Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-93-08

CoLab:
A Hybrid Knowledge Representation and Compilation Laboratory

Harold Boley,
Philipp Hanschke, Knut Hinkelmann, Manfred Meyer

January 1993

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

COLAB:
A Hybrid Knowledge Representation and Compilation Laboratory

Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer

DFKI-RR-93-08

To appear in a special issue of *Annals of Operations Research*
3rd International Workshop on Data, Expert Knowledge and Decisions
Reisensburg Castle, September 1991

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-8902 C4).

CoLab:
A Hybrid Knowledge Representation and Compilation Laboratory

Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer

DFKI
Kaiserslautern, Germany
{boley,hanschke,hinkelma,meyer}@dfki.uni-kl.de

Abstract

Knowledge bases for real-world domains such as mechanical engineering require expressive and efficient representation and processing tools. We pursue a declarative-compilative approach to knowledge engineering.

While Horn logic (as implemented in PROLOG) is well-suited for representing relational clauses, other kinds of declarative knowledge call for hybrid extensions: functional dependencies and higher-order knowledge should be modeled directly. Forward (bottom-up) reasoning should be integrated with backward (top-down) reasoning. Constraint propagation should be used wherever possible instead of search-intensive resolution. Taxonomic knowledge should be classified into an intuitive subsumption hierarchy.

Our LISP-based tools provide direct **translators** of these declarative representations into abstract machines such as an extended Warren Abstract Machine (WAM) and specialized inference engines that are interfaced to each other. More importantly, we provide source-to-source **transformers** between various knowledge types, both for user convenience and machine efficiency.

These formalisms with their translators and transformers have been developed as part of CoLab, a compilation laboratory for studying what we call, respectively, 'vertical' and 'horizontal' compilation of knowledge, as well as for exploring the synergetic collaboration of the knowledge representation formalisms.

A case study in the realm of mechanical engineering has been an important driving force behind the development of CoLab. It will be used as the source of examples throughout the paper when discussing the enhanced formalisms, the hybrid representation architecture, and the compilers.

Keywords and Phrases: Hybrid Knowledge Representation, Knowledge Compilation, Declarative Programming, Taxonomic Reasoning, Constraint Propagation, Relational-Functional Compuation, Bottom-up Deduction, Knowledge-Based Systems, Mechanical Engineering, NC-Programming

Contents

1 Introduction

A long-term goal of our research is to understand the principles of building knowledge bases (KBs) for real-world domains such as mechanical engineering (e.g., CAD/CAM). We feel that it will be important to reuse much of the domain-specific knowledge in a KB for **several** task categories such as both for (process) planning and quality control. After all, human experts are mostly domain experts, capable of performing all kinds of tasks within their domain. For instance, an NC-programmer can both write and debug programs because his domain models of the application's objects and structures, the task environment, the programming and database languages, the operating system, and the hardware equipment are reusable for both synthetic and analytic tasks.

High reusability of KBs will require declarative and expressive representation languages as well as flexible and efficient compilation tools for them. Declarative representations describe logically **what** the knowledge expresses without at the same time prescribing imperatively **how** it is to be used. Such a high descriptive level not only permits several uses of the same KB but also enhances the readability, maintenance, and parallelization of KBs. Moreover, the orientation towards logic (usually, variations of first-order predicate calculus) permits a clear semantics for representation languages and eases the tough business of KB verification/validation. All this will be particularly relevant as KBs are growing larger: it is becoming increasingly important to facilitate various kinds of "knowledge analysis" analogous to data analysis.

If knowledge representation does not prescribe knowledge use, how is knowledge going to be applied to the problems at hand? The answer of the declarative paradigm has traditionally been a general-purpose control component for interpreting problem requests ('goals', 'queries') with respect to a KB. Since the efficiency of language interpreters is normally not sufficient for large KBs, some knowledge-based systems and expert-system shells have generalized the **compilation** concept of ordinary programming languages: a declarative KB can be statically (before users input their queries) analyzed and accordingly rewritten into another, usually more efficient (procedural), form. Compiled KBs often run in abstract machines such as PROLOG's Warren Abstract Machine (WAM); except for their increased efficiency, they are not a concern of end-users. Advanced compilation methods thus enable users to write KBs declaratively while achieving the same efficiency procedural paradigms are offering: knowledge compilation can make the declarative paradigm a practical option for artificial intelligence (AI) programming.

As a step in that direction we have implemented a prototypical knowledge compilation laboratory, CoLab, on the basis of LISP. Supported by a battery of compilation tools, it provides a hybrid integration of four principal declarative representation languages, developing and extending well-known AI formalisms (Section 2). While Horn logic (as implemented in PROLOG) is well-suited for representing (0) relational facts and rules, other kinds of declarative knowledge call for several extensions: (1) functional dependencies/associations and higher-order knowledge should be modeled directly (Section 5). (2) forward (bottom-up) reasoning should be integrated with backward (top-down) reasoning (Section 6). (3) Constraint propagation should be used wherever possible instead of search-intensive resolution (Section 4). (4) Taxonomic knowledge should be collected into intuitive and efficient subsumption hierarchies (Section 3). We permit KBs to consist of several **types** of items (marked by infixes or tags, Appendix A), refining the knowledge kinds (0)-(4). This hybrid language will be construed as an 'extended-ABox' KL-ONE language

in the tradition of KRYPTON [18] by collecting (0)-(3) into an **affirmative component**, whose knowledge is structured by the 'type' system of (4), the **taxonomic component**.

Besides **interpreting** its hybrid language for interactive KB development, CoLab provides **source-to-code translators** for compiling KBs down to efficient abtract machines. Some of these translators employ a functionally extended WAM, called RFM. Also, CoLab provides **source-to-source transformers** between various knowledge types, for both user convenience and machine efficiency: 'hybrid' knowledge can be transformed to several 'homogeneous' forms and (with interaction) vice versa, permitting tailored user representations and machine implementations. For example, a hybrid KB whose items are forward and backward rules can be homogenized to bidirectional rules; conversely, an interactive transformer can split bidirectional rules according to their actual use directions, proceeding from a more declarative to more procedural representations. Both translators and transformers are being developed for studying the trade-offs between what we call, respectively, 'vertical' and 'horizontal' compilation of knowledge (Section 2.2).

While vertical compilation has the obvious purpose of making KBs ultimately efficient, it is often preceded by preparatory horizontal compilation steps, permitting, complexity-reducing optimizations at the highest possible algorithmic level. Horizontal compilation can also help hybrid shells to avoid the costly development of a complete vertical compiler and emulator (run-time system) for each individual subformalism, with the associated problems of dynamic interfaces between these run-time systems: by horizontally reducing a subformalism F1 (e.g. functional nestings) to a subformalism F2 (e.g. relational conjunctions) we may circumvent the vertical compiler for F1; if the F1-to-F2 transformation (e.g. Horn clauses to constraints) is only partial but works for the KBs of a particular domain, we still have the option to concentrate our vertical-compiler development on F2 for speeding up the knowledge represented in those domain KBs. Moreover, horizontal compilation is important for gaining flexibility in the very development of future (hybrid) representation languages: since in the AI community it is not yet clear which subformalisms should be part of an 'ideal' hybrid shell or whether one homogeneous formalism could replace hybrid shells altogether, it is important to be able to 'save' the knowledge invested in a hybrid subformalism F1 by its horizontal transformation to some subformalism F2, should F1 be abandoned and F2 be kept. A related point concerns horizontal transformations between specialized KB languages and a standard knowledge-sharing language such as KIF [30], permitting knowledge reuse. Finally, horizontal compilers can support a team of developers of a KB: if there are good transformation tools, everybody may freely use the language of their choice from the hybrid shell, which may differ from the language in which the KB will be represented for fellow developers or delivered to end users. On the other hand, horizontal transformation has the potential disadvantage of perpetuating some dependence of a language on the (moving) target language to which it is transformed; vertical compilation provides independence between languages on the same level by separately translating them down to the next lower (more efficient) level.

Therefore, with CoLab we have provided a **laboratory** of vertical as well as horizontal compilers, which enable the development, processing, and maintenance of knowledge formulated in evolving hybrid, declarative languages. This 'coherence-through-compilation' approach to hybrid, declarative KBs is at variance with the 'coherence-through-presentation' approach to hybrid, procedural KBs found in many commercial expert-system shells.

In the domain of mechanical engineering we have realized a non-toy application probing the declarative expressiveness of CoLab: The μCAD2NC system applies the "heuristic classifica-

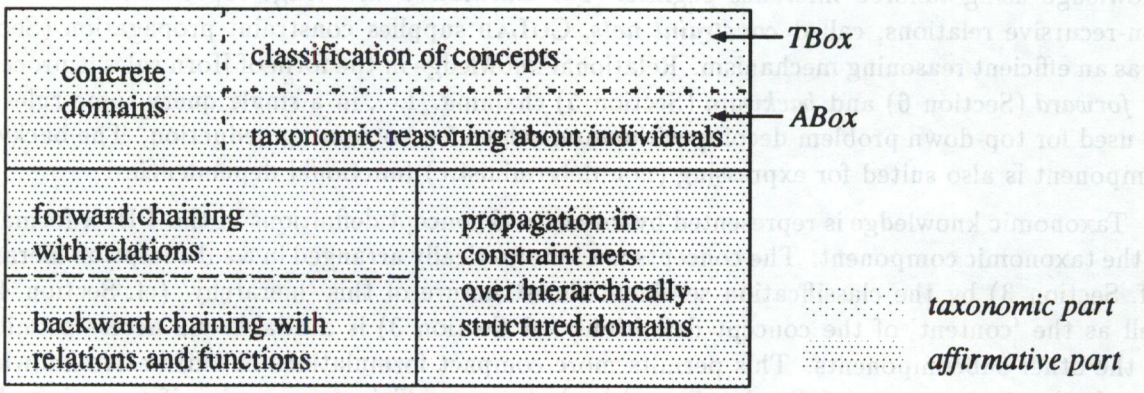

Figure 1: The CoLab Representation Architecture

tion" inference scheme to a subtask arising in CIM, namely computer-aided process planning (CAPP), transforming a (declaratively represented) CAD-like workpiece geometry into a (declarative representation of an) NC-program for manufacturing the workpiece. Two versions of this application have explored alternative uses of CoLab's representation and compilation capabilities, and have provided a challenging test environment for several CoLab developments. With the second μCAD2NC version we have also reached a stable CoLab, demonstrating that our hybrid declarative-compilative approach is viable for knowledge-based systems (Section 7, Appendix B).

2 The CoLab Representation and Compilation Architecture

2.1 The Representation Architecture

The CoLab system has been designed as a *compilation laboratory* aiming at a synergetic collaboration of different knowledge representation and reasoning formalisms. It is comprised of subcomponents dealing with the different kinds of knowledge and that can also be used in a stand-alone manner. The components are indicated in Figure 1 as dashed boxes. Dynamic cooperation of the components is based on access primitives providing interfaces between the reasoning services. In this paper we shall introduce also conceptually more involved integrations of the components whose implementation encapsulates the access primitives, thus hiding them from hybrid KBs. A hybrid KB of CoLab can be composed of items from *all* components which are managed as a unit. An infix syntax (or a prefix tag) indicates the type of knowledge items and determines how they are processed. This facilitates the development of knowledge modules that comprise different kinds of knowledge, and thus employ more than one formalism.

Similarly as in terminological knowledge representation systems such as KL-ONE [20], the CoLab representation architecture splits into two main parts, an *affirmative* part, sometimes also called 'assertional', and a *taxonomic* part. We use the term 'ABox' to denote the assertional formalism managed by the taxonomic component Taxon.

The former part provides efficient reasoning with different kinds of relational or functional

knowledge using tailored inference engines. For affirmative knowledge represented as net-like, non-recursive relations, called *constraint nets*, CoLab supplies constraint propagation (Section 4) as an efficient reasoning mechanism. Relational knowledge in the form of Horn rules is processed by *forward* (Section 6) and *backward* (Section 5) chaining. I.e., in a single query some rules can be used for top-down problem decomposition and others for bottom-up deduction. The backward component is also suited for expressing (non-deterministic) functional dependencies.

Taxonomic knowledge is represented by intensional concept definitions in the TBox formalism of the taxonomic component. The concepts are automatically arranged in a subsumption hierarchy (cf. Section 3) by the classification service. The structure of this 'hierarchy' (cf. Section 4) as well as the 'content' of the concept definitions (cf. Section 3) is available via access primitives to the other subcomponents. This permits more compact formulations of affirmative knowledge by referring to concepts (cf. Section 7) and leads to more efficient processing if reasoning about instances can be lifted to reasoning on the concept level (cf. Section 4).

2.2 An Overview of Compilation

CoLab, designed as a knowledge compilation laboratory, provides several compilation tools for its representation languages. All knowledge representation formalisms available in CoLab are integrated in a LISP-based system, hence are available in one runtime environment. Each of them provides an efficient inference engine tailored to the specific needs of that representation formalism:

- efficient algorithms for various reasoning services of the terminological component (e.g. subsumption, satisfiability test, or classification);
- an extended hierarchical arc-consistency algorithm for propagating constraints over hierarchically structured domains, working on an object-oriented (CLOS) representation of the constraint net;
- a functionally extended higher-order Horn-clause prover based on SLV resolution for backward chaining of valued clauses, interpreting/compiling functional values and nestings.
- a semi-naive evaluation strategy for forward rules which avoids multiple derivations of the same facts, exploiting a generalized *magic-set* transformation to restrict the number of derived facts.

2.2.1 Vertical Compilation

Run-time efficiency can be improved by pre-evaluating parts of the computation and optimizing the data representation for the specific way data are accessed and modified. A native-code compiler for some procedural programming language generates the sequence of machine instructions more specialized than those which would be executed by a high-level interpreter for that language. In principle, the same technique can be applied by defining a (software) *abstract machine* for which the compiler will generate code. Such an abstract machine defines specialized *abstract instructions* working on data representations designed to support the efficient implementation of these

instructions. Such an abstract machine can then be realized by defining a low-level interpreter for the abstract instructions which *emulates* the behavior of the abstract machine.

A very elaborate abstract-machine model for implementing logic programming and thus logic-oriented knowledge-representation languages has been defined in [63], and is known as the Warren Abstract Machine (WAM). In principle, the WAM, specially designed as an abstract machine for evaluating Horn-clause programs using SLD-resolution, supports a set of instructions which implement all aspects of the most time-consuming *unification* task. The compiler, then, has to analyze the program and to generate those abstract unification instructions needed in each specific case. Together with an efficient data representation supporting the variable bindings management and the restoration of earlier computation states in case of backtracking, this results in a much better runtime performance than an interpretation of the original source program.

For this reason, CoLab also uses such techniques and supports 'vertical compilation' tools which translate a subset of the entire knowledge representation language vertically 'down' to an extended WAM. The abstract instructions are then processed by a WAM emulator, also implemented in LISP and hence available within the same runtime environment.

Currently, WAM-oriented vertical compilers are available for the relational-functional component (see Section 5, [12]) as well as for the rule component (see Section 6, [35]).

Additionally, the semi-naive evaluation strategy for bottom-up rules (see Section 6) and the hierarchical arc-consistency algorithm for constraints (see Section 4) also make use of vertical compilation techniques in order to translate the source KB into an internal representation more suitable for efficient processing.

Finally, an abstract machine for efficient bottom-up deduction has been developed, which is a modification of the Rete pattern match algorithm [26]. It has been extended to support hybrid reasoning. In particular, a homogeneous interface to the WAM is available, because both machines are implemented on equivalent levels of abstraction. The instruction sets and the term representations are very similar.

All these compilation aspects will be discussed in more detail in the following sections on the individual components of CoLab.

2.2.2 Horizontal Compilation

Besides the vertical compilation tools, which have been developed only for efficiency reasons, CoLab provides a set of 'horizontal' compilation tools which perform knowledge transformations in a source-to-source manner on the same level of abstraction.

As CoLab allows knowledge to be represented in a hybrid way by using the most appropriate representation formalism for each piece of knowledge (*knowledge item*), such horizontal transformers are very useful both for user convenience and runtime efficiency. They support, e.g., the transformation of 'hybrid' knowledge to several 'homogeneous' forms and thus can also provide something like *normal forms* for hybrid KBs.

CoLab KBs consist of a hierarchical system of knowledge items. Most knowledge items constitute concept definitions, rules (bottom-up, top-down, or bidirectional), or constraint definitions. Knowledge items can be distinguished syntactically; a summary of all CoLab knowledge items

is given in appendix A.

The main idea behind the CoLab architecture is to provide a set of interactive compilation tools which allow CoLab users—knowledge engineers designing an application—to interactively discover the best way of representing the kind of knowledge they are currently trying to implement. For instance, it is possible in CoLab to abstractly represent some piece of knowledge as rules without saying whether they shall be processed using the backward chaining or the forward chaining (bottom-up) component. Such *bidirectional* rules can later be interactively split into more concrete forward and backward rules according to their optimal use directions.

By using such interactive knowledge-compilation tools the user can experiment with different knowledge representations and is not forced to decide, in an early phase, once and for all, which representation formalism to use for which subtask, as is usually the case with common expert-system shells.

Horizontal knowledge-compilation tools partially or totally transform knowledge represented in one CoLab component into a representation of another formalism. Examples for such horizontal knowledge transformation in CoLab are:

- A source-to-source compiler for general rules with multiple conclusions into Horn rules for top-down evaluation (see Section 6);
- a transformer for bidirectional rules into backward-chaining clauses (see Section 6);
- partially compiling Horn clauses into primitive and compound constraints (see Section 5) and vice-versa (see Section 4);
- transliterating concept definitions into Horn clauses enhanced by a RelFun implementation of Taxon's reasoning services (see Section 5).

Since the representation languages provided in CoLab differ in their expressiveness, not all of these transformations can be realized as total mappings from one representation formalism into the other. For instance, while the full compilation of primitive and compound constraints into Horn clauses causes no problems, the opposite direction only works for a restricted class of Horn clauses (see Section 4). The various horizontal compilers available to the CoLab user are discussed in more detail in the sections on the formalisms that constitute the source of the compilation.

Besides these horizontal *inter-formalism* compilers there are also several transformation tools that perform horizontal compilation of knowledge within a single formalism (*intra-formalism* compilers). Examples of such intra-formalism compilers are

- the *flattener* for relational-functional clauses (see Section 5),
- the *folding* mechanism for primitive constraints (see Section 4),
- the implementation of bottom-up rules for goal-directed reasoning by an extended *generalized magic-set* strategy (see Section 6), and
- the *concept-classification* service that computes subsumption relations between concepts and constructs the cover graph used by other CoLab components (see Section 3).

The following sections discuss the representation and compilation tools of CoLab from the viewpoints of the four available components.

3 Taxonomic Reasoning

The taxonomic component, called Taxon, is a terminological knowledge representation system belonging to the family of systems (e.g. [19, 17, 40]) that originated with KL-ONE [20]. Two of the main advantages of the formalisms of this family is their precise declarative semantics and their adequacy for a human user. To make full use of the exact declarative semantics we employ sound and complete algorithms for terminological knowledge representation as developed in [57] and further elaborated e.g. in [38, 24, 4].

The intended use of CoLab in the realm of mechanical engineering heavily influenced the design of the taxonomic component. For example, in our sample application (Section 7) it is necessary to represent technological and geometrical aspects of lathe workpieces as well as more abstract features relevant for production planning. Taxonomic formalisms usually allow only the definition of concepts on an abstract logical level. But—and not only—in this application domain there is a need for reference to more concrete notions. For example, the adequate definition of geometric concepts requires to relate points in a co-ordinate system [5]. Similar motivations have already led to extensions of KL-ONE. The MESON system provides "a separate hierarchy for describing non-concepts (e.g., integer ranges and strings)" ([52], p. 8) which are given as user-defined or machine-defined predicates. The "test" construct in CLASSIC also provides access to concrete notions. In K-REP "the roles of concepts may in turn be other (complex) concepts, as well as numbers, strings and ... arbitrary Lisp objects" ([44], p. 62). Schmiedel's Temporal Terminological Logic [58] can also be seen in this light. In this case the concrete domain is given by an extension of Allen's interval calculus [2].

In [4] a scheme for extending concept languages with *concrete domains* is proposed. Already on the scheme level it is shown how well-known reasoning algorithms of concrete domains can be employed to get sound and complete algorithms for e.g. subsumption and realization [51] provided the chosen domain is *admissible*. This scheme has been extended w.r.t. quantification over attribute/role chainings and role interaction [31]. The taxonomic component Taxon of CoLab is an instance of a generalized version of this scheme that provides enhanced means to specify the interaction of roles.

The taxonomic component encompasses not only a formalism to deal with the intensional concept definitions. In addition, it possesses a formalism that is capable to instantiate concepts by instances. For this very restricted use of taxonomic knowledge well understood reasoning services such as membership test, realization, or consistency test are provided. Through these services, the affirmative part of CoLab has full access to the 'content' of the concept definitions. This enables new integrations of affirmative and taxonomic knowledge representation formalisms.

