Proceedings of the

ECAI 2010

Workshop on

Intelligent Engineering Techniques for Knowledge Bases (IKBET)

affiliated with the

19th European Conference on Artificial Intelligence

August 16-20, 2010

Lisbon, Portugal

Alexander Felfernig

Franz Wotawa (eds.)

Preface

Knowledge bases are encountered in AI areas such as configuration, planning, diagnosis, semantic web, game playing, and others. Very often the amount and arrangement of the elements in the knowledge base outstrips the capability of a knowledge engineer to survey them and to effectively perform extension and maintenance operations - a situation that triggers an enormous demand to extend and improve existing development and maintenance practices. This demand is further boosted by emerging Social Semantic Web applications that require the support of distributed engineering scenarios.

Intelligent development and maintenance approaches are of high relevance and have attracted lasting research and interest from industrial partners. This is demonstrated by numerous publications, for example, in the IJCAI, AAAI, ECAI, KCAP, CP, IUI, and ISWC conference and workshop series. These working notes include research papers that are dealing with various aspects of knowledge engineering including overviews of commercial and open source systems, automated testing and debugging, knowledge representation and reasoning, knowledge base development processes, and open research issues.

Eleven papers demonstrate the wide range of knowledge engineering research and the diversity and complexity of problems to be solved. Besides the contributions of research institutes we have received three industry-related contributions which show the relevance of intelligent knowledge engineering techniques in the commercial context. Best papers from the workshop have been invited to submit an extended version of the paper to the AICOM special issue on “Intelligent Engineering Techniques for Knowledge Bases”. This journal special issue will be available in Spring 2011.

Alexander Felfernig
Franz Wotawa

August 2010

Organization

Chairs

Alexander Felfernig, Graz University of Technology

Franz Wotawa, Graz University of Technology

Program Committee

Alexander Felfernig, Graz University of Technology

Dieter Fensel, University of Innsbruck

Gerhard Friedrich, University of Klagenfurt

John Gero, George Mason University

Albert Haag, SAP Germany

Dietmar Jannach, University of Dortmund

Tomi Männistö, Helsinki University of Technology

Monika Mandl, Graz University of Technology

Lars Hvam, Technical University of Denmark

Gregory Provan, Cork Constraint Computation Centre

Monika Schubert, Graz University of Technology

Steffen Staab, University of Koblenz

Gerald Steinbauer, Graz University of Technology

Markus Stumptner, University of South Australia

Juha Tiihonen, Helsinki University of Technology

Franz Wotawa, Graz University of Technology

Markus Zanker, University of Klagenfurt

Table of Contents

Modelling Product Families for Product Configuration Systems with Product Variant Master

Niels Henrik Mortensen, Lars Hvam, Anders Haug[1]

Abstract. This article presents an evaluation of applying a suggested method for modelling product families for product configuration based on theory for modelling mechanical products, systems theory and object-oriented modelling. The modelling technique includes a so-called product variant master and CRC-cards for modelling and visualising the parts and properties of a complete product family. The modelling techniques include:

- Customer, engineering and part views on the product assortment to model the properties, functions and structure of the product family. This also makes it possible to map the links between the three views.
- Modelling of characteristics of the product variants in a product family
- Modelling of constraints between parts in the product family
- Visualisation of the entire product family on a poster e.g. 1x2 meters

The product variant master and CRC-cards are means to bridge the gap between domain experts and IT-developers, thus making it possible for the domain experts (e.g. engineers from product development) to express their knowledge in a form that is understandable both for the domain experts and the IT-developers.

The product variant master and CRC-cards have currently been tested and further developed in cooperation with several industrial companies. This article refers to experiences from applying the modelling technique in three different companies. Based upon these experiences, the utility of the product variant master is evaluated.

Significance. Product configuration systems are increasingly used in industrial companies as a means for efficient design of customer tailored products. The design and implementation of product configuration systems is a new and challenging task for the industrial companies and calls for a scientifically based framework to support the modelling of the product families to be implemented in the configuration systems.

Keywords. Mass Customization, modularization, product modelling, product configuration, object-oriented system development.

1. INTRODUCTION

Customers worldwide require personalised products. One way of obtaining this is to customise the products by use of product configuration systems (Tseng and Piller, 2003, Hvam et al 2008).

A product configuration system is based on a model of the product portfolio. A product model can be defined as a model that describes a product’s structure, function and other properties as well as a product’s life cycle properties e.g. manufacturing, assembly, transportation and service (Krause, 1993, Hvam, 1999). A product model used as a basis for a product configuration system also includes a definition of the rules for generating variants in the product assortment (Hvam et al, 2008, Schwarze, 1996).

A product configuration system is a knowledge-integrated or intelligent product model, which means that the models contain knowledge and information about the products, and based on this is able to derive new specifications for product instances and their life cycle properties. Experiences from a considerable number of industrial companies have shown that often these product configuration systems are constructed without the use of a strict modelling technique.

As a result of this, many of the systems are unstructured and undocumented, and they are, therefore, difficult or impossible to maintain or develop further. Thus, there is a need to develop a modelling technique which can ensure that the product configuration systems are properly structured and documented, so that the configuration systems can be continually maintained and developed further.

In order to cope with these challenges, a technique for modelling product families for product configuration has been developed - which makes it possible to document the product configuration systems in a structured way.

This article evaluates the experiences from applying the suggested method (product variant master and CRC-cards) for modelling product families for product configuration systems. The suggested method is based on three theoretical domains:

- Object-oriented modelling (Bennett et al, 1999; Booch et al 1999; Felfernig; 2000, Hvam 1999)
- System theory (Skyttner 2005; Bertalanffy 1969)
- Modelling mechanical products (Hubka, 1988; Schwarze, 1996)

The theory for modeling mechanical products and systems theory are used for defining the structure in the PVM and the CRC-cards, and thereby the structure of the configuration system, reflecting the product families to be modeled and the user requirements of the configuration system.

[1] Centre for Product Modelling, Technical University of Denmark. www.productmodels.org, www.man.dtu.dk, www.sam.sdu.dk

2. MODELLING PRODUCT FAMILIES FOR PRODUCT CONFIGURATION SYSTEMS

2.1. The Product Variant Master

A company’s product range often appears to be large and have a vast number of variants. To obtain an overall view of the products, the product range is drawn up in a so-called product variant master (Hvam et al 2008).

