument (they are of the form $f(t) := s$). A run of an EA is a sequence of algebras generated by consecutively firing all the rules in the current algebra. Many extensions of these rules exist, notably conditional function updates. EA do not have a fixed proof system, but rather the user is to employ general mathematical proof techniques.

Troll

TROLL is an object-oriented specification language for information systems, and has a rich syntax. Objects with attributes and events can be specified, together with interactions between objects. The effects of events can be specified by giving constraints on the behaviour of the objects through time. The semantics of TROLL are obtained via a translation into OSL, a temporal logic for reasoning about objects. This logic is equipped with a proof system. An execution mechanism is provided for a fragment of TROLL (lacking the temporal language).

Language for Conceptual Modeling (LCM)

LCM was developed as a tool for the conceptual analysis of object-oriented databases. The basic language of LCM is equational logic (for specifying abstract data types). These data types can be used to specify objects. Finally, some version of Dynamic Logic may be used to specify effects and preconditions of actions (LCM is parameterized by the choice of the operators). LCM has a proof calculus.

4 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.

4.1 A Short Comparison

We will give a brief overview of the frameworks in terms of the three concepts mentioned in the introduction.

4.1.1 States

With the exception of PDDL, where a state is a propositional valuation, a state is either an algebra (EA 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 EA 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 TROLL, functions are not allowed to vary, however in $\mathcal{TR}$ they are.

4.1.2 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 EA, 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 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. (DDL adds to this the possibility to perform parallel updates and choice.) 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 EA, there is only one type of elementary transitions, namely function updates expressed as $f(t) := s$, which links two algebra's A and A' that only differ in the values for $f(t)$. 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.

4.1.3 Composed Transitions

In EA, the main possibility to specify composed transitions is to make them conditional using an if-then-else construction. 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 and (P)DDL, elementary transitions can be composed using a syntax derived from process algebra, which also amounts to having sequencing, iteration and choice for composition. 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.

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.

4.2 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.

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 favour of either approach resemble very much the arguments in favour 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 favour 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 always 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 EA. 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 EA it seems that there is no possibility at all to include any control vocabulary in the language: EA 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 2.1.3, 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 EA 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.

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 EA give us the means to make black box specifications. However, it is not possible *within* the EA framework to specify the behaviour of such black boxes, which by implication also precludes the possibility of proving within the EA framework that a given implementation (refinement) of a black box satisfies the specifications. The designers of the EA framework prefer to use general mathematical techniques for treating refinement. The simple mathematical structure of the EA framework makes this feasible.

Although the transaction base from $\mathcal{TR}$ resembles the external functions of EA, $\mathcal{TR}$ is stronger than EA in this respect: the transaction base can be used to model black-box transitions, but unlike the external functions in EA the transitions of $\mathcal{TR}$ can be specified by means of pre- and post-conditions. 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 a way 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.

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. 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 a proof calculus 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 calculus. TROLL has to rely on its translation to OSL in order to use the axiomatization of OSL, while EA relies on general mathematical reasoning, without a formal proof calculus. LCM has a proof calculus 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, EA 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 do indeed aim at being a full grown specification language, while formalisms like EA and $\mathcal{TR}$ 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.

4.3 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 M. Kifer for his comments on an earlier version of this paper.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BK93] 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.

[Fen95a] D. Fensel. Formal specification languages in knowledge and software engineering. *The Knowledge Engineering Review*, 10(4), 1995.

[Fen95b] D. Fensel. *The Knowledge Acquisition and Representation Language KARL.* Kluwer Academic Publ., Boston, 1995.

[FG96] 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.

[FvH94] 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.

[Gur94] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1994.

[Har84] 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.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 2nd edition, 1990.

[Jun93] R. Jungclaus. *Modeling of Dynamic Object Systems—A Logic-based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, 1993.

[Kro87] F. Kroeger. *Temporal Logic of Programs.* Springer–Verlag, Berlin, 87.

[Lin94] M. Linster (ed.). Special issue on the sisyphus 91/92 models. International Journal of Man-Machine Studies 40:2, 1994.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall Int., New York, 1989.

[Neb96] 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.

[PGT96] 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.

[SitV94] J. W. Spee and L. in 't Veld. The semantics of KBSSF: A language for KBS design. *Knowledge Acquisition*, 6, 1994.

[Spi92] J. M. Spivey. *The Z Notation. A Reference Manual.* Prentice Hall, New York, 2nd edition edition, 1992.

[SWM93] 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.

[SWM95] 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.

[TW93] J. Treur and Th. Wetter, editors. *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, New York, 1993.

[vEEF+97] P. van Eck, J. Engelfriet, D. Fensel, F. van Harmelen, Y. Venema, and M. Willems. Specification of dynamics for knowledge-based systems. Technical report, Vrije Universiteit Amsterdam, Faculty of Mathematics and Computer Science, 1997.

[vHB92] F. van Harmelen and J. Balder. $(\mathrm{ML})^2$: A formal language for KADS conceptual models. *Knowledge Acquisition*, 4(1), 1992.

[vLPT92] 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.

[Wie95] 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.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.