For example, the concepts can be regarded as an expressive, tailored vocabulary for formulating premises and conclusions of a rule (Section 7). This shows that the affirmative components are not restricted to use concepts purely as sorts in an extended unification.

In the remainder of this section we shall formally define the taxonomic formalism (Section 3.1), give an example (Section 3.2), discuss compilation aspects, and relate the component to the

other formailsms of CoLab (Section 3.3).

3.1 Formal Definitions

As already mentioned, the terminological component encompasses two parts, called ABox (box for assertional knowledge) and TBox (box for terminological knowledge). The concept definitions in the TBox are strictly intensional, and, do not refer to instances. Thus, TBox reasoning is reasoning about concepts independent of specific cases. Whereas the knowledge items of the ABox correspond to specific cases (observations) in the world that instantiate the vocabulary.

Concepts can be seen as unary predicates that are constructed from other concepts, roles and predicates using certain operators. Roles are binary predicates relating instances (i.e., members of concepts) to instances. We distinguish functional roles, which we call *attributes*, and *many-valued roles*. The former may relate an instance to at most one other instance, the latter to an arbitrary number. If we want to be unspecific we say *role*.

Taxon handles additionally abstract and concrete n-ary predicates, $n > 0$. The abstract predicates are atomic (i.e., not further defined). *Concrete predicates* belong to a *concrete domain* which consists of a set $dom(\mathcal{D})$ and a collection of concrete predicate names $\mathcal{P}_\mathcal{D}$ that structure the domain with fixed extensions $p^\mathcal{D} \subseteq dom(\mathcal{D})^n$, where n is the arity of p. As usual, the superscript $^\mathcal{D}$ is sometimes omitted. A concrete domain is called *admissible*, if it satisfies the following properties.

1. If $p \subseteq dom(\mathcal{D})^n$ is an n-ary predicate of the concrete domain, then there is a $\overline{p} \in \mathcal{P}_\mathcal{D}$ such that $\overline{p}^\mathcal{D} = dom(\mathcal{D})^n \setminus p$.
2. There is a decision procedure for the satisfiability problem of finite conjunctions of these predicates (with possibly shared variables). I.e., the algorithm has to check whether there is an assignment of elements of the concrete domain to the variables in the conjunction such that the conjunction is satisfied.

As an example consider the domain of rational numbers with comparison operators as predicates. The former requirement is technical. For instance, it requires that if $<$ is a predicate of this concrete domain, then $\geq$ must belong to the domain, too. The latter, requirement says that there is an algorithm that performs the following task: Given a conjunction, say $x < 1 \wedge x > y \wedge y = 1.1$, it checks whether the variables can be instantiated by rational numbers such that the conjunction becomes true. Because $1 > 1.1$ does not hold, there is no such variable assignment in the example.[1]

Definition 3.1 (TBox) *There are five, pairwise disjoint, alphabets $\mathcal{C}$, $\mathcal{R}^{nn}$, $\mathcal{R}^{n1}$, $\mathcal{P}$, $\mathcal{P}_\mathcal{D}$ of names for concepts, many-valued roles, attributes, abstract predicates, and concrete predicates. The letter "Λ" $\in \mathcal{R}^{n1}$ is a special attribute name.*

A role chaining *is an expression $R_1 \circ \ldots \circ R_m$, $m > 0$, where $\circ$ is an associative, binary, infix operator, and each $R_j \in \mathcal{R}^{nn} \cup \mathcal{R}^{n1}$ is a role name. An* attribute chaining *is a role chaining where each role is an attribute.* Concept terms *are inductively defined. Every concept name $C \in \mathcal{C}$ is a*

[1] See also the definition of a CSP in Section 4.

concept term. If s and t are concept terms, $u_1, \ldots, u_n$ are role chainings, and v_1, v_2 are attribute chainings, then the following expressions are concept terms:

$s \sqcap t$	*(conjunction)*
$s \sqcup t$	*(disjunction)*
$\neg s$	*(complement)*
$\exists u_1, \ldots, u_n.\rho$	*(exists-in restriction)*
$\forall u_1, \ldots, u_n.\rho$	*(value restriction)*
$v_1 \downarrow v_2$	*(agreement)*
$v_1 \uparrow v_2$	*(disagreement)*

The expression ρ is called restrictor and has to match one of the following cases.

1. *$n = 1$ and ρ is a concept term,*
2. *ρ is p or $\neg p$ where p is an n-ary predicate name,*
3. *ρ is an n-ary concrete predicate name.*

A concept definition *is a pair (C, t) written as $C =_{\mathsf{conc}} t$ where C is a concept name and t is a concept term. A* terminology *is a finite sequence of concept definitions $C_1 =_{\mathsf{conc}} t_1$, $C_2 =_{\mathsf{conc}} t_2$, $\ldots$, $C_k =_{\mathsf{conc}} t_k$ such that, for $i = 1, \ldots, k$, C_i does not occur in $C_1 =_{\mathsf{conc}} t_1, \ldots, C_{i-1} =_{\mathsf{conc}} t_{i-1}, t_i$. I.e., the terminology does not contain cycles and there is at most one concept definition per concept name.*

A precise semantics for terminologies is obtained through a mapping of concept terms into first-order formulas, which are interpreted in the usual way. There is only a slight complication caused by the concrete domain. We require that each interpretation $\mathcal{I}$ for this formulas satisfies the following conditions:

1. The domain $dom(\mathcal{I})$ of the interpretation is disjoint to $dom(\mathcal{D})$.
2. A concept name C is interpreted as unary predicate $C^{\mathcal{I}} \subseteq dom(\mathcal{I})$, a name of a many-valued role $r \in \mathcal{R}^{nn}$ is interpreted as a binary predicate $r^{\mathcal{I}} \subseteq dom(\mathcal{I}) \times (dom(\mathcal{I}) \cup dom(\mathcal{D}))$, an attribute name $f \in \mathcal{R}^{n1}$ is interpreted as the graph of a partial function $f : dom(\mathcal{I}) \longmapsto dom(\mathcal{I}) \cup dom(\mathcal{D})$, a name for an n-ary abstract predicate $p \in \mathcal{P}$ is interpreted as $p^{\mathcal{I}} \subseteq dom(\mathcal{I})^n$, and a name of an n-ary concrete predicate $p \in \mathcal{P}_{\mathcal{D}}$ is interpreted as $p^{\mathcal{I}} := p^{\mathcal{D}} \subseteq dom(\mathcal{D})^n$
3. The special symbol $\top_{\mathcal{D}}$ (resp., $\top$) is interpreted as $dom(\mathcal{D})$ (resp., $dom(\mathcal{I})$) and the attribute name Λ denotes the identity $\{(x, x); x \in dom(\mathcal{I})\}$.

Note that roles are the only link between the abstract and the concrete domain. It remains to map concept terms and concept definitions into first-order formulas. This is done by inductively defining a family of mappings $\{\psi_x\}_x$ where x ranges over tuples of variables of the first-order language.

Then the ψ_x are inductively defined as follows:

1. $\psi_x : N \longmapsto N(x)$, if N is a concept, role, or predicate name.

2. $\psi_{x,y} : c \circ c' \longmapsto \exists z : (\psi_{x,z} c \wedge \psi_{z,y} c')$ where z is a fresh variable.

3. $\psi_x : s \sqcap t \longmapsto \psi_x s \wedge \psi_x t$,
   $\psi_x : s \sqcup t \longmapsto \psi_x s \vee \psi_x t$, and
   $\psi_x : \neg\rho \longmapsto \neg\psi_x \rho \wedge \top(x_1) \wedge \ldots \wedge \top(x_n)$ where $x = (x_1, \cdots, x_n)$ and ρ is a concept term or an abstract predicate.

4. $\psi_x : \forall u_1, \ldots, u_n.\rho \longmapsto \exists y_1, \ldots, y_n : ((\psi_{x,y_1} u_1 \wedge \cdots \wedge \psi_{x,y_n} u_n) \Rightarrow \psi_{y_1,\ldots,y_n}\rho)$ where the y_i are fresh variables.

5. $\psi_x : v_1 \downarrow v_2 \longmapsto \forall y_1, y_2 : ((\psi_{x,y_1} v_1 \wedge \psi_{x,y_2} v_2) \Rightarrow y_1 = y_2)$

6. $\psi_x : v_1 \uparrow v_2 \longmapsto \exists y_1, y_2 : ((\psi_{x,y_1} v_1 \wedge \psi_{x,y_2} v_2) \wedge y_1 \neq y_2)$

A concept definition $C = t$ is mapped to $\forall x : (\psi_x C \Leftrightarrow \psi_x t)$.

An interesting service provided by the taxonomic component is *subsumption*. A concept term s *subsumes* a concept t iff $\forall x : (\psi_x t \Rightarrow \psi_x s)$ is a theorem in the logical theory generated by the terminology. The *classification* service computes the subsumption graph of the subsumption relation, which is actually the cover graph of the relation. A concept t is *satisfiable* w.r.t. the current terminology iff $\psi_x t$ is satisfiable in the logical theory generated by the terminology.

In the ABox formalism it can be stated that instances belong to concepts and furthermore the relation-ship of the instances can be modeled by instantiating roles and predicates with them.

Definition 3.2 (ABox) *There is an alphabet of instances, disjoint to the other alphabets. Let C be a concept name, p a predicate name with arity n (either abstract or concrete) R a role name, and the $a, a_1, \cdots a_n, b$ instances. Then the following expressions are* assertions*:*

$C(a)$	*(membership assertion)*
$R(a,b)$	*(role-filler assertion)*
$p(a_1, \cdots, a_n)$	*(predicate assertion)*
$a = b$	*(equality)*
$a \neq b$	*(negated equality)*

Viewing instances as constants these assertions can be immediately read as closed formulas in our first-order theory.

An ABox is *consistent* if it is consistent as a set of logical formulas (in the theory generated by the terminology). An object a is a *member* of a concept (term) t, if $\psi_a t$ is a logical consequence of the current ABox (as a set of logical formulas). The *realization* of an object a is the set of most specific concepts in the subsumption graph such that for each element C in this set a is a member of C.

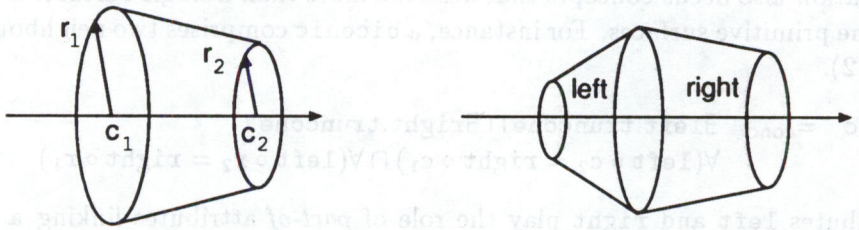

Figure 2: A Truncated Cone and a 'Biconic'

3.2 An Example

In Section 7 a prototypical process planning system is described that demonstrates the synergetic cooperation of the components of CoLab. In this section we develop an idealized terminology for this application in the domain of mechanical engineering.

The geometry, as the main ingredient of a CAD drawing, is given as a collection of rotational-symmetric surfaces that are fixed to the symmetry axis of the lathe work. An important geometric element is the truncated cone. Since the surfaces are fixed to an axis, they can be characterized by four rational numbers r_1, r_2, c_1, and c_2 (Figure 2).

But not all quadruples represent a truncated cone. So we have to restrict their values such that the radii are positive and the quadruples do not correspond to a line, a circle, or even a point. These restrictions are expressed by the four place predicate `truncone-condition` over the concrete domain of rational numbers.

$$\texttt{truncone} =_{\text{conc}} \exists(r_1, r_2, c_1, c_2).\texttt{truncone-condition}$$

This definition can be specialized to a cylinder by further restricting the radii as being equal using equality on rational numbers and the *conjunction* operator $\sqcap$. Similarly, the definitions of ascending and descending truncated cones, rings, etc. can be obtained by specialization. Truncated cones that are not cylinders are defined as the most specific generalization of ascending and descending truncated cones using the *disjunction* operator $\sqcup$. An equivalent definition would be $\texttt{not-cylinder} =_{\text{conc}} \texttt{truncone} \sqcap \forall(r_1 \neq r_2)$.

$$\begin{array}{lll}
\texttt{cylinder} & =_{\text{conc}} & \texttt{truncone} \sqcap \forall(r_1 = r_2). \\
\texttt{asc-tc} & =_{\text{conc}} & \texttt{truncone} \sqcap \forall(r_1 < r_2). \\
\texttt{desc-tc} & =_{\text{conc}} & \texttt{truncone} \sqcap \forall(r_1 > r_2). \\
\texttt{ring} & =_{\text{conc}} & \texttt{truncone} \sqcap \forall(c_1 = c_2). \\
\texttt{asc-ring} & =_{\text{conc}} & \texttt{ring} \sqcap \texttt{asc-tc}. \\
\texttt{desc-ring} & =_{\text{conc}} & \texttt{ring} \sqcap \texttt{desc-tc}. \\
\texttt{not-cylinder} & =_{\text{conc}} & \texttt{asc-tc} \sqcup \texttt{desc-tc}.
\end{array}$$

To improve readability, infix notation has been used for the comparison operators in the value restrictions.

The application also needs concepts that describe more than a single surface. So it is necessary to aggregate the primitive surfaces. For instance, a `biconic` comprises two neighbouring truncated cones (Figure 2).

$$\texttt{biconic} =_{\text{conc}} \exists\texttt{left.truncone} \sqcap \exists\texttt{right.truncone} \sqcap \forall(\texttt{left} \circ \texttt{c}_2 = \texttt{right} \circ \texttt{c}_1) \sqcap \forall(\texttt{left} \circ \texttt{r}_2 = \texttt{right} \circ \texttt{r}_1)$$

Here the attributes `left` and `right` play the role of *part-of* attributes linking a `biconic` to its components. Informally speaking, an object is a member of $\exists$`left.truncone` iff it has a truncated cone as a filler for `left`. The expression $\forall(\texttt{left} \circ \texttt{c}_2 = \texttt{right} \circ \texttt{c}_1)$ forces the right center of the left truncated cone to be equal to the left center of the right truncated cone.

If the CAD model provides an explicit topological model, it can be more sensible to define a `biconic` as

$$\exists\texttt{left.truncone} \sqcap \exists\texttt{right.truncone} \sqcap \forall\texttt{left,right.neighbouring}$$

where `neighbouring` is a binary, abstract predicate.

However, specializations of `biconic` are defined using the *value restriction* operator $\forall$. Informally speaking, an object belongs to $\forall$`left.cylinder` if it has no attribute filler or a `cylinder` as attribute filler for `left`.

$$\begin{aligned}
\texttt{ascasc} &=_{\text{conc}} \texttt{biconic} \sqcap \forall\texttt{left.asc-tc} \sqcap \forall\texttt{right.asc-tc}.\\
\texttt{hill} &=_{\text{conc}} \texttt{biconic} \sqcap \forall\texttt{left.asc-tc} \sqcap \forall\texttt{right.desc-tc}.\\
\texttt{rshoulder} &=_{\text{conc}} \texttt{biconic} \sqcap \forall\texttt{left.cylinder} \sqcap \forall\texttt{right.asc-ring}.\\
\texttt{lshoulder} &=_{\text{conc}} \texttt{biconic} \sqcap \forall\texttt{right.cylinder} \sqcap \forall\texttt{left.desc-ring}.\\
\texttt{shoulder} &=_{\text{conc}} \texttt{lshoulder} \sqcup \texttt{rshoulder}.
\end{aligned}$$

The next concept shows how two shoulders can be combined to a `groove`.

$$\texttt{groove} =_{\text{conc}} \exists\texttt{left.lshoulder} \sqcap \exists\texttt{right.rshoulder} \sqcap (\texttt{left} \downarrow \texttt{right}).$$

The *concept classification* service arranges the concepts as shown in Figure 3.

To represent a particular lathe work in a terminological system, the assertional formalism, called ABox, is employed. It allows to instantiate the concepts with instances and to fill in their attributes. A single truncated cone could for example be represented as:

$$\texttt{truncone}(\texttt{tc}_1).\quad \texttt{c}_1(\texttt{tc}_1, 0).\quad \texttt{r}_1(\texttt{tc}_1, 10).\quad \texttt{c}_2(\texttt{tc}_1, 5).\quad \texttt{r}_2(\texttt{tc}_1, 10).$$

Formally, numbers are not allowed in an ABox. But, we can replace a number, say 10, by a fresh object name, a, and add an assertion $p_{10}(a)$ to the ABox, where p_{10} is a unary predicate from the concrete domain with the extension $\{10\}$. The *realization* service of the ABox computes the set {`cylinder`} as the set of most specific concepts $\texttt{tc}_1$ belongs to.

3.3 Characteristics

What distinguishes terminological formalisms in the tradition of KL-ONE such as TAXON from the other formalisms of COLAB? Firstly, these formalisms should be decidable, secondly, they provide specialized reasoning services such as the prominent classification service.

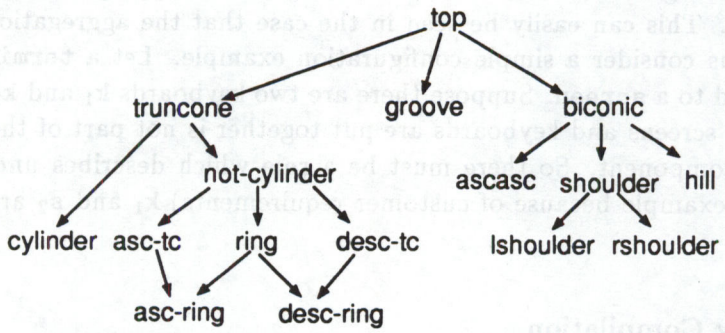

Figure 3: The Subsumption Graph of the Sample Terminology

3.3.1 Decidability

Terminological formalism focus unary (concepts) and binary predicates (roles), and, furthermore, the structure of the formulas in which these predicates may occur is rather restricted. As an achievement of the particular restrictions in TAXON the reasoning problems associated with the inference services are decidable and the formalism is still expressive enough to serve the needs of realistic applications. In particular, TAXON enhances the expressive power of conventional terminological logics by adding a facility to "ground" concept definitions: concrete domains.

Note, that terminological knowledge is represented independent of its intended use, there is no operational semantics which has to be considered by a knowledge engineer. In particular, there is no notion of left-to-right, top-down, or bottom-up evaluation of a KB or query as it is common with rule formalisms.

These advantages are complemented with some limitations w.r.t. expressive power. For example, in general it is not possible to deal with concrete domains (e.g., rational numbers) varying size aspects (e.g., sequences) in *one* concept language in a reasonable way, without having an undecidable subsumption problem [5]. Consequently, TAXON neither provides a transitive closure operator nor the ability to have cyclic terminologies.

Another principal limitation is related to assertional reasoning in TAXON. For example, we consider an additional truncated cone $\mathtt{tc}_2$ that neighbours the cylinder $\mathtt{tc}_1$ of the previous section:

$$\mathtt{truncone}(\mathtt{tc}_2).\quad \mathtt{c}_1(\mathtt{tc}_2, 5).\quad \mathtt{r}_1(\mathtt{tc}_2, 10).\quad \mathtt{c}_2(\mathtt{tc}_2, 5).\quad \mathtt{r}_2(\mathtt{tc}_2, 15).$$

The realization service would derive that $\mathtt{tc}_2$ is an ascending ring. But it **cannot** detect that they both form a 'biconic'—unless $\mathtt{tc}_1$ and $\mathtt{tc}_2$ are aggregated to a **single** instance. Once there is an object `bi` with assertions

$$\mathtt{left}(\mathtt{bi}, \mathtt{tc}_1).\quad \mathtt{right}(\mathtt{bi}, \mathtt{tc}_2).$$

`bi` can be classified as an **rshoulder**.

But this generation of a new instance is not a standard operation in terminological reasoning systems. The selection of instances that are composed to a new object does not depend on

terminological knowledge. On the contrary, knowledge about aggregation of instances is part of the assertional box. This can easily be seen in the case that the aggregation is not unique. To illustrate this, let us consider a simple configuration example. Let a `terminal` be defined as a `keyboard` connected to a `screen`. Suppose there are two keyboards $\mathtt{k}_1$ and $\mathtt{k}_2$ and two screens $\mathtt{s}_1$ and $\mathtt{s}_2$. If and how screens and keyboards are put together is not part of the terminological but of the assertional component. So there must be a rule which describes under which particular circumstances (for example because of customer requirements) $\mathtt{k}_1$ and $\mathtt{s}_2$ are connected to form a terminal $\mathtt{t}_1$.

3.3.2 Taxonomy Compilation

Probably, classification is the most prominent reasoning service of terminological knowledge representation systems. We shall use the notions of horizontal and vertical compilation to cast different perspectives on this reasoning service.

Vertical Compilation: This service analyses the terminology, extracts the inherent taxonomic knowledge (i.e., the subsumption relation), and stores it in a special data structure supporting efficient retrieval of the sub/super-concept relation.

Internally, concept terms are annotated to speed up later inferences: For example, in the current implementation the restrictor of a value-restriction or an exists-in restriction is classified. This information is used to efficiently detect redundancies and contradictions by looking up the precomputed subsumption relation.