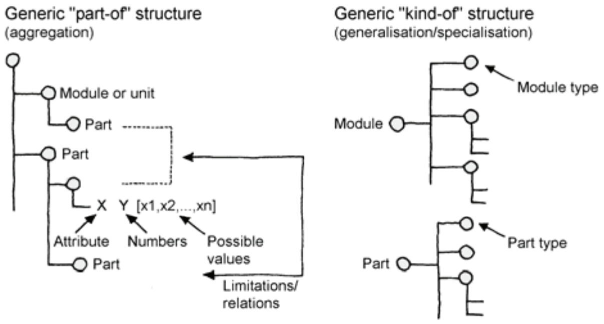


Figure 1. Principles of the Product Variant Master

A product variant master consists of two parts. The first of these, the “part-of” model (the left-hand side of the product variant master), contains those modules or parts which appear in the entire product family. For example, a car consists of a chassis, motor, brake system etc. Each module/part of the product range is marked with a circle. It is also possible to specify the number of such units used in the product - for example, 1 motor and 4 wheels in each car. The individual modules/parts are also modeled with a series of attributes which describe their properties and characteristics.

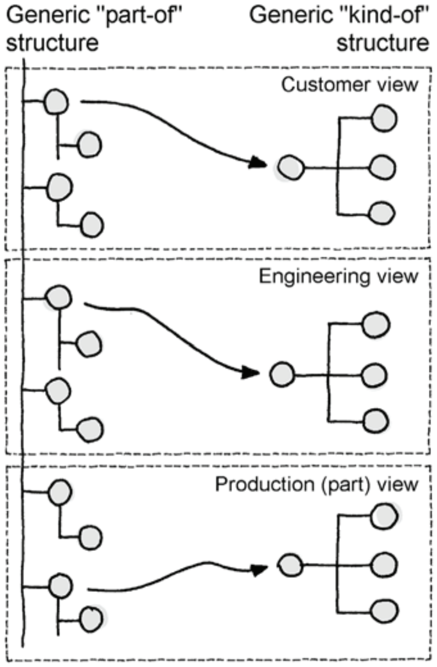


Figure 2. Structure of the Product Variant Master

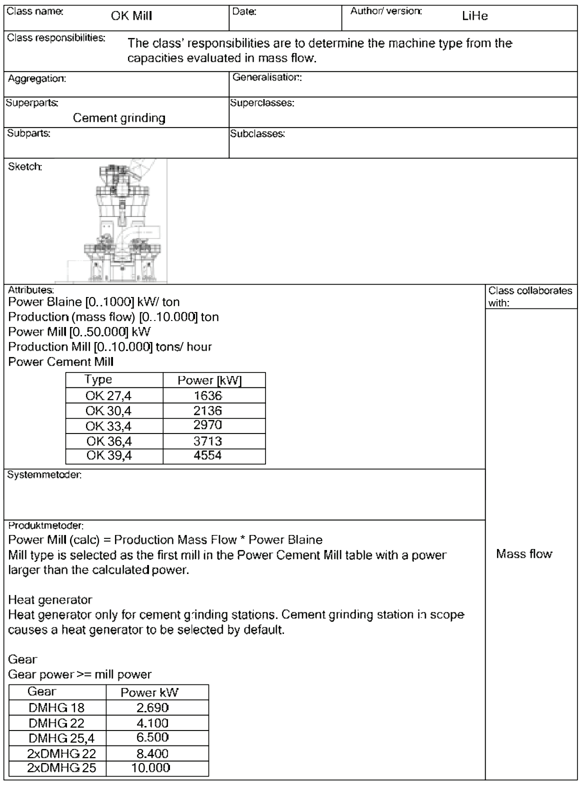
The other part of the product variant master (the right-hand side) describes how a product part can appear in several variants. A motor, for example, can be a petrol or diesel motor. This is shown on the product variant master as a generic structure, where the generic part is called the motor, and the specific parts are called petrol motor and diesel motor, respectively.

The two types of structure,”part of” and ”kind of”, are analogous to the structures of aggregation and specialization within object-oriented modelling.

The individual parts are also described with attributes, as in the part-of model. In the product variant master, a description is also given of the most important connections between modules/parts, i.e. rules for which modules/parts are permitted to be combined. This is done by drawing a line between the two modules/parts and writing the rules which apply for combining the modules/parts concerned. In a similar manner, the life-cycle systems to be modelled are described in terms of masters that for example describe the production system or the assembly system.

The individual modules/parts in the product variant master are further described on so-called CRC cards, which are discussed in more detail in the next section.

It is normally possible to create the product variant master in such a way that the products are described from the customer’s point of view at the top, followed by a description of the product range seen from an engineering point of view, while the products are described from a production point of view at the bottom as shown in figure 2.



Class name: OK Mill | Date: | Author/ version: LiHe

Class responsibilities: The class’ responsibilities are to determine the machine type from the capacities evaluated in mass flow.

Aggregation: | Generalisation:

Superparts: Cement grinding | Superclasses:

Subparts: | Subclasses:

Sketch:

Attributes:
Power Blaine [0..1000] kW/ ton
Production (mass flow) [0..10.000] ton
Power Mill [0..50.000] kW
Production Mill [0..10.000] tons/ hour
Power Cement Mill

Type	Power [kW]
OK 27,4	1636
OK 30,4	2136
OK 33,4	2970
OK 36,4	3713
OK 39,4	4554

Systemmetoder:

Produktmetoder:
Power Mill (calc) = Production Mass Flow * Power Blaine
Mill type is selected as the first mill in the Power Cement Mill table with a power larger than the calculated power.

Heat generator
Heat generator only for cement grinding stations. Cement grinding station in scope causes a heat generator to be selected by default.

Gear
Gear power >= mill power

Gear	Power kW
DMHG 18	2.690
DMHG 22	4.100
DMHG 25,4	6.500
2xDMHG 22	8.400
2xDMHG 25	10.000

Class collaborates with:
Mass flow

Figure 3. CRC-card

In connection with the specification of the product variant master and the final object-oriented class model, CRC cards are used to describe the individual object classes (Bellin, 1997), (Hvam et al., 2003), (Hvam et al, 2008). CRC stands for “Class, Responsibility, and Collaboration”. In other words, this is where a description is made of what defines the class, including the class' name, its possible place in a hierarchy, together with a date and the name of the person responsible for the class. In addition, the class' task (responsibility), the class' attributes and methods, and with which classes it collaborates (collaboration) are given.

Figure 3 shows an example of a CRC card. In connection with the modelling of products, a sketch of the product part is added, with a specification of the attributes of the product part.

The CRC cards are filled in gradually during the object-oriented analysis. The CRC cards can be associated with both the product variant master and the OOA model. The purpose of the CRC cards is to document detailed knowledge about attributes and methods for the individual object classes, and to describe the classes' mutual relationships. The CRC cards serve as documentation for both domain experts and system developers, and thus, together with the product variant master and the class diagram, become an important means of communicating and documenting knowledge within the project group.

A more detailed description of the individual fields on the CRC cards is as follows:

Class name:

The CRC card is given a name that is unique, so that the class can be identified in the overall structure.

Date:

Each card is dated with the date on which the card was created, and a date for each time the card is revised.

Author/ version:

The person who created the card and/or has the responsibility for revising the card and the card's version number.

Responsibilities:

This should be a short text describing the mission of the class. This makes it easier to get a rapid general view of what the class does.

Aggregation and generalization:

The class' position in relation to other classes is described by specifying the class' place in generalization-specialization structures, respectively, or aggregation structures. This is done by describing which superclasses or subclasses are related to the class within either a generalization-specialization hierarchy or an aggregation hierarchy. Generalization categorises classes with common properties from general to specific classes in the so-called generalization-specialization hierarchies, also known as inheritance hierarchies, because the specialized classes further down in the hierarchy "inherit" general properties from the general (higher level) classes.

The other type of aggregation is a structure in which a higher-level class (the whole) consists of a number of subordinate classes (the parts). Using ordinary language, decomposition of an aggregation structure can be expressed by the phrase "has a" and aggregation by "is part of".

Sketch:

When working with product descriptions, it is convenient to include a sketch, which in a concise and precise manner describes the attributes included in the class. Geometric relationships are usually easier to explain by using a sketch/drawing than by using words.

Class attributes:

The various parameters such as height-interval, width-interval etc., which the class knows about itself, are described in the field "Attributes". Attributes are, as previously mentioned, described by their names, datatype (if any), range of values and units (for example, Length, [Integer], [1..10] mm). It is often convenient to express attributes in table form.

Class methods:

What the class does (for example, calculation of an area) is placed in the field of "Methods", which, as stated, can be divided into system methods and product methods.

Methods can be described in a natural language with the help of tables, with a pseudo code, by using a formal notation such as Object Constraint Language (OCL) (Warmer et al., 1999), which is a standard under UML, or by using the notation from individual items of configuration software.

3. CASE STUDY

3.1. The Industry Applications

The Product Variant Master has been applied in several manufacturing companies. In the following we shall give a brief introduction to three of those companies and their configuration projects, and we will discuss the experiences and lessons learned by using the product variant master and CRC-cards for modelling product families for product configuration systems.

Company A

Company A is an engineering and industrial company with an international market leading position within the area of development and manufacturing of cement plants. The company has a turnover around 1 billion USD.

A modern cement plant typically produces 2-10,000 tonnes of clinkers per day (TPD), and the sales price for a 4,000 TPD plant is approx. 100 million USD. Every complete cement plant is customized to suit the local raw material and climatic conditions, and the lead-time from signing the contract to start-up is around 2½ years.

The company has initiated a project on the development of a product configuration system for the selection of the main machines of the factory and the overall determination of capacity, emissions etc. based on 2-300 input parameters on e.g. raw materials, types and qualities of finished goods, geographical conditions, energy supply etc. The configuration system is meant to be used in the early sales phase for making budget quotations, including an overall dimensioning of the cement factory, a process diagram and a price estimate.

In the product analysis, the cement factory was divided into 9 process areas. Based on the process areas the model was structured hierarchically, starting with scope list and mass flow leading to the selection of solution principles described as arrangements and lists of machines. The product analysis was carried out by using the product variant master and the CRC-cards. The product variant master proved to be a useful tool for modelling the overall cement milling processes and machine parts of the cement factory. The product variant master was built up through a series of meetings between the modelling team and the product experts at the company. The detailed attributes and constraints in the model were described on the CRC-cards by the individual product experts.

The product variant master and the CRC-cards formed the basis for making a formal object-oriented model of the configuration system. Based on the product variant master an object-oriented class structure was derived and the CRC-cards were checked for inconsistency and logical errors. The product variant master was drawn by using MS-Visio, the CRC-cards were typed in MS-Word documents. The product configuration system was implemented in an object-oriented standard configuration software “iBaan eConfiguration Enterprise.” The OOA model was derived by the programmer of the configuration system based on the PVM and the CRC-cards. The CRC-cards both refer to the product variant master and the object-oriented class model, which represents the exact structure of the product configuration system in the “iBaan eConfiguration Enterprise.” The experiences are summarised in table 1 below.

Table 1. PVM and CRC-cards in company A

Modelling tool for domain experts – how the PVM was received by the domain experts	The domain experts were engineers from product design and project engineering. The domain experts could easily learn the modelling techniques for the purpose of evaluation of the models and input of information. The modelling was carried out by the configuration team and external consultants.
The possibilities for structuring product knowledge	The customer, engineering and part views have proved to be useful when modelling these products of high complexity. Structuring the PVM requires knowledge on domain theory for modelling mechanical products.
Visualisation and decision making on the product families in focus	The Product Variant Master proved to be able to capture and visualise the product assortment in this case on a high level of abstraction. Need to control the discussions to avoid non relevant details in the model. Important to involve the right domain experts in order to make decisions needed on the product variants to include in the configuration system.
PVM as a basis for programming the configuration system	Based on the PVM an OOA-model was set up to be used as a basis for implementing and maintaining the configuration software. Easy to transform the PVM to a formal OOA model.
PVM and CRC cards as documentation when maintaining and updating the configuration system	The PVM is only used in the analysis phase and is not maintained in the phase of running the configuration system. The OOA-model is currently being updated and used in the running phase of the system. The company has over the last 10 years increased the level of documentation and searched for IT-systems to manage the documentation task.

The product configuration system is currently being updated and further developed by a task force who is responsible for updating and further developing the model and the configuration system in cooperation with the product specialists. Maintaining the documentation in Visio and Word has proved to be a tedious work, as the same attributes and rules will have to be updated in several systems. In order to improve this, the company has during the last year implemented an IT-system for documentation, which means that the rules and attributes now only will have to be entered once into the documentation system.

The project at Company A has proven that the use of the three views in PVM can be useful when modelling complex products. Finally, the configuration system has been implemented in two different product configuration systems. The first configuration system was not object-oriented, while the second configuration system (“iBaan eConfiguration Enterprise”) was fully object-oriented. Applying an object-oriented system made it considerably easier to implement and maintain the product configuration system.

3.1.1 Company B

Company B is an international engineering company that has a market leading position within the area of design and supply of spray drying plants. The company is creating approx. 340 mio. USD in turnover a year. The products are characterised as highly individualised for each project.

The configuration system was implemented in 2004 at company B and is in many ways similar to the configuration system at company A. The project focuses on the quotation process. During the development of the product model the need for an effective documentation system has emerged. Early in the project it was decided to separate the documentation system from the configuration software due to the lack of documentation facilities in the standard configuration systems. Lotus Notes Release 5 is implemented throughout the company as a standard application, and all the involved people in the configuration project have the necessary skills to operate this application. The documentation tool is, therefore, based on the Lotus Notes application.