Horizontal Compilation: Classification transforms items of one high-level formalism into another high-level structure, i.e., the subsumption relation. Both structures are used to communicate with human users.

The constraint formalism of CoLab uses concepts to structure the domain of the constraint predicates. At runtime the computed subsumption graph speeds up the inferences in cases where reasoning on type/concept level can be used to avoid reasoning with large numbers of tuples of instances. In [61] it is exemplified how reasoning with a (more general) sorted logic can be done more efficiently than without sorts.

Making the subsumption relation explicit and showing it to the knowledge engineer can also be seen as a knowledge-evolution service. It helps to detect unwanted or surprising consequences of the current concept definitions. For example, in case of an unwanted subsumption relation the engineer can then refine the respective knowledge items.

The classification precomputes the subsumption relation. Similarly, the other services (pre)compute member-ship relations or check for disjointness, consistency, satisfyablity etc. Each of these services may aid analyzing and compiling hybrid KBs refering to concepts, roles etc.

Summary of Characteristics

The decidability of the terminological logic enables sound and complete, terminating algorithms to analyze the terminology as well as the assertional knowledge. The intuitive character of subsumption as such and the precise, model-theoretic semantics enables simple, easy to understand

interfaces of the services, in particular the classification. This is an important prerequisite of natural integrations of terminological reasoning with other knowledge representation paradigms.

Two possible integrations are described in Section 4 and Section 6.2. The latter can handle varying size aspects and aggregation in the affirmative part of CoLab, which contains the ABox. Another interesting way is constraint resolution. See [3] for a survey of related approaches.

4 Constraint Propagation

Many AI problems can be formulated as constraint satisfaction problems, starting with the Waltz-Algorithm for arc-labeling and getting semantic descriptions of polyhedral scenes [62]. Since then many applications have been developed and many AI problems have been formulated as constraint satisfaction problems. Constraint satisfaction mechanisms are now widely used in expert systems [27], in planning systems [23], or in logic programming [11]. Various approaches and algorithms have been developed to tackle the constraint satisfaction problem (CSP). The computational complexities of these algorithms heavily depend on the level of consistency they compute (cf. [43]). To reduce the complexity, terminological knowledge can be used to structure the domains of the variables occurring in the CSP.

The constraint component Contax [47] available in CoLab supports constraint propagation methods to compute locally or globally consistent assignments of values from the given domains to the variables of the CSP. Especially, Contax provides a mechanism for solving constraints over hierarchically structured domains defined e.g. using Taxon (see Section 3).

4.1 Constraint Satisfaction and Local Consistency

Given a set of n variables, each with an associated domain and a set of constraining relations each involving a subset of the variables, a constraint satisfaction problem can informally be defined as to find all possible n-tuples such that each n-tuple is an instantiation of the n variables satisfying the relations. The constraining relations are called *constraints*. Constraints may be of any arity, whereas many constraint systems restrict them to be unary or binary. The variables of the CSP together with the constraints defined over them can be regarded as a constraint graph or *constraint net*. More formally, the general constraint satisfaction problem can be defined as follows:

Definition 4.1 (CSP) *Assume a finite set of variables $V = \{X_1, \ldots, X_n\}$, a finite set $D = D_1 \cup \ldots \cup D_n$ (domains), and a finite set R of relations R_i are given, such that $R_i \subseteq D_{i_1} \times \ldots \times D_{i_{k_i}}$, where $D_{i_j} \in \{D_1, \ldots, D_n\}$ and k_i is the arity of R_i. The constraint satisfaction problem is to find an assignment $\sigma : V \rightarrow D$ for the variables such that all constraints are satisfied simultaneously.*

A common example of a constraint satisfaction problem is the graph-coloring problem. Since graph-coloring is NP-complete, it is most unlikely that a polynomial time algorithm exists for solving general CSPs. However, a number of algorithms based on local propagation have been developed. These algorithms do not necessarily solve a CSP completely but they eliminate, once and for all, local inconsistencies that cannot participate in any global solution. These inconsistencies would otherwise have been repeatedly discovered by any backtracking solution. Hence local consistency algorithms can play the role of a preprocessor for subsequent backtracking

search, or they can be coupled with case analysis or simple domain splitting to recover the complete set of solutions to the CSP.

Constraint satisfaction algorithms can be classified by the level of consistency they establish between the variables of the constraint net. A k-consistency algorithm removes all inconsistencies involving all subsets of size k of the n variables. For example, the *node*, *arc*, and *path* consistency algorithms detect and eliminate inconsistencies involving $k = 1$, 2, and 3 variables, respectively. Freuder's generalization of those algorithms for $k \rightarrow n$ can be used to produce the complete set of solutions to the CSP [28].

Local propagation computes arc- or path-consistency. Values not appearing in any solution are eliminated. Local consistency gives an assignment of sets of values to the variables. Since the constraints are not satisfied simultaneously by the same values, this relaxes the problem. Every globally consistent solution is locally consistent too, but not vice-versa. By that relaxation the complexity of algorithms is reduced to polynomial time. Thus, local propagation can be used efficiently in large search spaces to pre-process and improve the behavior of backtracking algorithms, which run over the reduced search space afterwards. Nevertheless, the pruning-effect by local propagation depends on the kind of the problem: local propagation does not necessarily reduce the search space at all. On the other hand, some special instances of the CSP can be solved without any backtracking, provided there is some additional information about the structure of the constraint net [45].

4.2 Hierarchically Structured Domains and Hierarchical Arc-Consistency

In many real world applications, objects can be clustered and grouped to classes according to some of their properties. These classes often form a hierarchy, which can be described by the cover graph of the subclass-superclass relation. Knowledge representation using such taxonomic hierarchies enables efficient use of attributes and properties of the considered domains. The transitivity of the subclass-superclass-relation guarantees the inheritance of properties from super- to subclasses; subclasses can be seen as specializations.

Any arbitrary domain can be transformed into a directed acyclic graph (*dag*), which describes the domain as a heterarchy—in the worst case only consisting of nodes. In principle, the algorithms for solving any CSP, especially those dealing with large domains and hence large search spaces, can make use of structuring the domains.

The domains and their hierarchical structure can be defined directly by enumerating the subsumption links between classes. Moreover, the classes may declaratively be defined in terms of concept definitions in the sense of terminological languages like KL-ONE, which then are classified to get the subsumption graph. For this purpose, CONTAX employs the terminological component TAXON (see Section 3) and its classification algorithms to structure the domain. The classifier does not build a tree, usually, since one concept may be a subconcept of more than one other. More likely, the resulting graph describing the domain of some constraint variables becomes a dag representing a lattice.

To exploit the hierarchical structure of domains, the propagation algorithms had to be extended to deal with concepts instead of elements of a domain. The main aim is to reduce the complexity measured by the number of evaluations of the constraining relations. Therefore,

CONTAX provides an extended and improved version of the hierarchical arc-consistency algorithm (HAC) presented in [42]. It uses two new predicates, which evaluate the constraints between arbitrary concepts by using inheritance mechanisms. Since the concepts represent a large number of elements at once, this improves the efficiency of the propagation algorithm.

In [42] some assumptions about the constraints and hierarchies are made. The HAC algorithm only deals with binary constraints over binary, single-rooted, strict trees as domains. For any real-world CSP the restrictions made by HAC seem to be inappropriate. In addition to that, it is useful—especially for large domains—to allow definitions of constraints between arbitrary concepts. Since the hierarchies are seen as structured inheritance networks, we have to make clear what inheritance means for constraints:

Definition 4.2 (inheritance of constraints) *Let $R_j \subseteq D_1 \times \ldots \times D_i \times \ldots \times D_k$ be a constraint, $(d_1, \ldots, d_i, \ldots, d_k) \in R_j$ be a tuple in R_j. Then for all $\tilde{d}_i \in D_i$ the tuple $(d_1, \ldots, \tilde{d}_i, \ldots, d_k) \in R_j$ iff d_i subsumes $\tilde{d}_i$.*

A locally consistent value assignment can be defined in terms of hierarchical arc-consistency. For simplicity reasons, we only give the definition for binary constraints here. However, the actual CONTAX implementation uses an extended notion dealing with n-ary constraints:

Definition 4.3 (arc-consistency) *A value assignment $\sigma : V \rightarrow 2^D$ of a set of values to each variable of the constraint net with $\sigma(X_i) = \tilde{D}_i \subseteq D_i$ is* **arc-consistent** *iff for all variable pairs $(\tilde{X}_i, \tilde{X}_j)$ and for all constraints R_{ij} defined over them it holds that for each $d \in \tilde{D}_i$ there exists at least one $\tilde{d} \in \tilde{D}_j$ such that the pair $(d, \tilde{d})$ satisfies the constraint R_{ij}, that is $R_{ij}(d, \tilde{d})$ holds.*

The image $\tilde{D}$ of the value assignment σ only includes the most universal concepts that establish arc-consistency.

Hierarchical arc-consistency can now be defined based on the inheritance of constraints through subsumption links in the cover graph of the domains:

Definition 4.4 (hierarchical arc-consistency) *A value assignment $\sigma : V \rightarrow 2^D$ with $\sigma(X_i) = \tilde{D}_i \subseteq D_i$ is* **hierarchically arc-consistent**, *if it is arc-consistent and most universal, i.e. for all $d \in \tilde{D}_i$ there does not exist a more general concept $q \in D_i$ subsuming d such that the assignment*

$$\tilde{\sigma}(X_k) = \begin{cases} (\tilde{D}_k \setminus d) \cup \{q\} & if \quad k = i \\ \sigma(X_k) & if \quad k \neq i \end{cases}$$

is arc-consistent, too.

The hierarchical constraint satisfaction problem (HCSP) then is to compute a hierarchical arc-consistent value assignment which can then be further restricted using backtracking towards a globally consistent value assignment satisfying all constraints simultaneously.

4.3 Using the Constraint Component

Using CONTAX to formalize and solve a constraint satisfaction problem involves the following principal steps:

- Identify the variables and constraints that constitute the given problem and define the domains over which the variables range,
- define the problem constraints and connect the variables and constraints to build the constraint net, and
- finally, propagate some initial value assignments through the constraint net to restrict the domains of the variables and to achieve a solution for the underlying HCSP.

4.3.1 Defining Domains

In its simplest form, plain domains can be defined by simply enumerating all the elements belonging to the domain. For example, the knowledge item

```
alloy-steel = {low-alloy-steel, high-alloy-steel}.
```

introduces a new domain of some workpiece materials. Using the hierarchical structure of the domain, knowledge items like

```
steel = {building-steel, alloy-steel, stainless-steel}.
material = {steel, cast, alu}.
```

define the hierarchical domains **steel** and **material** to be the unions of some more specialized domains which have been defined before as plain or even hierarchical domains.

If some considered domain relates to a terminology defined using TAXON, the terminology along with all its concepts will automatically be imported and used by CONTAX. The classified concepts (i.e., the subsumption dag) can directly be used as the domain hierarchy for CONTAX, where the TAXON ABOX instances constitute the leaves.

4.3.2 Defining Constraints

CONTAX provides different types of constraints: primitive (or extensional), predicative and compound constraints. All constraint types may be defined over any number of variables.

Primitive constraints are defined by enumerating all the tuples satisfying the constraint. This kind of constraint can also be regarded as a *database constraint*. One step towards a more comfortable definition of constraints is to make use of non-leaf concepts when enumerating the relations. Consider, for example, the following constraint defining compatibility between workpiece-material and cutting-plates:

```
compatible(Material:material,Plate:plate) :-
    {(cast, cnmm),... (alloy-steel, dnmm-41),... (steel, dnmm-71),... }.
```

Here the fact that all kinds of **steel** are compatible with the **dnmm-71** plate is expressed by simply including the 'abstract' tuple **(steel dnmm-71)** instead of all the tuples for different kinds of **steel**. For the use of the hierarchical arc-consistency algorithm it is necessary that for each argument position the domain is specified from which the values in that position may come from.

Some constraints occurring in a real-world application are difficult or even impossible to be explicitly enumerated as primitive constraints. This is true, for example, for numerical constraints which should be evaluated by the underlying LISP system. Therefore, constraints can also be defined by providing a LISP function or **lambda**-expression which then will be evaluated to test a given tuple for membership in the relation. Consider, for example, the <_180 constraint in the μCAD2NC-II application:[2]

```
<_180(TCEA:angles, EA:angles, Alpha:angles) :-
   lambda(TCEA, EA, Alpha . (180 > TCEA + EA + Alpha)).
```

Often it may happen that copies of the same constraint subnet occur between different variables of the CSP. Therefore, it becomes very useful to define this subnet as a compound constraint which itself represents an entire constraint net. In contrast to primitive or predicative constraints, for compound constraints no domains have to be specified with the arguments; they can be computed from the constraints in the body. *Local variables* of the constraint subnet that only serve to connect local constraints need not to occur in the argument list. For example, the constraint net used in the μCAD2NC case study (Section 7) has been defined as a single compound constraint named `tool_sel`. The local variable `Edge_Angle` is determined by the variable `Plate` and therefore need not to be visible from outside the `tool_sel` constraint:

```
tool_sel(Holder,Tool,Plate,Process,Direction,Cutting,Material,
         Alpha,TC-Edge-Angle) :-
   {holder-tool(Holder,Tool),
    process-holder(Process,Holder),
    holder-description(Holder,Direction,TC-Edge-Angle,Plate),
    holder-cutting(Holder,Cutting),
    process-material-tool(Process,Material,Tool),
    plate-eangle(Plate,Edge-Angle),
    process-eangle(Process,Edge-Angle),
    tc-ea-al(TC-Edge-Angle,Edge-Angle,Alpha)}.
```

Although this looks very similar to a Horn clause, as in RELFUN (see Section 5) and the rule component (see Section 6), the brackets '{' and '}' mark it as the definition of an entire constraint net, and propagating this constraint net, e.g., by entering a goal

```
?- sol_of(tool_sel[Holder, Plate, roughing, cast, 80, TC-EA, EA]).
```

will result in a set of tuples all satisfying the constraints instead of only one (the first) solution as computed e.g. by RELFUN for a pure relational formulation of the tool-selection problem.

4.3.3 Computing a Hierarchically Arc-Consistent Value Assignment

After having defined all variables, constraints, and their connections forming a constraint net, CONTAX is ready to perform its real job, namely to propagate value restrictions through the

[2] Although the variables `Alpha`, `Edge-angle`, and `TC-Edge-Angle` range over finite discrete domains and it therefore would be possible to explicitly enumerate all tuples satisfying the <_180 constraint, in practice it is much more comfortable and even more efficiently computable to define this constraint as a predicative constraint using the underlying LISP system.

constraint net in order to compute a hierarchically arc-consistent value assignment. The basic idea of the local constraint propagation algorithm is the following:

1. All constraints are pushed onto a queue Q of constraints that have to be revised, that is, checked for hierarchical arc-consistency.

2. A constraint $C(X_1, \ldots, X_n) \in Q$ is selected to get revised and is deleted from Q. The domains of the variables $X_1, \ldots, X_n$ are then checked for hierarchical arc-consistency w.r.t. C.

3. **If** the domain of some variable becomes empty, an inconsistency has been detected and the propagation results in a failure.
   Otherwise, if the domain of some variable X has been restricted due to the application of some constraint, all other constraints $C_1, \ldots, C_m$ connected with X have to be revised again: $Q \leftarrow Q \cup \{C_1, \ldots, C_m\}$

4. **If** the constraint queue Q is not empty, the process continues with step 2.
   Otherwise, the current value assignment is hierarchically arc-consistent and is returned as the result of the local propagation procedure.

Step 2 contains the very heart of the constraint propagation algorithm, namely, how to select the next constraint from Q to revise. Here, a set of heuristics is used, for example, to prefer the constraint with the maximally restricted variable domains.

If the local propagation succeeds, the resulting value assignment can further be checked for global solutions by making choices (selecting values from the restricted domains) and using backtracking to enumerate all or any required number of solutions.

4.4 Constraint Compilation

The CONTAX component is implemented in an object-oriented fashion based on the Common Lisp Object System (CLOS). Therefore, all constraints and variables are compiled into CLOS objects. For each primitive or predicative constraint a CLOS object is created that represents the constraint. Additionally, for each argument to the constraint one variable object is created and linked to the constraint. When compiling compound constraints, variable objects are created for all variables occurring in the constraint definition including local variables. Then the body constraints that make up the compound constraint are linked to the variable objects according to the constraint definition.

This vertical compilation process also involves some optimizations that result in a more efficient propagation. For example, primitive constraints that are called with the same variable in different argument positions can be *folded*, that is, a copy of the constraint will be created which definition only contains those tuples for which the values at the considered argument positions have a non-empty intersection. Moreover, these values are replaced by their least upper bound within the cover graph representing their domain.

Knowledge represented as constraints can also be horizontally compiled into Horn clauses. This horizontal compilation process consists of the following steps:

- Since the relational-functional component RELFUN does not currently support sorted Horn clauses, the main task is to represent the domain structure as a collection of unary predicates: Each definition of the form

  $dom = \{el_1, \ldots, el_n\}.$

  is compiled into a set of n clauses, one for each el_i, of the following form:

  dom`(X) :-` el_i`(X).`

  Additionally, for each leaf el_i occurring in the domain definitions a fact of the form

  $el_i(el_i)$.

  has to be generated.[3]

  By applying this scheme the domain definitions from the previous subsection 4.3 are compiled into:

```
alloy-steel(X) :- low-alloy-steel(X).
alloy-steel(X) :- high-alloy-steel(X).
low-alloy-steel(low-alloy-steel).
high-alloy-steel(high-alloy-steel).
...
steel(X) :- building-steel(X).
steel(X) :- alloy-steel(X).
steel(X) :- stainless-steel(X).
...
material(X) :- steel(X).
material(X) :- cast(X).
material(X) :- alu(X).
```

  If some domains are represented using TAXON, the same compilation process has to be performed for each element of the cover graph that will be computed by the TAXON classifier for the considered domain.

- A primitive constraint of the form

  $p(A_1 : dom_1, \ldots, A_n : dom_1)$:-
  $\{ (X_{1,1}, \ldots, X_{1,n}), \ldots, (X_{i,1}, \ldots, X_{i,n}), \ldots, (X_{m,1}, \ldots, X_{m,n}) \}.$

  is compiled into a set of m clauses of the form

  p`(X1,...,Xn) :-` $X_{i,1}$`(X1),...,`$X_{i,n}$`(Xn).`

  one for each tuple $(X_{i,1}, \ldots, X_{i,n})$.[4]

  By applying this scheme the primitive constraint from the previous subsection 4.3 is compiled into:

[3] In the cases where el_i represents a leaf, we could also generate just a fact of the form $dom(el_i)$ instead of generating a clause dom`(X) :-` el_i`(X)` together with the el_i self-application $el_i(el_i)$.

[4] In the cases where $X_{i,j}$ represents a leaf, the corresponding argument `Xj` can be replaced by $X_{i,j}$ and the premise $X_{i,j}$`(Xj)` can be deleted.

```
compatible(X1,X2) :- cast(X1), cnmm(X2).
...
compatible(X1,X2) :- alloy-steel(X1), dnmm-41(X2).
...
compatible(X1,X2) :- steel(X1), dnmm-71(X2).
```

- A compound constraint of the form

  $p(A_1,\ldots,A_n)$:- { $C_1,\ldots,C_m$ }.

  is then simply compiled into a Horn clause of the following form:

  $p(A_1,\ldots,A_n)$:- $C_1,\ldots,C_m$.

  By applying this scheme the compound constraint from the previous subsection 4.3 is compiled into:

```
tool_sel(Holder,Tool,Plate,Process,Direction,Cutting,Material,
         Alpha,TC-Edge-Angle) :-
   holder-tool(Holder,Tool),
   process-holder(Process,Holder),
   ...
   process-eangle(Process,Edge-Angle),
   tc-ea-al(TC-Edge-Angle,Edge-Angle,Alpha).
```

- Instead of propagating initial value assignments through a constraint net by using the `sol_of` built-in, the `tupof` built-in of RELFUN is used to compute the set of all solutions for a given constraint. That is, each goal of the form `sol_of(`*Constraint*`)` is compiled into the goal `tupof(`*Constraint*`)`.

  By applying this scheme the sample call from the previous subsection 4.3 is compiled into:

```
?- tupof(tool_sel(Holder, Plate, roughing, cast, 80, TC-EA, EA)).
```

While the horizontal compilation of constraints into Horn clauses works for the whole constraint language, that is, it can be seen as a total mapping from the constraint component into the relational component, the same is not true for the compilation of Horn clauses into constraints. This compilation direction is restricted to the subclass of (non-recursive) DATALOG programs. Checking for a given set of Horn clauses whether this restriction is fulfilled is a non-trivial task which requires a lot of dependency analysis whereas the transformation itself can be done in a simple syntactic way. Moreover, since the hierarchical constraint satisfaction algorithm heavily depends on explicit knowledge about the structure of the domains but CoLab users currently cannot write order-sorted Horn clauses, for optimally exploiting the HAC algorithm the domains and their structure (taxonomy) would have to be extracted from the unsorted Horn clauses, too.