The documentation system is built as a hierarchical file sharing system. The UI is divided into two main parts, the product variant master and the CRC cards. However, the product variant master is used in a different way. Only the whole-part structure of the product variant master is applied in the documentation tool. Main documents and responses are attached to the structure of the product variant master. The configuration system is implemented in Oracle Configurator. Since the standard configuration software does not provide full object orientation, the CRC cards described in section xx has been changed to fit the partly object-oriented system. The fields for generalization and aggregation have been erased. The aggregation relations can be seen from the product variant master and generalization is not supported.

To ease the domain expert’s overview, an extra field to describe rules has been added. In this way, methods (does) have been divided into two fields, does and rules. “Does” could be e.g. print BOM while “Rules” could be a table stating valid variants. The CRC Card is divided into three sections. The first section contains a unique class name, date of creation and a plain language text explanation of the responsibility of the card. The second section is a field for sketches. This is very useful when different engineers need quick information about details. The sketch describes in a precise manner the attributes that apply to the class. The last section contains three fields for knowledge insertion and collaborations. Various parameters such as height, width etc., which the class knows about itself, are specified in the “knows” field. The “knows” field contains the attributes and the “rules” field describes how the constraints are working. The “does” field describes what the class does, e.g. print or generate BOM. Collaborations specify which classes collaborate with the information on the CRC card in order to perform a given action.

The documentation system has been in use throughout the project period. As mentioned, the documentation system was created by using standard Notes templates. This gives limitations according to functions of the application. Class diagrams are not included in the documentation tools. They must be drawn manually. Implementing class diagrams would demand a graphical interface, which is not present in the standard Lotus Notes application. Table 2 below lists the main findings from company A.

Table 2. PVM and CRC-cards in company B

Modelling tool for domain experts – how the PVM was received by the domain experts	The modelling is carried out by the configuration team and reviewed by the domain experts/ engineers. However, the domain experts contributed with parts of the model reflecting their special field of competence. Easy for the domain experts to learn to read and review the PVM and CRC-cards.
The possibilities for structuring product knowledge	The model includes complex and highly engineered products. The three views in the PVM were used to clarify the interdependencies between customer requirements, main functions in the products and the Bill of material (BOM) structure. These interdependences were expressed as rules and constraints in the configuration model.
Visualisation and decision making on the product families in focus	The PVM was used to determine the preferred solutions to enter into the configuration system. The links between the three views provided insight into the consequences of delimiting which customer requirements to meet main functions and the number of main BOM's to include in the configuration system. The sketches on the CRC-cards have proved to be very useful in the communication with the domain experts.
PVM as a basis for programming the configuration system	The PVM and the CRC-cards were used as documentation for programming the configuration system. No formal OOA model was made.
PVM and CRC cards as documentation when maintaining and updating the configuration system	The PVM was transferred into a file structure in the company's file share system (Lotus Notes). The CRC-cards are stored and currently updated in Lotus notes. Domain experts are responsible for updating the individual CRC-cards.

Lessons learned from the project at Company B were that an IT-based documentation tool is necessary in order to secure an efficient handling of the documents in the product model (product variant master and CRC-cards). The configuration system is implemented in Oracle Configurator, which is not a fully object-oriented configuration system. This means that e.g. inheritance between object classes in the product configuration system is not possible. The company uses a variant of the product variant master, where only the whole part structure on the right side of the product variant master is applied. However, the experiences from the project are that the revised product variant master and the CRC-cards still secure a structure and documentation of the system.

3.1.2 Company C

Company C produces data centre infrastructure such as uninterruptible power supplies, battery racks, power distribution units, racks, cooling equipment, accessories etc. The total turnover is approx. 4 billion USD (2008). The company has applied the PVM and CRC-cards since 2000. Today, the company has 8-9 product configuration systems. The company has formed a configuration team with approx. 25 employees situated in Kolding, Denmark. The configuration team is responsible for development and maintenance of the product configuration systems, which are used worldwide.

Table 3. PVM and CRC-cards in company C

Modelling tool for domain experts – how the PVM was received by the domain experts	The PVM is used for modelling the product families in a cooperation between the configuration team and engineers from product development
The possibilities for structuring product knowledge	Relative to company A and B this company has a more "flat" structure in the PVM and configuration system, meaning that the level of structuring is lower than company A and B. The PVM and CRC-cards is set up by the configuration team and afterwards discussed with product development.
Visualisation and decision making on the product families in focus	The product configuration systems are set up after the product development project is completed. The PVM and CRC-cards lead to clarification of decisions on product variants that haven't been made in the development project.
PVM as a basis for programming the configuration system	The configuration system is programmed based on an object-oriented model made from the PVM and on the CRC-cards.
PVM and CRC cards as documentation when maintaining and updating the configuration system	Only the CRC-cards are used for maintenance and update of the product configuration systems. The CRC-cards are stored and currently updated in Lotus notes. The configuration team members are responsible for updating the individual CRC-cards.

The product assortment is modeled in a co-operation between the configuration team and the product development teams. The product variant master and the CRC cards are used in the modeling process and document the configuration systems throughout programming and maintenance of the product configuration systems. Similar to Company B, company C has developed a Lotus Notes based documentation tool – called the CRC-card database - to handle the documentation of the models. The configuration systems are implemented in Cincom Configurator, which is a rule based configuration system.

Lessons learned from company C are that the need for an IT-based documentation tool is even bigger at this company, than at Company B. Running a configuration team with 25 employees, which have to communicate with product development teams around the world, requires a structured procedure for building the product configuration systems, as well as a Web-based documentation tool, which can be accessed by employees worldwide. At company C, a Notes based documentation tool has been developed similar to the Notes application at company B. The documentation system has now been running for 6 years. The experiences from running the documentation system is that the structure in the product variant master and the CRC-card database form a solid basis for communicating with e.g. product designers and for maintaining and further developing the product configuration systems. However, the fact that the documentation system is separated from the configuration software means that the attributes and rules have to be represented in both the documentation tool and in the configuration with only limited possibilities of relating the two systems to one another. This means that the configuration team at company C needs to be disciplined about updating the documentation system every time a change is made in the configuration system.

4. CONCLUSION

The proposed modelling techniques are based on well-known and proven theoretical elements; theory for modelling mechanical products, systems theory and object-oriented modelling. The aim of the product variant master and CRC-cards are to serve as a tool for engineers and programmers working with design, implementation and maintenance of product configuration systems. The experiences from applying the procedure in the above mentioned three industrial companies show that the product variant master and CRC-cards contribute to define, structure and document the product configuration systems. The product variant master and the CRC-cards make it possible to communicate and document the product assortment, including rules and constraints on how to design a customer tailored product.