5 Relational-Functional Computation

Logic or relational programming in PROLOG is now being employed in applications approaching the size of ordinary databases [50]. Functional or applicative programming in LISP and more pure but still efficient languages such as ML has served as a practical basis for symbolic algorithms [55]. Combining both programming styles can open new applications with inseparable database-like and algorithmic operations. RELFUN ([13], [14]) is a logic-programming language with call-by-value (eager), non-deterministic, non-ground functions, and higher-order operations. As part of COLAB it is coupled with the terminological-reasoning, constraint-handling, and forward-rule components.

In the following subsection we briefly introduce RELFUN (5.1). Besides its attempt at integrating basic notions of PROLOG and LISP, many of RELFUN's extended concepts can also be transferred to relational and functional programming individually. The next subsection (5.2) treats the extended relational subformalism, including higher-order relations. The subsequent subsection (5.3) will then augment this by the extended functional subformalism and discuss its benefits. The last subsection (5.4) will illustrate RELFUN's (WAM) compiler.

5.1 A Brief Introduction to RELFUN

Many approaches are possible for combining logic and functional programming, as illustrated by the collection [22]. RELFUN's integrating concept is valued clauses, encompassing both PROLOG-style Horn clauses (for defining relations) and directed conditional equations (for defining functions). While the former start off from Horn logic, the idea for the latter is to regard a function definition as a system of clauses each matching (in general, unifying) argument configurations and **returning** corresponding values. Thus, the binary *max*imum function, based on the 'built-in' relations "<", ">", and "=",

$$max(x,y) = \begin{cases} y & if \quad x < y \\ x & if \quad x = y \\ x & if \quad x > y \end{cases}$$

will not be construed as clauses of a logic with 'user-defined' equality (shown on the left) but as clauses that return the right-hand sides of the directed equations via a ("&"-marked) premise following after zero or more other premises (shown on the right):

```
eq(max(X,Y),Y) :- X < Y.          max(X,Y) :- X < Y & Y.
eq(max(X,X),X).                   max(X,X) :- & X.
eq(max(X,Y),X) :- X > Y.          max(X,Y) :- X > Y & X.
```

This means that function calls need not be embedded into `eq` calls with auxiliary request variables, as in `eq(max(2,7),MaxA), eq(max(9,5),MaxB), MaxA < MaxB`, but can be written directly, as in `max(2,7) < max(9,5)`. We then interpret value-returning premises (after the ampersand) as generalized Horn-rule premises: apart from being terms like `Y` they may be calls like `*(-1,X)` or `member(X,[-1,-3,-5])` and nestings like `+(*(-1,X),3)` or `member(X,rest([-1,-3,-5]))`. Nestings are evaluated strictly call-by-value, as, classically, in, e.g., FP [6].

The RELFUN notions of *relation* and *function* are amalgamated to an abstract *operator* concept: functions are generalized to non-ground, non-deterministic operators, hence relations can be viewed as characteristic functions. Our notion of relations as `true`-valued functions is like in SLOG [29], except that RELFUN's valued facts return `true` implicitly. Another amalgamating notion is akin to LISP's "useful non-`nil` values": relation-like operators may on success return a value more informative than `true` (e.g., we can let `member` return the list starting from the element found). All kinds of RELFUN operators can be applied in generalized Horn-rule premises, which are usable uniformly to the left as well as to the right of the "`&`"-separator. Actually, such premises constitute a *valued conjunction*, also permitted as a top-level query (e.g., `member(X,L) & member(X,M)` non-deterministically returns rest lists of `M` whose first element also occurs in `L`). A special valued conjunction calling only relations to the left of "`&`" and having a single variable to its right (e.g., `member(X,[-1,2,-3,4,-5]), <(X,0) & X`) can be viewed as an indefinite description or η-expression (e.g., $\eta(x)[x \in \{-1,2,-3,4,-5\} \wedge x < 0]$), also provided in other relational/functional amalgamations (see [53]). It will be shown that certain RELFUN functions can be inverted by calling them non-ground (by-value) on the right-hand side (rhs) of a generalized PROLOG `is`-primitive, mimicking relations (incl. the above `eq` predicate).

5.2 Relations Defined by Hornish Clauses

5.2.1 Open-World DATALOG

First we consider DATALOG i.e., PROLOG without structures (constructor symbols applied to arguments). This kernel language of deductive databases is also a subset of RELFUN. DATALOG clauses have identical syntax and equivalent semantics in PROLOG and RELFUN. Queries to RELFUN differ only as follows: they **return** the truth-value `true` instead of **printing** the answer `yes`; they signal failure by yielding the truth-value `unknown` instead of printing `no`.

When we stay in the relational realm of RELFUN this makes not much of a difference since `true` can be mapped to `yes` and `unknown` can be mapped to `no`. However, when proceeding to RELFUN's functional realm, queries will be able to return the third truth-value `false`: this is to be mapped to those of PROLOG's `no` answers for which the closed-world assumption is justified. In general, however, RELFUN does not make the closed-world assumption, and in the absence of explicit negative information modestly yields `unknown` instead of 'omnisciently' answering `no`.

For example, given the ('object-centered', TBOX-simulating) DATALOG knowledge base

```
subsumes(biconic,shoulder).    % a shoulder is made of two cones
subsumes(shoulder,rshoulder).  % a right shoulder is a kind of it
left(biconic,truncone).    % the left part of a biconic is a truncated cone
right(biconic,truncone).   % the right part of a biconic has the same type
left(rshoulder,cylinder).  % etc.
right(rshoulder,ring).                  % subsumes inheritance via rules:
left(Biconic,Truncone)  :- subsumes(Super,Biconic), left(Super,Truncone).
right(Biconic,Truncone) :- subsumes(Super,Biconic), right(Super,Truncone).
```

a successful query like `left(rshoulder,truncone)` returns `true` in RELFUN and prints `yes` in PROLOG; however, a failing query like `left(lshoulder,truncone)` yields `unknown` in RELFUN

but prints no in PROLOG. As with most real-life knowledge, what we know about production-relevant geometries is inherently open-ended; RELFUN's **unknown** reply agrees to the required open-world semantics.

Assuming the Herbrand universe of the above KB only contains the constants occurring in its facts, it could be horizontally compiled into a constraint system such as the one of Section 4 via the generation of all ground instances of its rules and the elimination of the `left` and `right` recursions by unfolding them to simple fact accesses. The above facts encode taxonomic knowledge (including 'functional' roles), while the rules are a dynamic special-purpose analogue to static inheritance in KL-ONE-like classifiers; the usual way for representing this knowledge is shown in Section 3. If our TBOX constants (e.g. `rshoulder` and `left`) are themselves applied as (unary and binary) relations in the ABOX, the above TBOX knowledge has a second-order characteristics (cf. Subsection 5.2.4); for first-order (bidirectional) rules assertionally involving `rshoulder` etc. see Section 6.

Later, in DATAFUN, certain relations such as **subsumes** will be reformulated as functions such as **subsumer** (cf. end of Subsection 5.3.1). This allows to reformulate Horn rules such as the `left` rule into rules which still define a relation but call a sub**function** embedded in a relation call: `left(Biconic,Truncone) :- left(subsumer(Biconic),Truncone).` To accommodate such functional (and is-'equational') extensions in relational rules, we speak of *hornish rules* or, generally, *hornish clauses.*

5.2.2 PROLOG-like Structures and Lists

Arguments to PROLOG relations must always be (passive) structures and can never be (active) calls. RELFUN, on the other hand, does support both of these categories, hence has a notational need to distinguish between them. We write round parentheses for 'active' operator calls and square brackets for 'passive' structured terms. N-element RELFUN lists, as in LISP and PROLOG, can be regarded as a short-hand for nested binary structures (we use the distinguished constructor "`cns`" instead of the usual "`.`"). For example, the list `[a,b,c]` reduces to the structure nesting `cns[a,cns[b,cns[c,nil]]]`. A vertical bar in lists causes their `cns`-reduction to end with the element after the "`|`" (usually a variable) rather than with the distinguished constant `nil`. Thus, `[a,b|Z]` reduces to `cns[a,cns[b,Z]]`.

5.2.3 Varying-Arity Structures and Relationships

Lists can be given a direct N-element interpretation because RELFUN permits *varying-arity structures* i.e., structures containing a vertical bar. We use `tup` as an N-ary list constructor ($N \geq 0$). That is, `[...]` should be regarded as an abbreviation for `tup[...]`. This convention holds even if `[...]` contains a "`|`".

Unlike PROLOG we permit the vertical bar to follow directly after an opening square bracket, both in lists and in (other) structures. For any list `X`, the list `[|X]` is the same as `X`; additionally given a constructor `c`, the structure `c[|X]` exclusively uses the elements of the list `X` as its arguments.

Proceeding from constructor terms to atomic formulas, we come to the LISP-inspired PRO-

LOG extension of *varying-arity relation applications* i.e., clause heads and bodies directly containing a "|". For example, using PROLOG's ternary list-concatenation relation **apprel**, we can define an N-ary append extension (N > 0), binding its **first** argument to the result:

```
append([]).
append(Total,Front|Back) :- apprel(Front,Inter,Total), append(Inter|Back).
```

Thus, both structures and applications can be ended by a vertical bar followed by an ordinary variable; equivalently, they could be ended by a "sequence variable" as used in KIF [30].

5.2.4 Higher-Order Relations

While PROLOG restricts constructors and relations to constants, RELFUN also permits them to be variables or structures. This enables a restricted kind of *higher-order operators*, syntactically reducible to first-order operators, but more expressive and cleaner than PROLOG's use of extralogical builtins like `functor`, "`=..`", and `metacall` as higher-order substitutes.

Relation variables in queries enable to find all relationships between given arguments. In the DATALOG KB (see Subsection 5.2.1) the query `Attribute(rshoulder,Filler)` needs only fact retrieval for the first two solutions binding `Attribute` to the relation `left` and `Filler` to the object `cylinder` or, `Attribute` to `right` and `Filler` to `ring`; the query `Attribute(shoulder,truncone)` requires rule deduction for binding `Attribute` to `left` or `right`. Such 'variable-attribute' queries are not conveniently expressible in usual object-centered formalisms such as KL-ONE.

Relation variables in clauses permit the use of higher-order facts (recognized as such by the context) like `cut-direction(to-left)` and `cut-direction(to-right)` to abstract rules like

```
turnable(X) :- lathe-tool(T), to-left(T,X).
turnable(X) :- lathe-tool(T), to-right(T,X).
```

to the single rule ("Turnable is that which can be cut in some direction by some lathe tool")

```
turnable(X) :- cut-direction(D), lathe-tool(T), D(T,X).
```

Here we apply `cut-direction` as a unary second-order relation over two binary relations, but more general higher-order relations can be useful.

5.3 Functions Defined by Footed Clauses

5.3.1 DATAFUN as a Functional Database Language

Let us consider a database example containing the following DATALOG facts about the areas of unit truncated cones (with radius and possible height equal to 1):

```
area(unit-cylinder,12.566370614359172).     % 4 * pi
area(unit-cone,     7.584475591748159).     % (1 + square-root(2)) * pi
area(unit-circle,   6.283185307179586).     % 2 * pi
```

Although these binary relations would permit requests like `area(Truncone,7.584475591748159)`, their normal use direction is of the kind `area(unit-cone,Area)`: to guarantee successful unification, a rounded real number is better used as an 'output' argument than as an 'input' argument. Indeed, in our opinion this DATALOG example should be rewritten functionally. For this we extract the second argument from the DATALOG facts and use it as the so-called *foot* after a ":-&"-infix (equivalent to ":- &"):

```
area(unit-cylinder) :-& 12.566370614359172.
area(unit-cone)     :-&  7.584475591748159.
area(unit-circle)   :-&  6.283185307179586.
```

The resulting special DATAFUN clauses are called *footed facts*, here used for the pointwise definition of the RELFUN function `area` mapping from truncated-cone symbols to real numbers. The definition emphasizes the natural `area` use direction, as in `area(unit-cone)`, a function call **returning** the value `7.584475591748159`.

The main advantage of distinguishing an output argument of a relation as the returned value of a corresponding function is the possibility of *nested calls* such as

```
+(area(unit-cylinder),area(unit-cone),area(unit-circle))
```

where the parenthesized inner applications are (not passive structures but) active function calls that return their values to the ternary `+` use (cf. Subsection 5.2.2); for reasons of conciseness, program analysis, and variable elimination this is preferable to flat relational conjunctions such as

```
area(unit-cylinder,A1), area(unit-cone,A2), area(unit-circle,A3),
                                              +(Area,A1,A2,A3)
```

The main disadvantage lies in the issue of *inverted calls*, which are easier and sometimes more logically complete for 'usage-neutral' relations. However, RELFUN's inversion method for functions appears quite natural, and for its DATAFUN subset completeness problems do not arise. A generalized form of PROLOG's `is`-primitive is employed to unify the values of a free function call with the value to be used as the argument of the inverse function, where a call is *free* if all its (actual!) arguments are different free variables.

As a simple example of an inversion with just one free variable consider `7.584475591748159 is area(Truncone)`, the inverse function call corresponding to the above-discussed relational inversion `area(Truncone,7.584475591748159)`. Independently from the context (e.g., in an `is`-rhs) the free call `area(Truncone)` non-deterministically returns the values `12.566370614359172`, `7.584475591748159`, or `6.283185307179586`, at the same time binding `Truncone` to `unit-cylinder`, `unit-cone`, or `unit-circle`, respectively, in the textual order of the `area` footed facts in the knowledge base. Within the above `is`-call only the second of the returned values unifies with the left-hand side, so the inversion correctly binds `Truncone` to `unit-cone`.

There are analogous DATALOG Horn facts about `volume`, which we think should be 'functionalized' to DATAFUN footed facts as demonstrated for `area`. On a relational basis, we could supply the ratio `volume-per-area` of a truncated cone, using the DATALOG rule

```
volume-per-area(Truncone,Vpa) :- volume(Truncone,V), area(Truncone,A),
                                                    Vpa is /(V,A).
```

This can be mimicked by the equivalent DATAFUN rule (with **is**-calls for **V** and **A**)

```
volume-per-area(Truncone)  :-  V is volume(Truncone), A is area(Truncone)  &
                                                    /(V,A).
```

which may be condensed to the DATAFUN rule (without **is**-calls or auxiliary variables)

```
volume-per-area(Truncone)  :-&  /(volume(Truncone),area(Truncone)).
```

Rules containing an "**&**" separator are called *footed rules*. The rule premises to the left of "**&**" are called *body premises* and act exactly like the premises of a hornish rule. The premise to the right of "**&**" is called a *foot premise* and differs from the other premises only in that its value becomes the value of the entire rule. The most natural use of the DATAFUN database would be functional calls like **volume-per-area(unit-cylinder)**, returning the ratio for the unit cylinder. However, these rule formulations could also be inverted or even be called freely to enumerate all pairs of unit truncated cones and their volume-per-area ratios as in the relational call **volume-per-area(Truncone,Vparat)** (delivering both informations as bindings) or the functional call **volume-per-area(Truncone)** (delivering the first information as a binding and the second one as a value).

While free **calls** for the inversion of the **area** and **volume-per-area** functions produce non-deterministic results, the **area** and **volume-per-area definitions** themselves are deterministic. In RELFUN *non-deterministic function definitions* are also allowed, which enumerate more than one value even for ground calls.

For instance, the **subsumes** relation of the DATALOG example in Subsection 5.2.1 could be extended and transcribed into a non-deterministic function **subsumer**, as in the following DATAFUN example:

```
subsumer(shoulder)  :-& concave.
subsumer(shoulder)  :-& biconic.
subsumer(lshoulder) :-& shoulder.
subsumer(rshoulder) :-& shoulder.
left(biconic)    :-& truncone.
right(biconic)   :-& truncone.
left(rshoulder)  :-& cylinder.
right(rshoulder) :-& ring.
left(Biconic)  :-& left(subsumer(Biconic)).
right(Biconic) :-& right(subsumer(Biconic)).
```

In this KB the ground call **subsumer(shoulder)** non-deterministically returns the values **concave** or **biconic**; finding a **subsumer** path from **lshoulder** to **biconic**, **left(lshoulder)** returns **truncone**. Note that, e.g., the former relation **left** became a function that nests the (non-deterministic!) **subsumer** function into the recursive **left** call. In μCAD2NC we will use a function **tool-select** which both builds up a list recursively and tests it non-deterministically (Section 7, Appendix B).

5.3.2 Full RelFun and Higher-Order Functions

When enriching DATAFUN with structures and lists we arrive at full RelFun (one can immediately transfer the relational varying-arity extensions). A simple but important definition permits the use of tup as a *self-passivating function*:

```
tup(|Z) :-& tup[|Z].      (or   tup(|Z) :-& [|Z].   or   tup(|Z) :-& Z.)
```

Now, tup may also be called actively, evaluating its arguments in the usual call-by-value manner and returning a passive list that uses the evaluated arguments as its elements.

Function variables in queries can be utilized much like the corresponding relation variables (see Subsection 5.2.4). For example, given the DATAFUN version of the `volume-per-area` database (see Subsection 5.3.1), the query `F(unit-circle)` asks for all unary properties of `unit-circle`, enumerating the attribute `F = area` with the returned value `6.283185307179586`, the attribute `F = volume` with its value, etc.

Function variables in clauses give us the abstraction power of functional arguments in the fashion of functional programming. Thus, `revise` is a ternary function applying any unary function `F` to the Nth element of a list (for N greater than the list length or N less than 1 it returns the list unchanged):

```
revise(F,N,[]) :-& [].
revise(F,1,[H|T]) :-& tup(F(H)|T).
revise(F,N,[H|T]) :-& tup(H|revise(F,1-(N),T)).
```

An example combining higher-order operators with non-ground, non-deterministic calls is F^{-1}, the inversion of a unary function `F`; it can be defined as a function structure `inv[F]` which calls `F` freely within an `is`-call only accepting `F` values that match the argument `V` of F^{-1}:

```
inv[F](V) :- V is F(X) & X.
```

Thus, `inv[subsumer](shoulder)` calls `shoulder is subsumer(X)`, hence non-deterministically returns `lshoulder` or `rshoulder` because for these arguments `subsumer` returns `shoulder`: `inv[subsumer]` is a (less efficient) substitute for a directly defined non-deterministic `subsumee` function.

5.4 Relational-Functional Compilation

We have developed a layered RelFun compiler system, called RFM, ranging from full-to-kernel language transformers (horizontal compilation) to a declarative classifier and a functionally extended optimizing WAM-code generator (vertical compilation), together with a minimally extended WAM emulator ([12], [15], [59]). There is also a (horizontal) translator to a relational subset of RelFun, henceforth to PROLOG. Referring to the above papers and their references, we only give simplified illustrations here, restricted to first-order DATAFUN.

For instance, a rule with a nested call-by-value premise such as

```
left(Biconic) :-& left(subsumer(Biconic)).
```

is first *flattened*, i.e. each embedded (call-by-value) expression is (recursively) replaced by a newly generated variable, which becomes bound to the expression via an additional `is`-premise:

```
left(Biconic) :- _1 is subsumer(Biconic) & left(_1).  % variable _1 is new
```

Full RelFun can be transformed to this nestingless subset, coming closer to both the extended WAM and to the relational subset.

For proceeding to relations (here, DATALOG), the flattend form is *extrarged*, i.e. an additional first argument is introduced for holding functional values as relational bindings:

```
left(_2,Biconic) :- subsumer(_1,Biconic), left(_2,_1).   % _2 is new
```

For proceeding towards the extended WAM, the flattened, still high-level, form is annotated, obtaining a *classified clause*, which contains emulator-specific information:

```
fun*eva[          % a functional (footed) clause with evaluative (non-term) foot
 perm[],          % clause-global annotations:     no permanent-variable info
 temp[[Biconic,[1,[1],[1]]],[_1,[1,[],[1]]]],     % two temporary-variable infos
 chunk[           % head chunk consisting of two user literals
  [usrlit[left[[Biconic,[first,safe,temp]]],[1,0,[]]],     % var-occurrence info
   usrlit[subsumer[[Biconic,[nonfirst,safe,temp]]],[1,0,[]]] ],     % is-rhs
  [1,[]] ],                                                % compiled
 chunk[                                                    % first
  [refl-xreg[[_1,[first,unsafe,temp]]],    % then returned value unified with X1
   usrlit[left[[_1,[nonfirst,unsafe,temp]]],[1,0,[]]] ],  % foot recursion sets X1
  [1,[]] ] ]
```

From this declarative basis we finally generate the following highly optimized and compacted *WAM instructions*:

```
allocate(0), call(subsumer/1,0), deallocate(), execute(left/1)
```

In the RFM emulator, value returning is performed via the temporary register `X1`. For unary functions like `left` this permits the optimization of an inner call directly returning its value to an outer call, as suggested by the source clause's nesting: no `put` instruction is needed for preparing `left/1`'s argument register (and the variable `_1` is eliminated, too) since `subsumer/1` already returns its value to `X1`.

6 Bottom-up Deduction

The rule component of CoLab is a declarative logic programming system. Bottom-up evaluation of logic programs implements the least fixpoint semantics. It is known to be complete, but it can be very inefficient if bindings for some argument positions are given in the query. A top-down strategy, however, would compute only the derivations necessary to answer the query. It applies a rule in backward direction by unifying a query with the conclusion of a clause.

Cooperation of various formalisms in a hybrid system makes additional demands on the evaluation and compilation of rules. In CoLab there are integrations of bottom-up evaluation with the top-down reasoning of RelFun (Section 5) and the taxonomic reasoning in Taxon (Section 3). Each of these pairwise integrations is described independently in more detail in [37] and [32], respectively. In this section a common framework from the viewpoint of bottom-up deduction will be given.

Based on the basic bottom-up and top-down evaluation procedures the rule component of CoLab offers two independent forward reasoning strategies: The first set-oriented approach interprets bottom-up rules directly using a fixpoint computation (Subsection 6.2). The premises are verified by look-up in the fact base. A generalized magic set transformation is implemented for goal-directed reasoning, which is complete and efficient (Subsection 6.3). The second, tuple-oriented scheme reasons forward to derive the consequences of an explicitly given set of initial facts. Rules are transformed to top-down evaluable RelFun Horn clauses (Subsection 6.4). The premises of triggered rules are tested by the backward reasoning proof procedure of RelFun. Both reasoning strategies are compiled into abstract machines (Subsection 6.5).

6.1 Hybrid Rules in CoLab

Horn clauses are the basic representation scheme of CoLab's rule component. Horn clauses are clauses with at most one positive literal. To allow more compact representations, in CoLab deduction rules can have multiple conjoined conclusions. For conformity with logic programs, the conclusion is written to the left of the antecedent:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge P_m$$

Multiple conclusions make bottom-up evaluation more efficient. For top-down evaluation these rules can easily be transformed into a sequence of Horn clauses with one conclusion by a simple horizontal compilation step. The conclusions $C_1, C_2, \ldots, C_n$ of such a *deduction rule* are literals which are true if the conjunction of premises $P_1, \ldots, P_m$ in the antecedent is satisfied – as opposed to production rules (cp. OPS5 [26]) where the conclusion consists of operations modifying the working memory.

One major idea of declarative programming is the separation of logic from control shifting the responsibility for control to the execution mechanism: the programmer should care as little as possible about it. For knowledge representation in a rule system this means, in the ideal case, that the application direction of a rule need not be visible to the knowledge engineer. On the other hand, he may have in mind a specific operation mode. Therefore in CoLab there are three types of relational rules, those which can be evaluated in both directions or those which can be evaluated in bottom-up or top-down fashion only. Bottom-up and bidirectional rules have to be range-restricted; this means that every variable in the conclusion has to be bound by a premise of the rule. We use different kinds of arrows to distinguish the rule types:

`<=` for bidirectional rules
`<-` for bottom-up rules
`:-` for top-down rules, which are equivalent to RelFun's hornish clauses.

A predicate p is said to be defined by rules, if it occurs in the conclusion of at least one rule. All the rules defining a predicate p must be of the same type. Depending on the type of rules defining a predicate p it is called either a *bottom-up predicate*, a *top-down predicate*, or a *bidirectional predicate*. The antecedent of a rule is a conjunction of expressions, each of which is of one of the following forms:

1. *Atomic assertion* $p(t_1, \ldots, t_n)$, where p is an n-ary predicate defined by rules and $t_1, \ldots, t_n$ are terms or expressions.
2. *Membership assertion* $C(t)$, where C is a concept and t is a term possibly containing variables.
3. *Role-filler assertion* $R(t, s)$, where R is a role or attribute and t, s are terms.
4. *Predicate assertion* $P(u_1(t_1), \cdots, u_n(t_n))$, where the u_i are possibly empty compositions of attributes, the t_i are terms, and P is an n-ary predicate from the concrete domain.
5. *Attribute-term assertion* $C(t, R_1[s_1], \cdots, R_n[s_n])$, i.e. an object-centered representation equivalent to the conjunction $C(t), R_1(t, s_1), \cdots, R_n(t, s_n)$ of membership and attribute-filler assertions for a single instance represented by the term t.
6. *built-in expression* for arithmetics, comparisons etc.

6.2 Bottom-up Evaluation of Hybrid Rules

Forward chaining – often called bottom-up evaluation – is a well-known strategy for evaluating logic programs. It implements the least fixpoint semantics and is known to be complete. Compared to the naive evaluation method the so-called semi-naive evaluation eliminates a lot of redundant derivations ([8], [9]). The objective of deriving at each iteration only new facts is approximated by an differential function [7]: a rule is applied in iteration i if at least one of its rule-defined premises is matched by a fact derived in iteration $i - 1$.

Algorithm 6.1 (Semi-naive Evaluation) *Let $\mathcal{F}$ be the set of all facts, $\mathcal{NDF}$ be the list of new facts derived in the current cycle, and $\mathcal{PDF}$ be the list of facts derived in the previous cycle.*

1. *Start with the initial facts $\mathcal{F}$, set $\mathcal{NDF} := \emptyset$ and $\mathcal{PDF} := \mathcal{F}$*
2. *For every clause, $H \leftarrow P_1, ..., P_m$ in $\mathcal{R}$ for which there is a substitution σ, such that at least one $P_i\sigma$ is in $\mathcal{PDF}$ and all $P_j\sigma$, $j \in \{1, \ldots, m\} \setminus i$, are constructively implied by the extended* ABox *$\mathcal{A} \cup \mathcal{F}$, set $\mathcal{NDF} := \mathcal{NDF} \cup \{H\sigma\}$*
3. *If $\mathcal{NDF} = \emptyset$, then stop, else set $\mathcal{PDF} := \mathcal{NDF} \setminus \mathcal{F}$, $\mathcal{F} := \mathcal{F} \cup \mathcal{PDF}$, and $\mathcal{NDF} := \emptyset$, goto 2*

In pure semi-naive evaluation the application of a rule in step 2 depends only on the facts in $\mathcal{F}$. In a hybrid system the satisfaction of the rule premises additionally depends on the facts in the ABox of the terminological component, the taxonomy and on the lemmas, which can be proved by backward rules. For this extended test the term *constructive implication* has been used:

Definition 6.2 (Constructive Implication) *Let $\mathcal{A}$ be an* ABox *and $\mathcal{F}$ be a set of facts. Let G, E be an expression, where G is a conjunction of membership assertions, role-filler assertions and predicate assertions and E is a conjunction of rule-defined premises. Then $\mathcal{A} \cup \mathcal{F}$* constructively implies *G, E by (the substitution) σ iff (i) $G\sigma$ is ground and (ii) $G\sigma$ is logically implied by $\mathcal{A}$ and the current terminology and (iii) each conjunct of $E\sigma$ is either in $\mathcal{F}$ or can be proved by the top-down proof procedure.*

The cooperation of the bottom-up reasoning component and TAXON combines the general-purpose reasoning power of rule-based systems with the inheritance abstraction provided by terminological systems. Additionally it extends the reasoning capabilities of TAXON. Because terminological systems provide decision procedures for their reasoning problems, it cannot be avoided that they have a restricted expressiveness. Therefore, for example, we deal with varying size (see Section 3.3) aspect by excluding them from the terminological formalism and deal with them in the rule language.

Until now concept and role predicates have only been allowed in the antecedent of rules. But for a restricted kind of rules, concept and role predicates are also allowed in rule conclusions. *Aggregation rules* collect objects or values to form a *new* object if certain conditions hold for the constituent parts. For example, the terminological component alone cannot detect that a cylinder and a neighbouring ascending truncated cone form a shoulder, unless they are aggregated into a single instance (cf. Section 3.3).

For a more detailed description of aggregation rules, the derivation of role values and the specification of varying size aspects by rules see [32].

Example 6.3 Let the KB contain the following knowledge items together with the taxonomy of Section 3: The first rule is an aggregation rule. Two neighbouring truncated cones are composed to a `biconic`. Logically `f` is a skolem function which replaces the existentially quantified variable representing the new instance. The second and the third rule specify kinds of cuts and how the workpiece has to be hold in the lathe-turning machine. Two truncated cones are neighbours in a workpiece, if their coordinates and radii coincide, as specified by the fourth rule.

Rules:

```
biconic(f[X,Y],left[X],right[Y]) <= truncone(X), truncone(Y), neighbour(X,Y).
cut(X,to-right,lengthwise) <= rshoulder(X).
chuck(right,clamping) <= workpiece(Y), cut(_,to-right,_), contour(Y,ascending).
neighbour(X,Y) :- c2(X,C), c1(Y,C), r2(X,R), r1(Y,R).
```

Atomic assertions:

```
workpiece(wp1).  contour(wp1,ascending).
```

Role-filler assertions:

```
c1(tc1,0).  c2(tc1,5).  r1(tc1,10).  r2(tc1,10).
c1(tc2,5).  c2(tc2,5).  r1(tc2,10).  r2(tc2,15).
```

Membership assertions:

```
cylinder(tc1).
asc-ring(tc2).
```

Given the state of the KB as specified above, the first rule can be applied: The first and the second premise are satisfied with substitution $\sigma = \{\texttt{tc1/X}, \texttt{tc2/Y}\}$, because every `cylinder` and

every `asc-ring` is a `truncone` (see Fig. 3). The third premise can be proved by the top-down rule. The derived biconic `f[tc1,tc2]` is realized as an `rshoulder` by the taxonomic component. Then in the next cycle the second rule is triggered, because an `rshoulder` is also a shoulder. It is derived that the shoulder is manufactured by lenghtwise cuts from left to right. This new fact then triggers the third rule, which derives the kind of chucking fixture at the right end of the workpiece.

This example shows, that it is not necessary to repeat the definition of a rule for every concept in the terminology which describes an aggregate. The automatically computed subsumption graph helps the knowledge engineer to find the most general level on which he can formulate a rule. For example, instead of defining aggregation rules for `hill`,`lshoulder`, `rshoulder` etc. separately, it is sufficient to do so only for a `biconic`, the most general composition of two neighbouring truncated cones.

6.3 Goal-directed Bottom-up Evaluation

Because semi-naive bottom-up evaluation computes the least fixpoint of a logic program it is very inefficient if bindings for some argument positions are given in the query. A top-down strategy, however, would compute only the derivations necessary to answer the query. A drawback of most top-down strategies, for example the one used by PROLOG, is that they are incomplete. To use the efficient and complete bottom-up evaluation also for query answering, rewriting strategies have been developed.

As pointed out by [10] there are two modes of information passing in evaluating a query in a logic program. The first is called *sideway information passing*: by solving a premise predicate variable bindings are obtained which can be passed to another premise in the same rule to restrict the computation for that predicate. In the second mode information is passed to a rule from the query by unification with the head of the rule. Since pure bottom-up evaluation does not take into account a query, *sideway* information passing is the only information passing mode. To support goal-directed reasoning and to simulate the second information passing mode in bottom-up evaluation, the Magic Set rewriting strategy introduces auxiliary predicates and an additional fact – called Magic Seed. The arguments of the seed fact are exactly the variable bindings of the query. All rules, which can derive instantiations of the query, will get an additional premise that can be satisfied by this fact. Thus, the variable bindings of the query are passed to the body of the applicable rules at compile time. The rewriting algorithms restrict the number of deducible facts to those relevant to answer the query. Generalized Magic Set (GMS) transformation, which is also applied in CoLab extends the sideway information passing strategy from base predicates to derived predicates.

Because of its horizontal compilation effort, GMS rewriting in CoLab is used only for query forms known at compile time. Unforeseen queries are answered by the top-down strategy. The GMS transformation can be applied only for rules having rule-defined predicates in their conclusion; it cannot be used, e.g. for aggregation rules (see above). Premises with predicates defined by another component of our hybrid knowledge representation system, are dealt with in the same way as built-in or base predicates.

6.4 Tuple-oriented Forward Reasoning

While the bottom-up approach of the semi-naive strategy as described in Section 6.2 computes all the consequences of the whole KB by a fixpoint operation, the objective of the tuple-oriented approach is to compute *only the derivations of an explicitly given set of facts.* Another difference to the set-oriented approach of Section 6.2 is, that the premises with rule-defined predicates are proved by SLD resolution instead of a simple look-up in the fact base.

The tuple-oriented forward reasoning approach is implemented by a horizontal transformation of the Horn rules to backward reasoning Horn clauses of RELFUN, thus performing forward reasoning in a backward reasoning system. This transformation is equivalent to the partial evaluation of a forward reasoning meta interpreter as described in [36]. Every rule

$$C :- P_1, \ldots, P_m$$

is translated into a *sequence* of forward clauses following this pattern:

$$\begin{array}{lll} forward(P_1, C) & :- & P_2, \ldots, P_m, retain(C). \\ forward(P_2, C) & :- & P_1, P_3, \ldots, P_m, retain(C). \\ & \ldots & \\ forward(P_m, C) & :- & P_1, \ldots, P_{m-1}, retain(C). \end{array}$$

Applying a forward clause corresponds to a one step forward execution of the original Horn rule triggered by P_i. *retain* is a built-in operator asserting the derived fact if it has not already been derived in a previous step. Because forward evaluation of a Horn clause can be triggered by a fact unifying any premise of the clause, for every premise $P_1, \ldots, P_m$ of the original clause a forward clause is generated. Exceptions are backward-defined premises and built-in operators like `is`. This is an important difference to Yamamoto's and Tanaka's translation for production rules [64].

Various control strategies are available: depth-first and breadth-first enumeration of results, and computing the derivations all at once. These strategies are themselves represented as Horn clauses, so that they can be adapted for specific applications. For more details on this transformation approach see [35].

Example 6.3 (continued): Consider the rules of Example 6.3. The third rule

```
chuck(right,clamping) <= workpiece(Y), cut(_,to-right,_), contour(Y,ascending).
```

will be transformed to

```
forward(workpiece[Y],chuck[right,clamping])
     :- cut(_,to-right,_), contour(Y,ascending), retain(chuck[right,clamping]).
forward(cut[_,to-right,_],chuck[right,clamping])
     :- workpiece(Y), contour(Y,ascending), retain(chuck[right,clamping]).
forward(contour[Y,ascending],chuck[right,clamping])
     :- workpiece(Y), cut(_,to-right,_), retain(chuck[right,clamping]).
```

Starting forward chaining with the fact `contour(wp1,ascending)` will trigger the third `forward` clause. The first premise `workpiece(wp1)` is unifiable with the corresponding fact. The second premise can be proved by top-down evaluation using the first two bidirectional rules of Example 6.3. Thus the fact `chuck(right,clamping)` is derived, which is retained and can itself be used to trigger any further rules.

6.5 Rule Compilation

In the previous subsections some of the horizontal rule compilation steps have already been mentioned:

- Rules with multiple conclusions are transformed to Horn clauses with exactly one conclusion for evaluation by the top-down proof procedure (Section 6.1).
- Bottom-up rules are transformed according to the extended Generalized Magic Set rewriting strategy to achieve goal-directed bottom-up reasoning (Section 6.3).
- Bidirectional rules are transformed to backward chaining forward clauses to derive the consequences of an explicitly given set of facts (Section 6.4).

This subsection gives a short introduction into vertical compilation for the various bottom-up reasoning strategies.

Vertical Compilation for Semi-naive Evaluation

An Abstract Machine for efficient execution of the semi-naive strategy has been developed [25]. It is a modification of the Rete pattern match algorithm [26], an implementation method for production systems. The approach is based on a discrimination network, which keeps the state of all partially instantiated rules. Rule premises are compiled into sequences of match operations similar to unification operations of the WAM. Associated with each premise is a memory containing the variable bindings found so far. These partial instantiations are propagated to find applicable rules. In contrast to production systems the order of rule application in a deduction system does not matter. Therefore, instead of maintaining a conflict set, rules can be applied as soon as an instantiation is found.

The abstract machine has been extended by some special feature to support hybrid reasoning. In particular, to prove premises with top-down predicates access to RelFun's relational-functional machine (RFM) is supported: the conjunction of top-down premises is compiled into an RFM query. The interface is homogeneous, because both machines are implemented on an equivalent level. The instruction set and the term representation are very similar.

Vertical Compilation into the RFM

After source-to-source transformation of a rule system $\mathcal{P}$ into a set of clauses $\mathcal{FP}$ for tuple-oriented forward reasoning (Section 6.4), the clauses of $\mathcal{P}$ and $\mathcal{FP}$ are compiled vertically into code for an extended version of the RFM, a Warren Abstract Machine for top-down evaluation of logic

programs [63], which is capable to handle functional clauses of RELFUN, [12]. For a detailed description of this compilation see [37].

The clauses obtained by horizontal transformation have one fundamental drawback: they are represented with a single predicate symbol **forward**. After compilation there is one large procedure with costly search for an applicable clause. A special code area for forward clauses can make this predicate implicit and clauses with the same trigger predicate can be grouped together into one procedure. Thus, the single large **forward** procedure is decomposed into one procedure for each trigger predicate. Since a direct compilation of the forward clause in this way would conflict with the original definition for the trigger predicate, a special code area is introduced – besides the original code area of the WAM – for the compilation of forward clauses.

While values on the local and global stacks may be destroyed on backtracking, derived facts must survive for the whole forward inference chain. Assertions of derived facts by the **retain** operator can be rather inefficient if program code is altered dynamically. At machine level information about derived facts can be held more compactly. Therefore the WAM is extended by a special stack area for derived facts, called *retain stack*, and no reference from the retain stack to any other memory cell is permitted. The operator **retain** is compiled into a sequence of WAM operations pushing its argument – the derived fact – onto the retain stack. To accept a derived fact, it must be ensured that it is not subsumed by any structure already existing on the stack. Therefore the actual fact is matched against every entry on the retain stack. If this subsumption test fails, the derived fact is pushed onto the retain stack.

7 The μCAD2NC Case Study

The μCAD2NC (micro-CAD-to-NC) case study has been conducted to test CoLab with a prototypical application solving a simplified model version of a real-world problem. μCAD2NC-II is a revised version of μCAD2NC-I, which was described in [16]. Both are knowledge-based systems generating workplans for idealized lathe CNC machines. They transform CAD-like geometries of rotational-symmetric workpieces into abstract NC programs, using declarative (term) representations for all processing steps:

> Given the geometry of a rotational-symmetric workpiece, generate NC macros for rough-turning the workpiece on an abstract CNC lathe machine.

The μCAD2NC models perform the most interesting **central** phases of the CAD-to-NC transformation; writing a front end for converting real CAD data to our KB representation, and a back end for converting our NC macros to programs for a real CNC machine would be a routine task. The focus of this work is on exemplifying techniques of the hybrid, declarative CoLab system for the central subtasks of CAD-to-NC transformations. Thus, the intention behind μCAD2NC-II is not to provide a polished solution tuned for production: instead, it enables us to study how AI techniques and formalisms can be combined to solve a non-toy problem. In μCAD2NC-II the components of CoLab collaborate in a typical synergetic manner.

The whole NC-planning process of μCAD2NC-II is an instance of the "heuristic classification" inference scheme [21] as illustrated in Figure 4. The input to a process planning system is a very

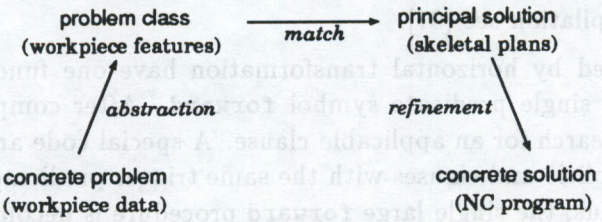

Figure 4: Clancey's Scheme Applied to Process Planning

'elementary' description of a workpiece as it comes from a CAD system. If possible at all, process planning with these input data starting from (nearly) first principles would require very complex algorithms. Thus, planning strategies on such a detailed level are neither available nor would they make sense. Instead, human planners apparently store a hierarchically organized library of prefabricated *skeletal plans* in their memories [56]. Each of these plans is accessed via a more or less abstract description of a characteristic (part of a) workpiece, which is called a workpiece *feature* [39]. The top-level feature thus associates a workpiece description (geometry/topology, technology) with the corresponding abstract manufacturing method (NC program, tool change, chucking). The first step of μCAD2NC-II is the generation of an abstract feature description from the elementary workpiece data; the features obtained characterize the workpiece w.r.t. its production. In the second step, the skeletal plans (associated with the features) are retrieved. They are merged and parameterized with concrete geometrical data and manufacturing information (e. g. selection of appropriate tools and chucking fixtures), resulting in an NC-like program.

In the following subsections, each of the three phases is described in more detail. Appendix B shows a representative portion of the μCAD2NC-II KB.

7.1 Feature Aggregation

Process planning starts when data representing a workpiece are given to the μCAD2NC-II system (cf. Fig. 5). Geometrical descriptions of the workpiece's surfaces and topological neighbourhood relations are the central parts of this representation. In the first phase an abstract feature tree is generated out of these input data.