The product variant master is developed based on the basic principles in object-oriented modelling and programming. However, only a very small part of the standard configuration systems based on rules and constraints and including an inference engine are fully object-oriented. In order to meet this actual situation the product variant master and the CRC cards have been changed to fit into configuration systems, which are not fully object-oriented. The experiences from applying the procedure in building configuration systems in non object-oriented systems are positive. However, structuring and maintaining configuration systems are considerably easier in an object-oriented standard configuration system. Furthermore, an integration of the documentation tool and the configuration systems would ease the work of design, implementation and maintenance of product configuration systems considerably.

5. REFERENCES

Bellin, D. & Simone, S. S. (1999). The CRC-card Book. Addison-Wesley.

Bennett, S., McRobb, S. & Farmer, R. (1999). Object-Oriented Systems Analysis and Design using UML. McGraw-Hill.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). The Unified Modeling Language User Guide. Addison-Wesley.

Bertalanffy L.; General System Theory; 1996.

Felfernig, A., Friedrich, G. E. & Jannach, D. (2000). UML as Domain Specific Language for the Construction of Knowledge-based Configuration Systems. In International Journal of Software Engineering and Knowledge Engineering, volume 10/4.

Hubka, V.; Theory of technical Systems, Springer 1988.

Hvam, L. (1994). Application of product modelling – seen from a work preparation viewpoint. Ph.D. Thesis. Dept. of Industrial Management and Engineering, Technical University of Denmark.

Hvam, L. (1999). A procedure for building product models. Robotics and Computer-Integrated Manufacturing, 15: 77-87.

Hvam, L. & Malis, M. (2002). A Knowledge Based Documentation Tool for Configuration Projects. In Mass Customization for Competitive Advantage (Ed.: M.M. Tseng and F.T. Piller).

Hvam, L. & Riis, J. (2003). CRC-card for product modeling. Computers in Industry, volume 50/1.

Hvam L., Mortensen N.H., Riis J.; Product Customization, Springer 2008.

Jackson, P. (1999). Introduction to Expert Systems. 3rd edition. Addison-Wesley.

Krause, F.L., Kimura, F. & Kjellberg, T. (1993). Product Modelling. In Annals of the CIRP, volume 42/2.

Schwarze, S. (1996). Configuration of Multiple-Variant Products. BWI, Zürich.

Skyttner L.; General Systems Theory, 2005.

Tiihonen, J., Soininen, T., Männistö, T. & Sulonen, R. (1996). State-of-the-practice in Product Configuration – a Survey of 10 Cases in the Finnish Industry. In Knowledge Intensive CAD, volume 1 (Ed.: Tomiyama, T., Mäntylä, M. & Finger S.). Chapman & Hall, pp. 95-114.

Tseng Mitchell M. and Piller, Frank. T. eds(2003); The Customer Centric Enterprise – Advances in Mass Customization and Personalization; Springer Verlag. ISBN 3-540-02492-1.

Warmer J., Kleppe A; The Object Constraint Language; Addison Wesley, 1999.

Whitten, J.L, Bentley, L.D. & Dittman, K.C. (2001). Systems Analysis and Design Methods. Irwin/McGraw-Hill, New York, USA.

Sigma: An Integrated Development Environment for Logical Theories

Adam Pease[1], Christoph Benzmüller[1]

Abstract. Sigma[1,2] is an open source environment for the development of logical theories. It has been under development and regular release for nearly a decade, and has been the principal environment under which the SUMO[3] has been created. We discuss its features and evolution, and explain why it is an appropriate environment for the development of expressive ontologies in first and higher order logic.

1 INTRODUCTION

There have been many environments created to support ontology development[25]. The majority, at least in recent years, have been to support creation of lightweight ontologies or taxonomies in the OWL language.

There are a limited number of language constructs in a frame-based or description-logic language. Frames have class membership and slots. Slots can have values and restrictions. The primary language construct is the taxonomy, which lends itself easily to tree-based views and editors. This is similar to object oriented language IDEs that typically have tree views for the hierarchy, and may have visual editors that allow the user to quickly create shells of code, based on the object taxonomy. Many ontology developers start by developing their products in a lightweight ontology editor that handles frame-based languages. Ontology developers who are used to that paradigm may wonder why Sigma does not offer an editing component as the primary method for developing ontologies Most modern software engineering however takes place in a text editor. Tools are an important part of the development process, and can help improve both productivity and quality. But the complexity of a modern programming language prevents modern software development from being reduced to simple forms entry and visual editors.

Modern and expressive languages for the development of formal theories, such as SUO-KIF[19] and TPTP[14] have a similar degree of expressiveness, in a broad sense, to a modern programming language. For that reason, we believe that the appropriate role for a knowledge engineering environment is in browsing, inference, analysis and other functions, rather than, at least primarily, authoring and editing.

There is promise in creating editing modes for text editors appropriate for knowledge engineering[26]. One challenge however is that the choice of a text editor, is, for a professional programmer, a very personal, and often a very strongly held preference. To the extent that knowledge engineers are also programmers, it will be difficult to create any environment so compelling that it will cause them to switch text editors. One alternative would be to capture just a portion of the "market" by working to add appropriate modes to just one text editor. Another would be to apply very significant resources, that do not appear yet to exist in the marketplace, to create modes in several powerful editors. For these reasons also, we have focused on tools other than text editing modes.

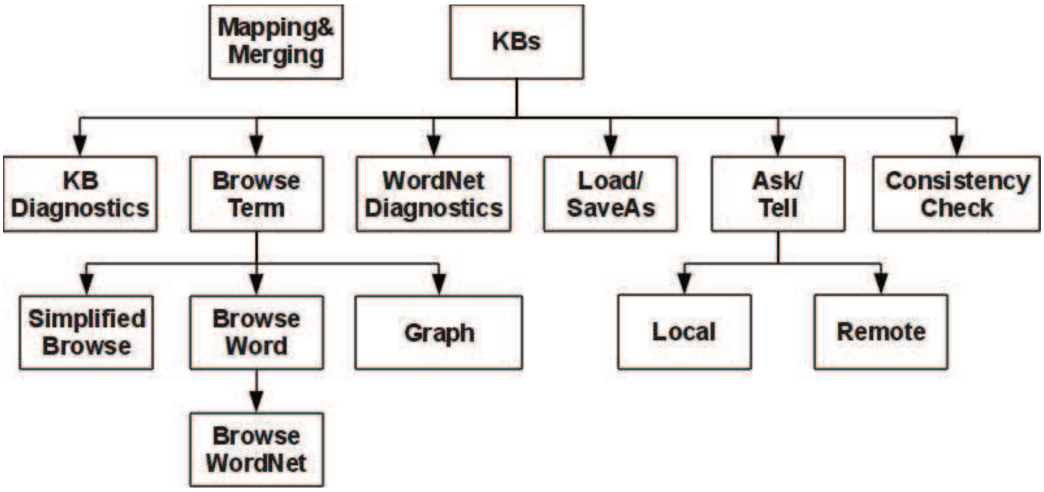


Figure 1: Major Sigma Functions

[1] Articulate Software, Angwin, CA. Email: {apease, cbenzmueller}@articulatesoftware.com.
The second author is currently supported by the German ResearchFoundation (DFG) under grant BE 2501/6-1.