Following the distinction between concepts and instances, it is rather natural to define all the possible features and surfaces as concepts in TAXON's TBOX and to represent a single case, i.e. a workpiece, by assertions in the ABOX. Examples of concept definitions have already been given in Section 3. Fig. 6 shows part of the taxonomy of μCAD2NC-II. The concept definitions are listed in Appendix B. Although both surfaces and features are represented by concept definitions, terminological and rule-based reasoning work together to derive the feature tree. The reasons have already been explained in previous sections (3.3 and 6.2).

In the tradition of declarative pictures, graphics, and geometries in functional [55] and logic programming [41, 34, 54], a term representation is used. Each surface region is represented by a set of attribute terms (see Section 6.2) asserted individually in COLAB's ABOX, where they become realized w.r.t. the taxonomy (cf. Section 3.1). For example, two neighbouring truncated

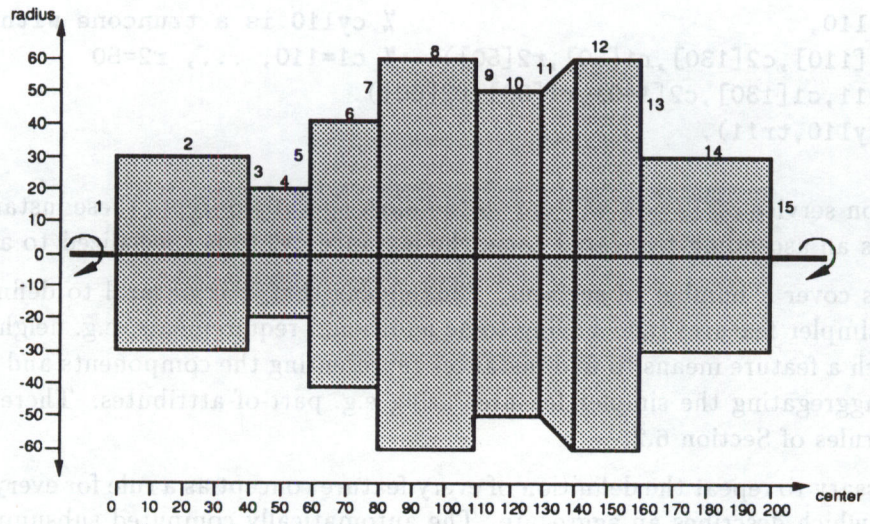

Figure 5: A Sample Workpiece

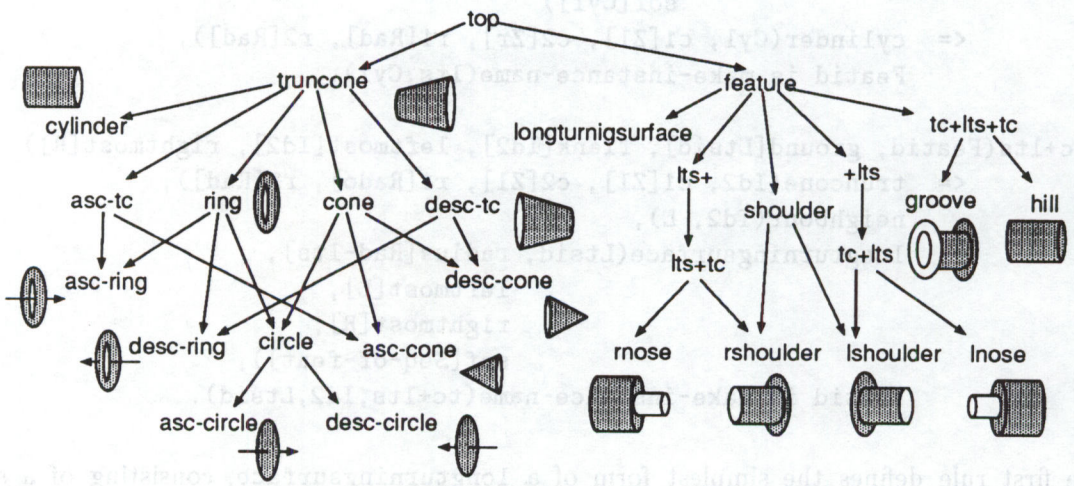

Figure 6: Concept Hierarchy of Relevant Surfaces and Features

cones cyl10 and tr11 of Fig. 5 are represented as two attribute-term assertions and a topological neighbourhood relation. The attributes represent their coordinates and radii (see Section 3):

```
truncone(cyl10,                                % cyl10 is a truncone with
         c1[110],c2[130],r1[50],r2[50]).       % c1=110, ..., r2=50
truncone(tr11,c1[130],c2[140],r1[50],r2[60]).
neighbour(cyl10,tr11).
```

The realization service of TAXON will derive the most special concepts these instances belong to. While tr11 is an ascending truncated cone, the instance cyl10 is specialized to a cylinder.

Most features cover a number of surfaces. This means that it is natural to define a feature as consisting of simpler features (and having some additional requirements, e.g. neighbourhood). Thus, finding such a feature means to find instances representing the components and to generate a new instance aggregating the simpler features using e.g. part-of attributes. Therefore we use the aggregation rules of Section 6.2.

It is not necessary to repeat the definition of every feature concept as a rule for every concept in the terminology which describes an aggregate. The automatically computed subsumption graph helps the knowledge engineer to find the most general level on which to formulate a rule. For example, the definition of aggregation rules for lts+, +lts, tc+lts+tc and longturningsurface is sufficient to aggregate all the features of Fig. 6. Two sample aggregation rules are (see also Appendix B):

```
longturningsurface(Featid, radius[Rad],
                           leftmost[Cyl],
                           rightmost[Cyl],
                           sof[Cyl])
        <=  cylinder(Cyl, c1[Zl], c2[Zr], r1[Rad], r2[Rad]),
            Featid is make-instance-name(lts,Cyl).

tc+lts(Featid, ground[Ltsid], flank[Id2], leftmost[Id2], rightmost[R])
        <=  truncone(Id2, c1[Zl], c2[Zl], r1[Rado], r2[Rad]),
            neighbour(Id2, L),
            longturningsurface(Ltsid, radius[Rad-lts],
                                      leftmost[L],
                                      rightmost[R],
                                      sof[Seq-of-feat]),
            Featid is make-instance-name(tc+lts,Id2,Ltsid).
```

The first rule defines the simplest form of a longturningsurface, consisting of a single cylinder. The second one aggregates a truncone and a longturningsurface to a tc+lts.

To generate the feature abstraction, the rule system starts bottom up from the assertions describing the workpiece and asserts the aggregated features that can be derived. As soon as a new feature instance or some additional information about an already existing instance is asserted, TAXON computes its most special concept associations using the realization service. This 'type'

information, resulting in new facts in the ABox, can again trigger rules to derive further features building on the feature just found. From the workpiece facts about `cyl10` and `tr11` and the rules above it can be derived that `lts-cyl10` is a `longturningsurface`.

To simplify the implementation we have restricted the rule/taxonomy interface to atomic instance names. So we use a built-in operation `make-instance-name` that generates *new atomic* instance names such as `lts-cyl10`. However, we have already seen that this less declarative built-in can be avoided using structures as instance identifiers (cf. Example 6.3).

Knowing that `lts-cyl10` is a `longturningsurface`, the second rule can be applied, deriving a new feature, `lts+tc-lts-cyl10-tr11`. This new feature will be realized as being an `rshoulder` by TAXON. The information that it is an `rshoulder` or any of its generalizations like `shoulder` can be used to satisfy the premises of other rules. Thus, the result of the feature aggregation phase is a feature tree.

The nodes of the tree are labeled with the features and surfaces of the workpiece. The label of the root of the tree is the workpiece itself. The nodes $N_1, \ldots, N_n$ are sons of a node N if the feature at node N is composed of the features or surfaces represented by nodes $N_1, \ldots, N_n$.

7.2 Skeletal-Plan Association

The next two phases of process planning are the association of skeletal plans with the generated features of the feature tree and their refinement and merging. Skeletal plans are abstract descriptions of the operations which have to be executed to manufacture the feature. This phase is complicated by the fact that each feature can occur as part of other features, but each feature instance should be manufactured exactly once.

Skeletal-plan association starts from the root node of the feature tree which represents the entire workpiece, and tries to find a fitting skeletal plan. If one can be found, then this phase is finished. If, for a feature F represented by a node N of the feature tree, no skeletal plan can be found, skeletal-plan association is recursively applied to the subfeatures represented by the sons $N_1, \ldots, N_n$ of N, etc., until the surfaces of the workpiece are reached.

Skeletal plans are represented as footed clauses with function name `gen-skp`. The conditions of the clauses are descriptions of the features or surfaces to which skeletal plans are applicable. The value of the clause is the skeletal plan for this feature, which sometimes has to be merged with the skeletal plans for the subfeatures. A skeletal plan is represented as a triple containing the cutting direction (from right to left or from left to right), the kind of the cut (lengthwise or contour) and the sequence of actions. In the following example the skeletal plan for an `rshoulder` has to be merged with the plan for the longturningsurface, which is the ground of the shoulder:

```
gen-skp(Fid) :- cylinder(Fid, []) & [skp].
gen-skp(Fid) :-
        rshoulder(Fid, [ground[Lts], leftmost[Lshid], rightmost[Rshid]]),
        truncone(Lshid, [c1[X1]]),
        truncone(Rshid, [c1[X2], c2[X3], r1[Y1], r2[Yh]]) &
        merge-skp(
          [skp,                          %  ----------------------->/ Yh
           dir[to-right],                %  ---------------------->/
           kind[lengthwise],             %  ---------------------/   Y1
           seq[                          %   X1                 X2 X3
            actions[
              [to-right, lengthwise,
               geo[p[Y1, X1], p[Y1, X2], p[Yh, X3]]]],
            actions[]]],
          gen-skp(Lts)). % skeletal plan for Lts subfeature merged in
```

7.3 Skeletal-Plan Refinement

Skeletal-plan refinement combines as its most important tasks the merging of individual skeletal plans and the selection of the appropriate tools.

Skeletal-plan merging is specified by the function **merge-skp**, which is defined by RELFUN's footed clauses, e.g. for merging **unfixed** with fixed (here, **both**) plans it contains the clause:

```
merge-skp([skp, dir[unfixed], kind[contour], com[A11, A12]],
          [skp, dir[both], Kind, seq[A21,A22]]) :-
    Dir is leftmost-dir(A21) &          % find the merge-point direction
    merge-skp(tup(skp,                  % recurse with both skps fixed:
                  dir[Dir], kind[contour],      % fix the direction
                  sequentialize(com[A11, A12], Dir)), % fix order of actions
              [skp, dir[both], Kind, seq[A21,A22]]).  % already fixed
```

The result is a skeletal plan, where the actions are merged in such a way that cuts with equal directions are put together. For each subsequence of actions the kind of the cut is derived such that a **lengthwise** cut is made only when all actions in the sequence can be performed by lengthwise cuts. Otherwise the kind of the cuts for this sequence becomes **contour**.

Candidate tools for each of the cutting sequences are selected by the constraint-propagation component of CoLab. Tool selection heavily depends on a lot of geometrical (e.g. edge-angle) as well as technological parameters (e.g. material, process etc.) which restrict the choice of a suitable tool system. Moreover, the tool system itself consists of some subparts which have to be combined. For our prototypical application we only consider three of them, namely the holder, the material and the geometry of the plate. In practice, there are a lot of restrictions, which holder to use for which plate, which kind of plate geometry to use for which workpiece contour, etc. As an example, the following CoLab item represents the definition of a primitive constraint

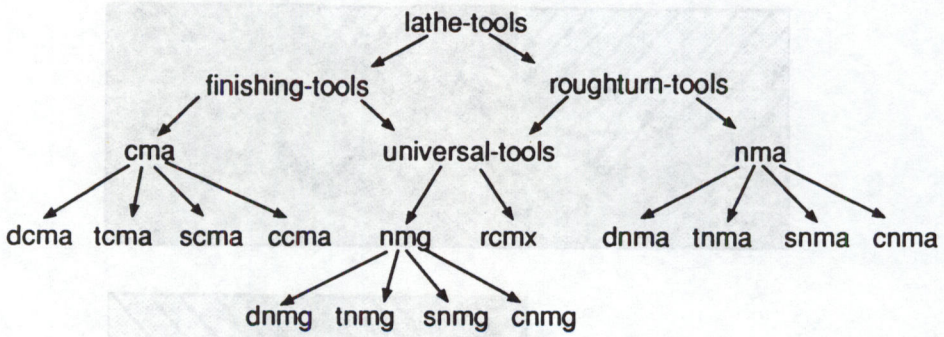

Figure 7: Part of the Hierarchical Domain of Lathe Tools

named `holder-tool` between the symbolic classes of holders and lathe tools.

```
holder-tool(Holder:holders, Tool:lathe-tools) :-
   {(pt,tnma), (pt,tnmg), (pt,tcma-41),
    (ps,nmg), (ps,snma-41),
    (pc,cnmm-71), (pc,cnmm-41), (pc,cnmg), (pc,cnma),
    ...
    (ss,finishing-tool)}.
```

All these restrictions, which need not be binary, constrain the search space for valid combinations of values for the various problem variables. Therefore, it seems to be most natural to use a constraint-propagation system to perform this subtask.

The fact that domains may be defined hierarchically, instead of explicitly enumerating all the elements of the domain, is very useful for the application of CONTAX within μCAD2NC-II, since the domains of lathe tools and holders can be hierarchically structured in a very natural way (see Fig. 7 and Appendix B). The outcome of the propagation process then is the set of all globally consistent assignments of values from the given domains to the problem variables such as `holder` and `tool`.

In the μCAD2NC-II application, CONTAX is called for the longest possible sequence of actions using a single tool to propagate through the constraint net the initial value assignments given for that particular sequence. If no such tool can be found, elements are repeatedly eliminated from the sequence via a backtracking function `tool-select` (cf. Appendix B). The constraint net itself does not change during the whole μCAD2NC-II session; only the initial assignments differ from feature to feature. Therefore it was possible to define and compile the whole constraint net used for tool selection, too, before running the μCAD2NC-II system.

The following tuple shows the representation of an NC-like program generated for our sample workpiece (cf. Fig. 8):

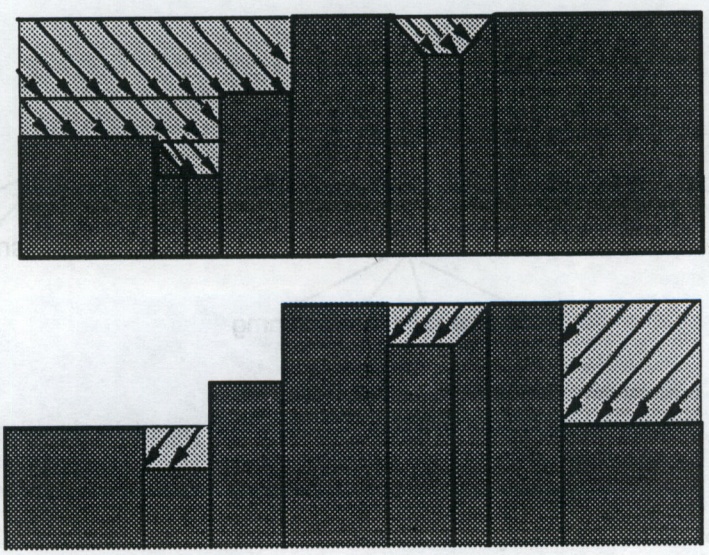

Figure 8: Example Workpiece with Contour Cuts

```
[skp,
 dir[both],
 kind[contour],
 [seq,
  actions[
   [tool[rcmx, tmaxp-prr30],
    [to-right, contour, geo[p[40, 0], p[40, 80], p[60, 80]]],
    [to-right, contour, geo[p[30, 0], p[30, 60], p[40, 60]]],
    [to-right, contour, geo[p[30, 40], p[20, 40], p[20, 60], p[30, 60]]],
    [to-right,
     contour,
     geo[p[60, 110], p[50, 110], p[50, 130], p[60, 140]]]],
   [tool[rcmx, tmaxp-prl40],
    [to-left,
     contour,
     geo[p[60, 110], p[50, 110], p[50, 130], p[60, 140]]],
    [to-left, contour, geo[p[30, 40], p[20, 40], p[20, 60], p[30, 60]]],
    [to-left, contour, geo[p[60, 160], p[30, 160], p[30, 200]]]]],
  actions[]]]
```

It consists of two subsequences of contour cuts. The cutting direction is of type **both**, because the first sequence consists of four cuts from left to right, while the second sequence consists of three cuts from right to left. The two cutting sequences are performed with different tools. The cutting kind for the whole action list has been fixed as **contour**.

8 Conclusions

Let us first conclude on all CoLab components and their cross-connections, and then come to the main conclusions concerning CoLab as a whole.

The RelFun language attempts to combine and extend programming concepts and techniques that have accumulated in the relational (principally, PROLOG) and functional (prototypically, LISP) communities. The relational/functional integration entails a continuing cross-fertilization of the two language styles. For instance, relational (logical) variables are reused for enabling non-ground function arguments and values. Conversely, varying-arity and certain higher-order operators are transferred from the functional to the relational world. Thus, RelFun provides a tunable system of orthogonal relational/functional language extensions of a pure-PROLOG-like kernel, which can be used in isolation and in free combination. The relational-functional language and the bidirectional rule system share relational backward rules as a common representation; this broad overlap has enabled an intensive reuse of compilation software and methods, discussed in Section 6. RelFun and Contax have (non-recursive) DATALOG clauses in common, which permits horizontal compilation as sketched in Section 4. The connection of relational/functional clauses to the taxonomic component is currently performed by calling Taxon from within relational rule premises, as applied in Section 7; another coupling currently investigated would employ 'concept-sorted' logical variables in clause conclusions [1]. Future research concerning RelFun's expressive power is planned on a tighter incorporation of attribute assertions and finite domains, an introduction of knowledge items for deterministic functions and adaptation rules, and the addition of dynamic assertions or local definitions and modules. With respect to RelFun's efficiency, we plan to improve the lower levels of the WAM compiler and to implement a more powerful emulator.

The rule formalism combines forward and backward reasoning of Horn rules. It supports a bottom-up strategy for fixpoint computation and a top-down proof procedure as in PROLOG. Goal-directed bottom-up reasoning for query answering and tuple-oriented derivation of consequences of a specified set of facts are achieved by horizontal rule transformation. The control strategy of the tuple-oriented forward reasoning mechanism is induced from the SLD-resolution procedure of logic programming. The set-oriented bottom-up reasoning mechanism for query answering makes it possible to integrate a deductive database system into CoLab. A common rule set is used for both reasoning directions. This requires a declarative representation of bidirectional rules. Especially, control information is not allowed in bidirectional rules themselves, but the knowledge engineer has the opportunity to fix the direction of individual rules, which in turn fixes the verification strategy of particular premises. The integration of the bottom-up and a complete top-down strategy into a single run-time environment is one of the future research topics for rule-based deduction. Additional evaluation strategies can be attained by horizontal transformation.

The constraint formalism Contax provides efficient propagation methods for constraints over hierarchically structured domains. Depending on the application, these domains and their structure can be defined directly or by providing concept definitions for the taxonomic component which are then classified and result in the domain structure used to speed up the constraint propagation. Contax uses an extended HAC algorithm and computes locally and globally consistent value assignments. The main topic for future research in the constraint formalism will

be on the abstraction of a more generic constraint solver over domains of different types (e.g., intervals, linearly ordered sets, compound variables) supporting specialized propagation methods, the embedding of finite domain consistency techniques within the logic programming framework [48, 33, 46, 49], as well as the integration of CONTAX as the constraint solver for concrete domains within the taxonomic component [60].

The taxonomic component TAXON with its integrated concrete domains allows the definition of a tailored vocabulary for an application (domain). With the decision procedures supplied for the reasoning services this vocabulary can be **analyzed** and **used** by the affirmative formalisms. For example, in Section 4 it has been shown how knowledge about the subsumption relation of concepts structuring a domain can be used to speed up constraint propagation. Also, the tight integration of forward reasoning and the taxonomic inferences, as exemplified in Section 7, has demonstrated that these formalisms complement each other in a nice way:

- The forward reasoning component employs the realization service of the taxonomic component to test complex premises that have been formulated in the tailored vocabulary of concepts.
- Conversely, the forward execution of the rules performs aggregations of compound instances, which cannot be carried out by the taxonomic component.

Further research related to the taxonomic formalism concerns the language constructs themselves (more expressiveness, still efficient reasoning) and further uses of a tailored vocabulary of concepts in affirmative formalisms such as the backward reasoning component [1].

COLAB as a hybrid system has already caused synergetic effects between the four main AI representation formalism it combines. In the **overlapping areas** we often found a new solution for one formalism by transcribing the solution of another formalism; for instance, algorithms for feature aggregation were exchanged between a RELFUN and a forward-rule/taxonomic version. In the **complementary areas** we have found natural hybrid solutions that require at least the pairwise combination of formalisms; for instance, the feature-tree description for workpieces alternates forward-rule aggregation and taxonomic classification.

A similar synergy was achieved for the compilation methods. The emphasis on horizontal compilers has helped us in creating COLAB-wide abstractions such as bidirectional rules and a shared ABOX, and in experimenting with alternative information-preserving language nuclei such as 'relationalized' and 'footed' RELFUN. The development of vertical compilers was begun with the RFM, whose LISP-implementation permitted fast extension for forward-rule execution and finite domains, and whose WAM principles also produced new insights in the compilation/emulation of taxonomies.

By supporting formalisms of different, sometimes even competing, subcommunities, COLAB also permits to absorb novel techniques developed in any of the subfields covered. This already happened, for example, by incorporating relational indexing techniques, a *magic-set* method for bottom-up processing, an extended HAC algorithm, and a complete tableau-based classification algorithm. Even when we will proceed toward a less hybrid language (see below), we can still profit from work done in the surrounding disciplines involved in the present hybrid COLAB system.

The above points concerning COLAB's hybridness, declarativeness, and compilability also mark essential differences to commercial expert-system shells: In their desire to present themselves

with a polished surface and to maintain compatible versions without using horizontal compilation, they do not incorporate very recent scientific results; in their desire to keep their customers, they do not regard declaratively formulated KBs, portable to other shells, as a number-one priority. Further distinguishing characateristics of CoLab are its supply of fine-grained knowledge items for hybrid modularization, its access primitives permitting its use as a toolbox with one to four components on top of LISP, its in-depth testing in technical domains such as mechanical and electrical engineering (see below), its flexible implementation by using a subset of COMMON LISP for rapid prototyping, and its free availability for research purposes. A final difference between CoLab and commercial hybrid expert-system shells is CoLab's KL-ONE/KRYPTON-inspired 'essential' (taxonomic-affirmative) hybridity and its development towards a further homogenized formalism (see below).

Although this paper has presented CoLab exclusively using examples from or around mechanical engineering, recently the industrial TOOCON project has proved that our tool system is equally applicable to electrical engineering: a prototypical configuration system for low-voltage switch boards was written in CoLab by three people in six months. This reinforces our feeling, acquired by testing smaller examples in various areas, that CoLab is in fact a general system.

Future work on CoLab as a whole mainly concerns the issue of destilling a more homogeneous formalism from the hybrid language without losing essential capabilities of the current version, where a running μCAD2NC version should always be maintained. This includes further encapsulation of the access primitives and a review of the knowledge items w.r.t. to two questions for each item: (1) Does it belong to the kernel of the envisaged homogeneous language? (2) Is it suitable as a formula that can be interpreted by a common semantics? In the new version, the uppermost layer of knowledge items may abstract and join several items still separated in the current CoLab, perhaps horizontally splitting and compiling them back into lower-level knowledge items.

A homogeneous successor version of CoLab will be needed for global knowledge analysis as planned in the VEGA project. For example, the classification service in CoLab analyzes concept definitions (Section 3.3.2). In the successor version we would like to have similar services that analyze both taxonomic and assertional knowledge in an analogous way. Since one aim of VEGA is the development of such a homogeneous language suitable for knowledge validation and exploration, we do not know at this time how much of the expressive power of CoLab can be kept in spite of the homogenization. However, we feel that homogenizing the application-oriented hybrid CoLab language is preferable to first developing a theoretical homogeneous language and only then trying it on practical problems.

Acknowledgements

The research presented in this paper and the development of CoLab have been carried out by the knowledge-compilation group of the ARC-TEC project at the DFKI, supported by the BMFT under grant ITW 8902 C4.

We thank Prof. Michael M. Richter for encouraging us computer scientists to go into the real-world area of mechanical engineering. Thanks are also due to Andreas Abecker, Dennis Drollinger, Klaus Elsbernd, Christian Falter, Martin Harm, Hans-Günther Hein, Michael Herfert, Björn Höfling, Christoph Jakfeld, Thomas Krause, Michael Kreinbihl, Thomas Labisch, Jörg

Müller, Thomas Oltzen, Bernd Reuther, Ralph Scheubrein, Michael Sintek, Harald Sohns, Werner Stein, Stefan Steinacker, and Frank Steinle, who implemented main parts of the CoLab system and the μCAD2NC application. We are also grateful to our colleagues Bernd Bachmann, Anne Schoeller and Holger Wache who have chosen CoLab as the basic representation language for the DFKI project TOOCON, supported by Daimler-Benz AG, and provided early feedback on the design and functionality of the CoLab system, and to Otto Kühn for carefully proofreading the final draft. The anonymous reviewers also made helpful comments on the presentation of this paper.

References

[1] A. Abecker. TAXLOG: Taxonomische Wissensrepräsentation und Logische Programmierung. Projektarbeit, 1993. In German.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the* ACM, 26(11):832–843, 1983.

[3] F. Baader, H.-J. Bürckert, B. Hollunder, W. Nutt, and J. H. Siekmann. Concept logics. Technical Report RR-90-10, DFKI, Kaiserslautern, Germany, 1990.

[4] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *Proceedings of the* 12^{th} *International Joint Conference on Artificial Intelligence*, 1991.

[5] F. Baader and P. Hanschke. Extensions of concept languages for a mechanical engineering application. In *Proceedings German Workshop on Artificial Intelligence, GWAI-92*. Springer, September 1992.

[6] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[7] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4:259–262, 1987.

[8] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD Conference*, pages 16–52. ACM, 1986.

[9] F. Bancilhon and R. Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 441–517. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[10] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–299, October 1991.

[11] H. Beringer and F. Porcher. A relevant scheme for Prolog extensions: CLP(conceptual theory). In *Proc. of ICLP 89*, pages 131–148, 1989.

[12] H. Boley. A relational/functional language and its compilation into the WAM. SEKI Report SR–90–05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.

[13] H. Boley. Extended logic-plus-functional programming. In *Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, LNAI. Springer, 1992.

[14] H. Boley. A direct semantic characterization of RELFUN. In E. Lamma and P. Mello, editors, *Proceedings of the 3rd International Workshop on ELP '92*, volume 660 of *LNAI*. Springer, 1993.

[15] H. Boley, K. Elsbernd, H.-G. Hein, and T. Krause. RFM manual: Compiling RELFUN into the relational/functional machine. Document D–91–03, DFKI, 1991.

[16] H. Boley, P. Hanschke, M. Harm, K. Hinkelmann, T. Labisch, M. Meyer, J. Mueller, T. Oltzen, M. Sintek, W. Stein, and F. Steinle. μCAD2NC: A declarative lathe-workplanning model transforming CAD-like geometries into abstract NC programs. Technical Report Document D–91–15, University of Kaiserslautern, DFKI, November 1991.

[17] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *International Conference on Management of Data*. ACM SIGMOD, 1989.

[18] R. J. Brachman, V. P. Gilbert, and H. J. Levesque. An essential hybrid reasoning system: knowledge and symbol level accounts in KRYPTON. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 532–539, August 1985.

[19] R. J. Brachman, V. P. Gilbert, and H. J. Levesque. An essential hybrid reasoning system: knowledge and symbol level accounts in KRYPTON. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 532–539, 1985.

[20] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[21] W. J. Clancey. Heuristic classification. *Artificial Intelligence*, 27:289–350, 1985.

[22] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, 1986.

[23] Y. Descotte and J.-C. Latombe. Making compromises among antagonist constraints in a planner. *Artifical Intelligence*, 27:183–217, 1985.

[24] F. Donini, B. Hollunder, M. Lenzerini, A. M. Spaccamela, D. Nardi, and W. Nutt. The complexity of existential quantificatoin in concept languages. Research Report RR-91-02, DFKI, January 1991.

[25] C. Falter. Compilation von Vorwärtsregeln in einer hybriden Expertensystem-Shell. Diploma thesis, University of Kaiserslautern, FB Informatik, 1992. In German.

[26] C. L. Forgy. *OPS5 User's Manual*. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania 15213, 1981.

[27] F. Frayman and S. Mittal. COSSACK: A constraints–based expert system for configuration tasks. In D. Sriram and R. Adey, editors, *Knowledge Based Expert Systems in Engineering: Planning & Design*. Computational Mechanics, 1987.

[28] E. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[29] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *1985 Symposium on Logic Programming*, pages 172–184. IEEE Computer Society Press, 1985.

[30] M. R. Genesereth and R. Fikes. Knowledge interchange format version 3.0 reference manual. Technical Report Logic-92-1, Stanford University, Computer Science Department, Logic Group, June 1992.

[31] P. Hanschke. Specifying role interaction in concept languages. In *Third International Conference on Principles of Knowledge Representation and Reasoning (KR '92)*, October 1992.

[32] P. Hanschke and K. Hinkelmann. Combining terminological and rule-based reasoning for abstraction processes. In *Proceedings German Workshop on Artificial Intelligence, GWAI-92*. Springer, September 1992.

[33] H.-G. Hein and M. Meyer. A WAM compilation scheme. In A. Voronkov, editor, *Logic Programming: Proceedings of the 1st and 2nd Russian Conferences*, volume 592 of *LNAI*, pages 201–214. Springer, 1992.

[34] R. Helm and K. Marriott. Declarative graphics. In E. Shapiro, editor, *Third International Conference on Logic Programming (ICLP)*, LNCS 225, pages 513–527. Springer, July 1986.

[35] K. Hinkelmann. Bidirectional reasoning of horn clause programs: Transformation and compilation. Technical Memo TM-91-02, DFKI, January 1991.

[36] K. Hinkelmann. Forward logic evaluation: Developing a compiler from a partially evaluated meta interpreter. Technical Memo TM-91-13, DFKI, October 1991.

[37] K. Hinkelmann. Forward logic evaluation: Compiling a partially evaluated meta-interpreter into the WAM. In *Proceedings German Workshop on Artificial Intelligence, GWAI-92*. Springer, September 1992.

[38] B. Hollunder, W. Nutt, and M. Schmidt-Schauß. Subsumption algorithms for concept description languages. In *9th European Conference on Artificial Intelligence (ECAI'90)*, pages 348–353. Pitman Publishing, 1990.

[39] C. Klauck, R. Legleitner, and A. Bernardi. FEAT-REP: Representing features in CAD/CAM. In *4th International Symposium on Artificial Intelligence: Applications in Informatics*, Cancun, Mexiko, 1991. An extended Version is also available as Research Report RR-91-20, DFKI.

[40] A. Kobsa. The SB-ONE knowledge representation workbench. In *Preprints of the Workshop on Formal Aspects of Semantic Networks*, 1989. Two Harbors, Cal.

[41] R. Kowalski. Logic as a computer language for children. In *European Conference on Artificial Intelligence (ECAI)*, pages 2–10, 1982.

[42] A. Mackworth, J. Mulder, and W. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[43] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–73, 1985.

[44] E. Mays, C. Apté, J. Griesmer, and J. Kastner. Experience with K-Rep: an object centered knowledge representation language. In *Proceedings of* IEEE CAIA-88, pages 62–67, 1988.

[45] P. Meseguer. Constraint satisfaction problems: An overview. *AI Communications*, 2(1):3–17, 1989.

[46] M. Meyer. Parallel constraint satisfaction in a logic programming framework. In *Proceedings of the International Conference on Parallel Computing Technologies (PaCT-91)*, pages 148–157. World Scientific Publishing Co., Singapore, September 1991.

[47] M. Meyer. Using hierarchical constraint satisfaction for lathe-tool selection in a CIM environment. In *Fifth International Symposium on Artificial Intelligence (ISAI'92)*, pages 167–177. AAAI Press, December 1992.

[48] M. Meyer, H.-G. Hein, and J. Müller. FIDO: Finite domain consistency techniques in logic programming. In A. Voronkov, editor, *Logic Programming: Proceedings of the* 1^{st} *and* 2^{nd} *Russian Conferences*, volume 592 of *LNAI*, pages 294–301. Springer, 1992.

[49] M. Meyer and J. Müller. Weak looking-ahead and its application in computer-aided production planning. In *Seventh International Symposium on Methodologies for Intelligent Systems (ISMIS'93)*, LNAI. Springer, to appear 1993.

[50] C. Moss. Commercial applications of large Prolog knowledge bases. In H. Boley and M. M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volume 567 of *LNAI*, pages 32–40. Springer, 1991.

[51] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems.* PhD thesis, University of Saarbrücken, 1989.

[52] P. F. Patel-Schneider, B. Owsnicki-Klewe, A. Kobsa, N. Guarino, R. McGregor, W. S. Mark, D. McGuiness, B. Nebel, A. Schmiedel, and J. Yen. Report on the workshop on term subsumption languages in knowledge representation. AI *Magazine*, 11(2):16–23, 1990.

[53] L. C. Paulson and A. W. Smith. Logic programming, functional programming, and inductive definitions. In P. Schroeder-Heister, editor, *ELP '91*, pages 283–309, Berlin, Heidelberg, New York, 1991. Springer. LNCS 475.

[54] F. Pereira. Can drawing be liberated from the von Neumann Style. In M. van Caneghem and H. Warren, editors, *Logic Programming and its Applications*, volume 2 of *Ablex series in Artificial Intelligence*. 1986.

[55] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[56] F. Schmalhofer, O. Kuehn, and G. Schmidt. Integrated knowledge acquisition from text, previously solved cases, and expert memories. *Applied Artificial Intelligence*, 5:311–337, 1991.

[57] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. Journal of Artificial Intelligence, 47, 1991.

[58] A. Schmiedel. A temporal terminological logic. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 2, pages 640–645. AAAI, 1990.

[59] W. Stein and M. Sintek. A generalized intelligent indexing method. In *Workshop "Sprachen für KI-Anwendungen, Konzepte - Methoden - Implementierungen" in Bad Honnef, 12/92-1*. Institute of Applied Mathematics and Computer Science, University of Münster, May 1992.

[60] F. Steinle. HAMLET: Erweiterung eines Constraint-Systems um Negation und Disjunktion und dessen Anbindung an eine Konzeptbeschreibungssprache. Projektarbeit, 1993. In German.

[61] C. Walther. A mechanical solution of Schubert's steamroller by many-sorted resolution. Technical Report A31-84, Universität Karlsruhe, Institut für Informatik I, Karlsruhe, Germany, 1984.

[62] D. E. Waltz. Generating semantic descriptions of scenes with shadows. *Technical Report MAC AI-TR-271, MIT, Cambridge MA*, 1972.

[63] D. H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[64] A. Yamamoto and H. Tanaka. Translating production rules into a forward reasoning Prolog program. *New Generation Computing*, 4:97–105, 1986.

A The Knowledge Items of CoLab

Following is a summary of CoLab's knowledge items and patterns sketching their infix syntax. In the LISP implementation of CoLab, each knowledge item is distinguished by a prefix tag.

Fact (Tag: `fact`)
< *Conclusion* > .

ABox **Assertion** (Tag: **asse**)
asse < *Assertion* > .

Bidirectional Rule (Tag: `rl`)
< *Conclusion* >$^+$ <= < *Premise* >$^+$.

Bottom-up Rule (Tag: up)
< *Conclusion* >$^+$ <- < *Premise* >$^+$.

Top-down Rule (Tag: hn)
< *Conclusion* >$^+$:- < *Premise* >$^+$.

Hornish Clause (Tag: hn)
< *Conclusion* > :- < *Premise* >$^+$.
< *Conclusion* > .

Footed Clause (Tag: ft)
< *Conclusion* > :- < *Premise* >* & < *Value* > .

Domain Definition (Tag: dd)
< *Domain name* > = {< *Element* >$^+$}.

Primitive Constraint (Tag: pc)
< *Constraint head* > :- {< *Tuple* >$^+$}.

Predicative Constraint (Tag: lc)
< *Constraint head* > :- lambda(< *Parameters* > . < *Expression* >).

Compound Constraint (Tag: cc)
< *Constraint head* > :- {< *Constraint* >$^+$}.

Primitive Concept (Tag: prim)
< *Concept name* > : prim.

Role (Tag: role)
< *Role name* > : role.

Attribute (Tag: attr)
< *Attribute name* > : attr.

Open Family of Pairwise Disjoint Primitive Concepts (Tag: ofam)
< *Family name* > $=_{\text{ofam}}$ {< *Concept name* >$^+$}.

Closed Family of Pairwise Disjoint Primitive Concepts (Tag: cfam)
< *Family name* > $=_{\text{cfam}}$ {< *Concept name* >$^+$}.

Concrete Predicate (Tag: cpred)
< *Predicate name* > $=_{\text{cpred}}$ lambda(< *Parameters* > . < *Expression* >).

Abstract Predicate (Tag: apred)
< *Predicate name* > $=_{\text{apred}}$ < *arity* > .

Concept Definition (Tag: conc)
< *Concept name* > $=_{\text{conc}}$ < *Concept term* > .

B A Hybrid Knowledge Base

This appendix presents a comprehensive selection of knowledge items of the μCAD2NC-II sample application. It starts with the definition of the terminology, followed by feature-aggregation rules, constraint definitions for tools selection, and function definitions for skeletal-plan association and refinement.

The following knowledge item introduces an open family, **levels-of-composition**, of concepts **atomic**, $\cdots$, **h-lts**, **lts+**, **+lts**, **+lts+**. These concepts are pairwise disjoint and are not restricted any further. The same effect could be achieved using several primitive concept names and the boolean connectives $\sqcap$, $\sqcup$, $\neg$.

$$\texttt{levels-of-composition} =_{\text{ofam}} \{\texttt{atomic}, \cdots, \texttt{h-lts}, \texttt{lts+}, \texttt{+lts}, \texttt{+lts+}\}.$$

A truncated cone is given by two centers and two radii, and is prohibited to degenerate.

r_1: attr.
r_2: attr.
c_1: attr.
c_2: attr.

$$\texttt{truncone-condition} =_{\text{cpred}} \texttt{lambda}(r_1, r_2, c_1, c_2 \,.\, r_1 \geq 0 \wedge r_2 \geq 0 \wedge (c_1 = c_2 \wedge r_1 \neq r_2 \vee c_1 \neq c_2 \wedge (r_1 > 0 \vee r_2 > 0))).$$

$$\texttt{truncone} =_{\text{conc}} \texttt{atomic} \sqcap \exists(\texttt{r}_1, \texttt{r}_2, \texttt{c}_1, \texttt{c}_2).\texttt{truncone-condition}.$$

Important specializations of truncated cones and the 'adjectives' **ascending** and **descending**.

$$\begin{aligned}
&\texttt{ring} &&=_{\text{conc}} \texttt{truncone} \sqcap \forall(\texttt{c}_1 = \texttt{c}_2).\\
&\texttt{cylinder} &&=_{\text{conc}} \texttt{truncone} \sqcap \forall(\texttt{r}_1 = \texttt{r}_2).\\
&\texttt{circle} &&=_{\text{conc}} \texttt{ring} \sqcap (\forall(\texttt{r}_1 = 0) \sqcup \forall(\texttt{r}_2 = 0)).\\
&\texttt{cone} &&=_{\text{conc}} \texttt{truncone} \sqcap (\forall(\texttt{r}_1 = 0) \sqcup \forall(\texttt{r}_2 = 0)).\\
&\texttt{ascending} &&=_{\text{conc}} \forall(\texttt{r}_1 \leq \texttt{r}_2).\\
&\texttt{descending} &&=_{\text{conc}} \forall(\texttt{r}_1 \geq \texttt{r}_2).\\
&\texttt{asc-tc} &&=_{\text{conc}} \texttt{truncone} \sqcap \texttt{ascending}.\\
&\texttt{desc-tc} &&=_{\text{conc}} \texttt{truncone} \sqcap \texttt{descending}.\\
&\texttt{asc-ring} &&=_{\text{conc}} \texttt{ring} \sqcap \texttt{ascending}.\\
&\texttt{desc-ring} &&=_{\text{conc}} \texttt{ring} \sqcap \texttt{descending}.\\
&\cdots
\end{aligned}$$

A (workpiece) feature has at least a leftmost and a rightmost truncated cone.

$$\begin{aligned}
&\texttt{leftmost} &&: \text{attr}.\\
&\texttt{rightmost} &&: \text{attr}.\\
&\texttt{feature} &&=_{\text{conc}} \exists\texttt{leftmost}.\texttt{truncone} \sqcap \exists\texttt{rightmost}.\texttt{truncone} \sqcap\\
&&&\quad \forall(\texttt{leftmost} \circ \texttt{c}_1 \leq \texttt{rightmost} \circ \texttt{c}_1).
\end{aligned}$$

For a long-turning surface only necessary conditions can be expressed.

$$\begin{aligned}
&\texttt{list} &&: \text{prim}.\\
&\texttt{radius} &&: \text{attr}.\\
&\texttt{sof} &&: \text{attr}.\\
&\texttt{car} &&: \text{attr}.\\
&\texttt{cdr} &&: \text{attr}.\\
&\texttt{lts} &&=_{\text{conc}} \texttt{h-lts} \sqcap \texttt{feature} \sqcap \exists\texttt{sof}.\texttt{list} \sqcap \exists(\texttt{radius} > 0).
\end{aligned}$$