Also in keeping with the modern software development model, we have utilized the Concurrent Version System (CVS) for collaborative ontology development. Developers are typically given authority over one or more ontologies, required to check in progress at least weekly so that other developers can sync up with their changes. This has also resulted in a detailed public record of the development and evolution of the Suggested Upper Merged Ontology (SUMO) [22]

While Sigma was created to support SUMO, that is by no means the only theory that it can handle. Sigma works on knowledge bases that can be composed from various files selected by the user. Those files can be coded in a small number of different formal languages, including OWL, as well as SUO-KIF. The Sigma user can easily work with very small theories or very large ones by composing only the theories that are needed for the work at hand. A typical use of Sigma would involve loading just the upper level of SUMO and whatever domain theory is needed for the user's chosen application area.

Tools within Sigma (Figure 1) can be broadly segmented into several groups, (1) browsing and display, (2) analysis and debugging, (3) inference, and (4) mapping, merging and translation. We describe each of these topics in the following sections, but first give a very brief introduction to the SUMO, which is the logical theory Sigma was initially developed to support.

2 SUMO

The Suggested Upper Merged Ontology [3,10] began as just an upper level ontology encoded in first order logic. The logic has expanded to include higher order elements. SUMO itself is now a bit of a misnomer as it refers to a combined set of theories: (1) SUMO "proper", the original upper level, consisting of roughly 1000 terms, 4000 axioms and including some 750 rules. (2) A MId-Level Ontology (MILO) of several thousand additional terms and axioms that define them, covering knowledge that is less general than those in SUMO. We should note that there is no objective standard for what should be considered upper level or not. All that can be said (simplistically) is that terms appearing lower in a taxonomy (more specific) are less general than those above. To avoid pointless argument about what constitutes an "upper level" term, we simply try to keep SUMO about 1000 terms with their associated definitions, and any time content is added, the most specific content, as measured by its having the lowest level in the subclass hierarchy, is, if necessary, moved to MILO or a domain ontology. (3) There are also a few dozen domain ontologies on various topics including theories of economy, geography, finance and computing. Together, all ontologies total roughly 20,000 terms and 70,000 axioms. We might also add a fourth group of ontologies which are theories that consist largely of ground facts, semi-automatically created from other sources and aligned with SUMO. These include YAGO[4], which is the largest of these sorts of resources aligned with SUMO.

SUMO has been mapped by hand to the WordNet lexicon[5]. Initially each term in SUMO proper was mapped and in later phases all WordNet synsets appearing above a frequency threshold in the Brown Corpus[7,8] were mapped to a roughly equivalent term in SUMO's lower level ontologies. If a rough equivalent didn't exist, one was created and defined. One caveat is that some words in English are vague enough to defy logical definition, so some such words still lack direct equivalences.

SUMO proper has a significant set of manually created language display templates that allow terms and definitions to be paraphrased in various natural languages, including non-western character sets. These include Arabic, French, English, Czech, Tagalog, German, Italian, Hindi, Romanian, Chinese (traditional and simplified characters). Automatically generated natural language paraphrases can be seen in the rightmost column of the screen display given as Figure 2.

Take for example that we have the SUO-KIF statement that **(authors Dickens OliverTwistBook)**. We have the following statements that have been coded to support the paraphrasing of statements with the authors relation.

```
(format EnglishLanguage authors "%1 is %n the &%author of %2")
(format it authors "%1 è l' &%autore di %2")
```

If a Sigma user has loaded this information in a knowledge base, and English is selected as the presentation, the user will see "Dickens is the author of Oliver Twist." next to the SUO-KIF statement. If Italian is selected, the paraphrase will be "Dickens è l'autore di Oliver Twist". Arguments to predicates are recursively substituted for the %1, %2 etc parameter variables, allowing much larger expressions to be constructed from more complex logical expressions.

The Global WordNet effort [6,9] links lexicons in many languages, following the same model of computational lexicon development as the original English WordNet. Wordnets have now been developed for some 40 languages. This rich set of cross-linguistic links that includes SUMO has the promise of being the basis for much work in language translation and linguistics generally. A simple idea for taking advantage of some of this work would be to expand the set of language translations for individual terms available for SUMO.

SUMO is defined in the SUO-KIF language[19], which is a derivative of the original Knowledge Interchange Format[20].

When we speak in this paper about a "formal theory", we mean a theory, such as SUMO, in which the meaning of any term is given only by the axioms in a logical language that use that term. In contrast, in an informal ontology, terms must be understood by recourse to human intuitions or understandings based on natural language term names, or natural language definitions.

3 BROWSING and DISPLAY

Sigma was originally just a display tool. Its original, and still most heavily used function is for creating hyperlinked sets of formatted axioms that all contain a particular term (Figure 1). Clicking on a term in turn gives a hyperlinked display of all the axioms that contain the new term. Next to each axiom is given the file and line(s) where the axiom was written. Also, shown is an automatically generated natural language paraphrase of each axiom. While the language generation component is relatively rudimentary, it gains significant power when tied to a rich ontology, in this case, SUMO. Much productive work remains to extend the functionality of this component to take into account the latest work in language generation. In particular, significant improvement would come from natural use of prepositions in paraphrasing statements about actions and the participants in actions.

In 2008 we added a simplified browser view that may be more appropriate for users who are transitioning from use of frame and description logic languages. It gives prominence to a tree view of the subclass hierarchy and presents binary relations in a simple tabular format, relegating rules to an area lower in the browser pane, and rendering them in the natural language paraphrase form only.