The definitions of a left and a right shoulder are more complicated (i.e., realistic) than in Section 3.2.

```
tc+lts        =conc  +lts ⊓ feature ⊓
                     (leftmost ↓ flank) ⊓ (rightmost ↓ ground ∘ rightmost) ⊓
                     ∃flank.truncone ⊓ ∃ground.lts.
lnose         =conc  tc+lts ⊓ ∀flank.ascending.
lshoulder     =conc  tc+lts ⊓ ∀flank.descending.
lts+tc        =conc  lts+ ⊓ feature ⊓
                     (rightmost ↓ flank) ⊓ (leftmost ↓ ground ∘ leftmost) ⊓
                     ∃flank.truncone ⊓ ∃ground.lts.
rnose         =conc  lts+tc ⊓ ∀flank.descending.
rshoulder     =conc  lts+tc ⊓ ∀flank.ascending.
```

Grooves and hill comprise two shoulders or noses that share a ground.

```
left                  : attr.
right                 : attr.
tc+lts+tc             =conc  +lts+ ⊓ feature⊓
                             ∃left.tc+lts ⊓ ∃right.lts+tc ⊓
                             (leftmost ↓ left ∘ leftmost) ⊓
                             (rightmost ↓ right ∘ rightmost) ⊓
                             (left ∘ ground ↓ right ∘ ground).
up-step               =conc  tc+lts+tc ⊓ ∀left.lnose ⊓ ∀right.rshoulder.
down-step             =conc  tc+lts+tc ⊓ ∀left.lshoulder ⊓ ∀right.rnose.
hill                  =conc  tc+lts+tc ⊓ ∀left.lnose ⊓ ∀right.rnose.
groove                =conc  tc+lts+tc ⊓ ∀left.lshoulder ⊓ ∀right.rshoulder.
insertion-condition   =cpred lambda(d2w, d .
                                 d2w < 0.25 ∧ d < 30 ).
insertion             =conc  groove ⊓ ∀depth2width, depth.insertion-condition.
...
```

The following bidirectional rules aggregate surfaces and features to build a feature tree. Features derived by these rules are asserted into the ABox. The name of each feature is a new symbol generated by the function `make-instance-name`. Attributes common to all features are the `leftmost` and `rightmost` surface the feature is covering. This is necessary to check neighbourhood of surfaces and features. The first three rules aggregate a `longturningsurface` and one or two `truncones`, then some of the rules defining a

longturningsurface are presented. More complex features are not presented here.

```
lts+tc(Featid, ground[Ltsid], flank[Id2], leftmost[L], rightmost[Id2])
        <=  truncone(Id2, c1[Z1], c2[Zr], r1[Rad], r2[Rado])),
            neighbour(R,Id2),
            longturningsurface(Ltsid, radius[Rad-lts],
                                      leftmost[L],
                                      rightmost[R],
                                      sof[Seq-of-feat]),
            Featid is make-instance-name(lts+tc, Ltsid, Id2).

tc+lts(Featid, ground[Ltsid], flank[Id2], leftmost[Id2], rightmost[R])
        <=  truncone(Id2, c1[Z1], c2[Z1], r1[Rado], r2[Rad]),
            neighbour(Id2, L),
            longturningsurface(Ltsid, radius[Rad-lts],
                                      leftmost[L],
                                      rightmost[R],
                                      sof[Seq-of-feat]),
            Featid is make-instance-name(tc+lts,Id2,Ltsid).
```

A tc+lts+tc is an aggregation of a tc+lts (which is a generalization of lshoulder and lnose) and a lts+tc (which is a generalization of rshoulder and rnose) with a common ground. The following rules represent the affirmative knowledge that two shoulders or two noses can be aggregated, but that no nose can be combined with a shoulder. A tc+lts+tc can be taxonomically specialized to groove (an aggregation of shoulders) or to a hill (an aggregation of noses) as can be seen in Fig. 6. A workpiece as a whole is also a hill, if both its leftmost and rightmost surfaces are circles.

```
tc+lts+tc(Featid, left[Lshid], right[Rshid], leftmost[Id1], rightmost[Id3])
        <=  rshoulder(Rshid, ground[Id2],
                             flank[Id3],
                             leftmost[Rsleft],
                             rightmost[Rsright]),
            lshoulder(Lshid, ground[Id2],
                             flank[Id1],
                             leftmost[Lsleft],
                             rightmost[Lsrigh]),
            Featid is make-instance-name(tc+lts+tc,Id1,Id2,Id3).

tc+lts+tc(Featid, left[Lnid], right[Rnid], leftmost[Id1], rightmost[Id3]))
        <=  rnose(Rnid, ground[Id2],
                        flank[Id3],
                        leftmost[Rsleft],
                        rightmost[Rsright]),
            lnose(Lnid, ground[Id2],
                        flank[Id1],
                        leftmost[Lsleft],
                        rightmost[Lsright]),
            Featid is make-instance-name(tc+lts+tc,Id1,Id2,Id3).
```

...

A longturningsurface is a section on the workpiece, over which a horizontal cut at height **radius** can be made. It can have surfaces with radii less than the radius of the longturningsurface itself. The simplest longturningsurface consists of a cylinder only.

```
longturningsurface(Featid, radius[Rad],
                           leftmost[Cyl],
                           rightmost[Cyl],
                           sof[Cyl])
        <=  cylinder(Cyl, c1[Zl], c2[Zr], r1[Rad], r2[Rad]),
            Featid is make-instance-name(lts,Cyl).
```

If a right shoulder occurs at the left end of the workpiece, then there is a longturningsurface. The radius of the longturningsurface depends on the radius of the following **asc-tc**. The attribute **sof** has as value the list of all surfaces covered by the longturningsurface. The argument of the predicate **sub-lts**, which is not listed here, is a subsection of a longturningsurface. It covers sections, where horizontal cuts can be made over a complex furrowed contour.

```
longturningsurface(Featid, radius[Rad],
                           leftmost[Left],
                           rightmost[Rightm],
                           sof[Rshid | Seq-of-feat])
        <=  circle(Leftm, c1[Z], c2[Z], r1[0], r2[Rad-lim]),
            neighbour(Leftm, Left),
            rshoulder(Rshid, ground[_ground],
                             flank[_flank],
                               leftmost[Left],
                               rightmost[Right]),
            sub-lts(Right, Asc-tc, Rad, Seq-of-feat),
            asc-tc(Asc-tc, c1[Zl], c2[Zr], r1[Rad1], r2[Rad2]),
            neighbour(Rightm, Asc-tc),
            Rad2 > Rad,
            Featid is make-instance-name(lts,Leftm,Rightm).
```

Symmetrically, if a left shoulder occurs at the right end of the workpiece, then there is a longturning-

surface. The radius of the longturningsurface depends on the radius of the neighbouring **desc-tc**.

```
longturningsurface(Featid, radius[Rad],
                           leftmost[Leftm],
                           rightmost[Right],
                           sof[Seq-of-feat]))
        <=  circle(Rightm, c1[Z], c2[Z], r1[Rad-lim], r2[0]),
            neighbour(Right, Rightm),
            lshoulder(Lshid, ground[Ground],
                             flank[Flank],
                             leftmost[Lsleft],
                             rightmost[Right]),
            sub-lts(Desc-tc, Lsleft, Rad, Slsof),
            desc-tc(Desc-tc, c1[Zl],c2[Zr], r1[Rad1], r2[Rad2]),
            neighbour(Desc-tc, Leftm),
            Rad1 > Rad,
            apprel(Slsof, Lshid, Seq-of-feat),
            Featid is make-instance-name(lts, Leftm, Rightm).
```

We omit the remaining rules defining **longturningsurface**, and also the definition of **sub-lts**.

The knowledge relevant for the lathe-tool selection phase is represented using the constraint formalism. First, some finite domains are defined. Then, definitions of the constraints are given that link the variables together. Finally, all constraints are grouped together as one compound constraint **tool_sel** that represents all relevant knowledge that constrains the selection of appropriate lathe-tools for the different workpiece features:

The following COLAB specification represents some knowledge about the structure of the domain of lathe-tools:

```
lathe-tools     = {finishing-tools, roughturn-tools}.
roughturn-tools = {universal-tools, mm71, nma, mm41}.
finishing-tools = {universal-tools, mm53, cma}.
universal-tools = {nmg, mm52, rcmx}.
mm71            = {dnmm-71, tnmm-71, snmm-71, cnmm-71}.
mm41            = {tnmm-41, snmm-41, dnmm-41, cnmm-41}.
nma             = {tnma, dnma, snma, cnma}.
nmg             = {rnmg, tnmg, snmg, dnmg, cnmg}.
mm52            = {tcmm-52, dcmm-52, scmm-52, ccmm-52, rcmm-52}.
...
```

The domain of the tool systems (holders) can also be hierarchically structured: The names of the holders result from a projection of the relevant criteria from the ISO names of workpieces. For example, **tmaxp-PTL90** means: The holder type is **tmaxp**, the fixing system is **p**, the form of the cutting plate is **t**, the cutting direction is **l** (from right to left), and the tool-cutting edge-angle is **90** degrees.

```
holders = {tmax-p, tmax-u}.

tmax-p  = {pt, ps, pc, pr, pd}.
pt      = {tmaxp-PTL90, tmaxp-PTL80, tmaxp-PTL45, tmaxp-PTN60,
           tmaxp-PTR90, tmaxp-PTR80, tmaxp-PTR45}.
ps      = {tmaxp-PSL75, tmaxp-PSL45, tmaxp-PSN45, tmaxp-PSR75, tmaxp-PSR45}.
```

```
pc      = {tmaxp-PCL95, tmaxp-PCL75, tmaxp-PCN65, tmaxp-PCR95,
           tmaxp-PCR75, tmaxp-PCR65}.
pr      = {tmaxp-PRL30, tmaxp-PRL40, tmaxp-PRR30}.
pd      = {tmaxp-PDL93, tmaxp-PDR93}.

tmax-u  = {st, ss, sr, tmaxu-SCN95, tmaxu-SDL93}.
st      = {tmaxu-STL90, tmaxu-STL75, tmaxu-STN60, tmaxu-STN45,
           tmaxu-STR90, tmaxu-STR75}.
ss      = {tmaxu-SSR75, tmaxu-SSN45, tmaxu-SSL75, tmaxu-SSL45}.
sv      = {tmaxu-SVL93, tmaxu-SVN72, tmaxu-SVR93}.
sr      = {tmaxu-SRN, tmaxu-SRN30}.
...
```

The domain of cutting processes simply consists of two elements:

```
processes = {roughing, finishing}.
```

The following definitions describe a part of the structure of the domain of workpiece materials:

```
materials   = {steel, cast, alu}.
steel       = {building-steel, alloy-steel, stainless-steel}.
cast        = {gg, ggg}.
alloy-steel = {low-alloy-steel, high-alloy-steel}.
...
```

Having defined the domains of all variables, constraints can be defined over these domains.

The constraint **holder-tool** describes the cutting-plates fitting to several holders for reasons of geometry:

```
holder-tool(Holder:holders, Tool:lathe-tools) :-
   {(pt,tnma), (pt,tnmg), (pt,tcma-41),
    (ps,nmg), (ps,snma-41),
    (pc,cnmm-71), (pc,cnmm-41), (pc,cnmg), (pc,cnma),
    ...
    (ss,finishing-tool)}.
```

The constraint **process-material-tool** specifies the usability of cutting-plates w.r.t. the working-process to be done and the properties of materials. The constraint reflects the suitability of the cutting-plate materials (which are implicitly contained in their names) for certain workpiece materials, e.g. short-cutting, long-cutting, stainless, hard:

```
process-material-tool(Process:processes, Mat:materials, Tool:lathe-tools) :-
   {(roughing,steel,mm71),      (roughing,cast,mm71),
    (roughing,cast,nma),        (roughing,stainless-steel,mm41),
    (roughing,alloy-steel,nma), (roughing,low-alloy-steel,mm52),
    (processes,alu,nmg), (processes,steel,rcmx), (processes,cast,rcmx)}.
```

The constraint **tc-ea-al** gives expression to the requirement that the sum of the tool-cutting-edge angle, the edge-angle and the angle alpha must be less than 180 degrees.

```
tc-ea-al(TCEA:tc-edge-angles, EA:edge-angles, Alpha:acute-angles) :-
   lambda(TCEA, EA, Alpha . (180 > TCEA + EA + Alpha)).
```

Finally, a compound constraint **tool_sel** is defined that represents the conjunction of all relevant constraints for lathe-tool selection:

```
tool_sel(Holder,Tool,Plate,Process,Direction,Cutting,Material,
         Alpha,TC-Edge-Angle) :-
   {holder-tool(Holder,Tool),
    process-holder(Process,Holder),
    holder-description(Holder,Direction,TC-Edge-Angle,Plate),
    holder-cutting(Holder,Cutting),
    process-material-tool(Process,Material,Tool),
    plate-eangle(Plate,Edge-Angle),
    process-eangle(Process,Edge-Angle),
    tc-ea-al(TC-Edge-Angle,Edge-Angle,Alpha)}.
```

The knowledge represented in RELFUN describes the transformation of features into skeletal plans and, calling CONTAX' above-defined **tool_sel** constraint, their refinement into parameterized NC programs:

```
feat2nc() :-& insert-tools-in-skp(gen-skp(root-id())).
             % root-id returns the root feature identifier
```

Complex features are decomposed until skeletal plans can be retrieved for primitive features like **cylinder**:

```
gen-skp(Fid) :- cylinder(Fid, []) & [skp].
...
gen-skp(Fid) :-
        rshoulder(Fid, [ground[Lts], leftmost[Lshid], rightmost[Rshid]]),
        truncone(Lshid, [c1[X1]]),
        truncone(Rshid,
                 [c1[X2], c2[X3],
                  r1[Y1], r2[Yh]]) &
        merge-skp(
          [skp,
           dir[to-right], kind[lengthwise],
           seq[
            actions[
             [to-right, lengthwise,
              geo[p[Y1, X1], p[Y1, X2], p[Yh, X3]]]],
            actions[]]],
          gen-skp(Lts)).
```

These primitive skeletal plans are merged resulting in complex ones:

```
...
merge-skp([skp, dir[unfixed], kind[contour], com[A11, A12]],
          [skp, dir[Dir], Kind, seq[A21, A22]])
     :- fix-dir(Dir),
        seq[Na11, Na12] is sequentialize(com[A11, A12], Dir) &
        tup(skp, dir[Dir], kind[contour],
            tup(seq, append-act(Na11, A21), append-act(A22, Na12))).
...
```

Finally, the skeletal plan is being extended by inserting tool information obtained by constraint propagation:

```
insert-tools-in-skp(Skp) :-
         [skp, Dir, Kind,
          [Act-type, actions[ | Act1], actions[ | Act2]]] is Skp &
         tup(... insert-tools(appfun(Act1, Act2)) ... ).
...
insert-tools([Act | Actrest]) :-
         ...  tool-select([], [Act | Actrest], Dir, Kind, 0, 0) & ... .
```

The longest possible sequence of actions is being reduced by the non-deterministic `tool-select` function until an applicable tool can be found:

```
tool-select(Actlist1, [A21 | A22], Dir, Kind, Alpha-max, Beta-max) :-
         same-dir(A21, Dir) &
         tool-select(appfun(Actlist1, [A21]), A22, Dir, Kind,
                     once(compute-alpha-angles(Alpha-max, A21)),
                     once(compute-beta-angles(Beta-max, A21))).
tool-select(Actlist1, Actlist2, Dir, Kind, Alpha-max, Beta-max) :-
         Tool is cheapest()(contax-tool-select(Dir, Kind, Alpha-max, Beta-max)) &
         % cheapest is a global constant (currently the selection function first)
         [[[tool | Tool] | Actlist1], Actlist2].
```

Possible tool candidates for an action sequence are solutions of a propagation in the instantiated constraint net. The function `contax-tool-select` initializes the data needed by the constraint net:

```
contax-tool-select(Dir, Kind, Alpha-max, Beta-max) :-&
         cn-tool-sel(process(), Kind, wp-material(), quality(), % process etc. are
                     Alpha-max, Beta-max, Dir).                 % global constants
```

The function `cn-tool-sel` invokes the constraint propagation and returns a list of tool candidates, failing if no candidates can be found (`sol_of` is described in Section 4):

```
cn-tool-sel(Process, Cut-kind, Wp-material, Quality, Alpha, Beta, Direction) :-&
          get-cn-tools(       % select relevant data (holder and cutting plate)
            sol_of(tool_sel[Wp-material, Process, Cut-kind, Alpha, Beta,
                            Direction, lathe-tools, holders, plate-geometries,
                            edge-angles, tc-edge-angles]),
            tool, holder).
```

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-11
Susane Biundo, Dietmar Dengler, Jana Koehler:
Deductive Planning and Plan Reuse in a Command Language Environment
13 pages

RR-92-13
Markus A. Thies, Frank Berger:
Planbasierte graphische Hilfe in objektorientierten Benutzungsoberflächen
13 Seiten

RR-92-14
Intelligent User Support in Graphical User Interfaces:
1. InCome: A System to Navigate through Interactions and Plans
   Thomas Fehrle, Markus A. Thies
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
   Markus A. Thies, Frank Berger

22 pages

RR-92-15
Winfried Graf: Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

RR-92-16
Jochen Heinsohn, Daniel Kudenko, Berhard Nebel, Hans-Jürgen Profitlich: An Empirical Analysis of Terminological Representation Systems
38 pages

RR-92-17
Hassan Aït-Kaci, Andreas Podelski, Gert Smolka:
A Feature-based Constraint System for Logic Programming with Entailment
23 pages

RR-92-18
John Nerbonne: Constraint-Based Semantics
21 pages

RR-92-19
Ralf Legleitner, Ansgar Bernardi, Christoph Klauck: PIM: Planning In Manufacturing using Skeletal Plans and Features
17 pages

RR-92-20
John Nerbonne: Representing Grammar, Meaning and Knowledge
18 pages

RR-92-21
Jörg-Peter Mohren, Jürgen Müller
Representing Spatial Relations (Part II) -The Geometrical Approach
25 pages

RR-92-22
Jörg Würtz: Unifying Cycles
24 pages

RR-92-23
Gert Smolka, Ralf Treinen:
Records for Logic Programming
38 pages

RR-92-24
Gabriele Schmidt: Knowledge Acquisition from Text in a Complex Domain
20 pages

RR-92-25
Franz Schmalhofer, Ralf Bergmann, Otto Kühn, Gabriele Schmidt: Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations
12 pages

RR-92-26
Franz Schmalhofer, Thomas Reinartz, Bidjan Tschaitschian: Intelligent documentation as a catalyst for developing cooperative knowledge-based systems
16 pages

RR-92-27
Franz Schmalhofer, Jörg Thoben: The model-based construction of a case-oriented expert system
18 pages

RR-92-29
Zhaohui Wu, Ansgar Bernardi, Christoph Klauck: Skeletel Plans Reuse: A Restricted Conceptual Graph Classification Approach
13 pages

RR-92-30
Rolf Backofen, Gert Smolka
A Complete and Recursive Feature Theory
32 pages

RR-92-31
Wolfgang Wahlster
Automatic Design of Multimodal Presentations
17 pages

RR-92-33
Franz Baader: Unification Theory
22 pages

RR-92-34
Philipp Hanschke: Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35
Manfred Meyer:
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke:
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer:
An Alternative to Θ-Subsumption Based on Terminological Reasoning
9 pages

RR-92-40
Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41
Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42
John Nerbonne:
A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster: WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi:
Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51
Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52
Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-54
Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55
John Nerbonne, Joachim Laubsch, Abdel Kader Diagne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56
Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58
Franz Baader, Bernhard Hollunder: How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59
Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
14 pages

RR-92-60
Karl Schlechta: Defaults, Preorder Semantics and Circumscription
18 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist: Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi: An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

DFKI Technical Memos

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel: ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Busemann: Prototypical Concept Formation An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh:
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben: The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

DFKI Documents

D-92-07
Susanne Biundo, Franz Schmalhofer (Eds.):
Proceedings of the DFKI Workshop on Planning
65 pages

D-92-08
Jochen Heinsohn, Bernhard Hollunder (Eds.):
DFKI Workshop on Taxonomic Reasoning
Proceedings
56 pages

D-92-09
Gernod P. Laufkötter: Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes
86 Seiten

D-92-10
Jakob Mauss: Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken
87 Seiten

D-92-11
Kerstin Becker: Möglichkeiten der Wissensmodellierung für technische Diagnose-Expertensysteme
92 Seiten

D-92-12
Otto Kühn, Franz Schmalhofer, Gabriele Schmidt:
Integrated Knowledge Acquisition for Lathe Production Planning: a Picture Gallery
(Integrierte Wissensakquisition zur Fertigungsplanung für Drehteile: eine Bildergalerie)
27 pages

D-92-13
Holger Peine: An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis
55 pages

D-92-14
Johannes Schwagereit: Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM
98 Seiten

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht 1991
130 Seiten

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.):
UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.):
Kooperierende Agenten
78 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann:
Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch:
Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

D-92-28
Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-02
Gabriele Schmidt, Frank Peters, Gernod Laufkötter: User Manual of COKAM+
23 pages

COLAB: A Hybrid Knowledge Representation and Compilation Laboratory

Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer

RR-93-08
Research Report