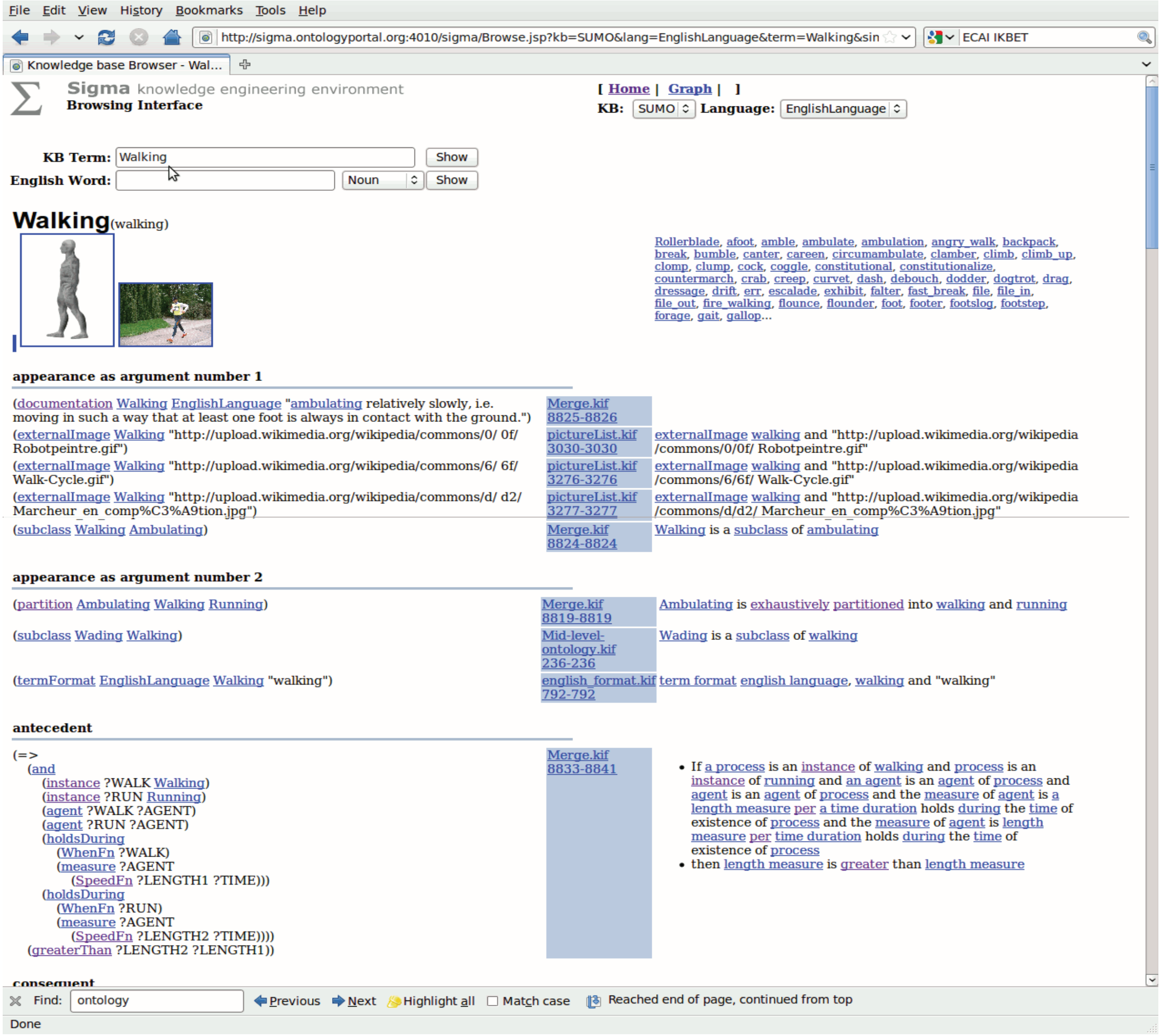
Sigma includes a tree browser display. In contrast to many ontologies developed in frame languages, SUMO has several hierarchies that can be used to organize and display the theory. These include hierarchies of physical parts, relations, attributes, processes and others. As such, the tree browser allows the user to select any transitive binary relation as the link by which the hierarchy display is created.

4 ANALYSIS and DEBUGGING

Sigma includes a number of specialized and general tools for ensuring ontology quality. The ultimate tool for quality checking on a formal ontology is theorem proving. However, there is no escape from the reality that on first- and higher-order theories, a theorem prover is not guaranteed to find all contradictions that exist. So in a practical system, there must be a combination of special purpose tests that are complete, and general purpose testing which is incomplete.

We will discuss theorem proving in the following section, so in this section we describe the various special case tests that we have found to be useful, and included in Sigma. While the number of possible tests is potentially infinite, there are a number of common problems that result from errors that are easy to make. The special case tests aim to cover these most common cases.

The SUMO-WordNet mappings also offer the opportunity to find problems exposed by differences in the two products. We believe that the two hierarchies should necessarily be isomorphic. A formal theory is a human engineered product, largely free of redundancy, and which can be edited to remove any kind of bias that is recognized by the developers. A formal theory can also contain concepts which are not lexicalized in any language. This is especially valuable at the upper levels, in which linguistic elements are so vague or ambiguous they cannot serve as a direct model for formalization. Being able to create new terms at will, when needed to formalize important notions in


File Edit View History Bookmarks Tools Help
ECAI IKBET
Knowledge base Browser - Wal...
Sigma knowledge engineering environment
Browsing Interface
[Home | Graph |]
Show
Show
Previous
Next
Highlight all
Match case

http://sigma.ontologyportal.org:4010/sigma/Browse.jsp?kb=SUMO&lang=EnglishLanguage&term=Walking&sin

KB: SUMO Language: EnglishLanguage

KB Term: Walking

English Word: Noun

Walking(walking)

Rollerblade, afoot, amble, ambulate, ambulation, angry_walk, backpack, break, bumble, canter, careen, circumambulate, clamber, climb, climb_up, clomp, clump, cock, coggle, constitutional, constitutionalize, countermarch, crab, creep, curvet, dash, debouch, dodder, dogtrot, drag, dressage, drift, err, escalade, exhibit, falter, fast_break, file, file_in, file_out, fire_walking, flounce, flounder, foot, footer, footslog, footstep, forage, gait, gallop...

appearance as argument number 1

(documentation Walking EnglishLanguage "ambulating relatively slowly, i.e. moving in such a way that at least one foot is always in contact with the ground.")	Merge.kif 8825-8826	
(externalImage Walking "http://upload.wikimedia.org/wikipedia/commons/0/ 0f/ Robotpeintre.gif")	pictureList.kif 3030-3030	externalImage walking and "http://upload.wikimedia.org/wikipedia /commons/0/0f/ Robotpeintre.gif"
(externalImage Walking "http://upload.wikimedia.org/wikipedia/commons/6/ 6f/ Walk-Cycle.gif")	pictureList.kif 3276-3276	externalImage walking and "http://upload.wikimedia.org/wikipedia /commons/6/6f/ Walk-Cycle.gif"
(externalImage Walking "http://upload.wikimedia.org/wikipedia/commons/d/ d2/ Marcheur_en_comp%C3%A9tion.jpg")	pictureList.kif 3277-3277	externalImage walking and "http://upload.wikimedia.org/wikipedia /commons/d/d2/ Marcheur_en_comp%C3%A9tion.jpg"
(subclass Walking Ambulating)	Merge.kif 8824-8824	Walking is a subclass of ambulating

appearance as argument number 2

(partition Ambulating Walking Running)	Merge.kif 8819-8819	Ambulating is exhaustively partitioned into walking and running
(subclass Wading Walking)	Mid-level-ontology.kif 236-236	Wading is a subclass of walking
(termFormat EnglishLanguage Walking "walking")	english_format.kif 792-792	term format english language, walking and "walking"

antecedent

```
(=>
  (and
    (instance ?WALK Walking)
    (instance ?RUN Running)
    (agent ?WALK ?AGENT)
    (agent ?RUN ?AGENT)
    (holdsDuring
      (WhenFn ?WALK)
      (measure ?AGENT
        (SpeedFn ?LENGTH1 ?TIME)))
    (holdsDuring
      (WhenFn ?RUN)
      (measure ?AGENT
        (SpeedFn ?LENGTH2 ?TIME))))
  (greaterThan ?LENGTH2 ?LENGTH1))
```

Merge.kif 8833-8841

- If a process is an instance of walking and process is an instance of running and an agent is an agent of process and agent is an agent of process and the measure of agent is a length measure per a time duration holds during the time of existence of process and the measure of agent is length measure per time duration holds during the time of existence of process
- then length measure is greater than length measure

consequent

Find: ontology

Reached end of page, continued from top

Done

Figure 2: Sigma browsing screen

the world, is an important characteristic of a formal theory, and makes it possible to have constructs which are clear, and efficient for representation as well as inference.

There are two special case tests for errors. We test for terms without a root in the subclass hierarchy at the term Entity, which is the topmost term in SUMO. This commonly results from either omitting a subclass or instance statement when defining a new term, or by misspelling the name of the intended parent term. The second special case test is for where two terms have parents that are defined to be disjoint. In a large theory like SUMO, it can be easy to lose track of this case, especially when the ultimately conflict may be between terms that are many levels up in the subclass hierarchy.

There are also a number of tests for cases that are indicative of a problem, yet not strictly an error that would result in a logical contradiction. The first of these is for terms lacking documentation. In theories under construction, theories that are the results of importing and merging another ontology, or simply for large lists of domain instances, like city names, it may be reasonable, temporary, or expected for such terms to lack documentation. But this does often reflect an outright error, where a term name was simply misspelled in the documentation definition, or in some other axiom.

We test for cases where terms do not appear in any rules. This again is common in collections of instance-level facts, but undesirable for many terms, where it should be possible to define precisely the intended meaning of the term with a small number of formal rules, as well as statements like class membership.

Because knowledge bases are often composed from SUMO's general and domain specific component ontologies, it is desirable to limit dependencies among the files as much as possible. For that reason we include a tool to specify dependencies between pairs of files. It is typically most desirable at least to ensure that dependencies are only from one file to another, and not between both files. All domain files will of course depend at least upon SUMO proper, since they form a single integrated theory that is decomposed into separate files for convenience and efficiency of inference.

Diagnostics are provided for the SUMO-WordNet mappings. Sigma finds WordNet synsets without mapped formal terms and those for which a formal term is provided, but is not found in the current loaded knowledge base. This helps to find cases where terms have been changed or renamed and the mappings not updated. Most significant is the taxonomy comparison component. Given that we have terms A and B in SUMO and synsets X and Y in WordNet, if A is mapped to X and B to Y, Sigma checks whether if B is a subclass of A then Y is also a hyponym of X. The reverse case is also checked. It is not always the case that a hierarchy mismatch is an error. SUMO has a much richer set of relations than WordNet, as is appropriate for a formal ontology. A linguistic product must focus on linguistic relations that are directly evident in language. For example, WordNet considers a "plumber" to be a "human", whereas SUMO considers plumber to be an occupational position, and therefore an attribute that holds true about a particular human at a particular time.

5 INFERENCE

Since 2003, Sigma has used an open-source, customized version of the Vampire theorem prover called KIF-Vampire. Because SUMO has contained a limited number of higher-order constructs, and Vampire is strictly a first order prover, we have employed a number of pre-processing steps to translate SUMO into the more limited strict first order interpretation that Vampire (and other provers) can handle. These steps include (1) creating two approaches for removing variables from the predicate position. Our first approach was to add a "dummy" predicate to all clauses. This however resulted in worse performance for provers that give special indexing priority to the predicate when searching the proof space. The second approach was to instantiate every predicate variable with all possible values for predicates in the knowledge base that meet the type restrictions that may be implied by the axiom. For example,
in the following axiom, the axiom will be duplicated with the variable ?REL being instantiated with every TransitiveRelation.

```
(<=>
    (instance ?REL TransitiveRelation)
    (forall (?INST1 ?INST2 ?INST3)
        (=>
            (and
                (?REL ?INST1 ?INST2)
                (?REL ?INST2 ?INST3))
            (?REL ?INST1 ?INST3))))
```

This results in an automated expansion of the number of axioms, but does give good performance. One limitation however is that the semantics of predicate variables is thereby limited to the set of predicates existing in the knowledge base, rather than ranging over all possible predicates.

In preprocessing step (2) we turn embedded higher order formulas into uninterpreted lists of symbols. This removes most of the semantics of such statements, including the semantics of logical operators, but does at least allow for unification, thereby giving the appearance of higher order reasoning in very limited situations. In step (3) Sigma translates SUMO basic arithmetic functions into the native symbols required by KIF-Vampire. For step (4) we note that SUMO includes row variables [11], which are akin to the LISP @REST reference for variable-arity functions. We treat these as a "macro" and expand each axiom with a row variable into several axioms with one to seven variables for each occurrence of a row variable. In the few cases where axioms have two row variables, this can result in 48 new axioms. This limits the semantics of the row variables, but only to the cases that are found in practice.

Since we wish to keep Sigma as a completely open source system, we have not been able to upgrade to subsequent versions of Vampire, which are not open source, resulting in an inference component that is now somewhat out of date with respect to the state of the art. We have worked to integrate the TPTPWorld suite that has many different theorem provers, all operating under a common interface[12]. The different provers do however have different performance characteristics, and some do not provide proofs, so using this component does require a bit more expertise along with more choice. It also offers the capability to use the servers at U. Miami to run the user's inferences, which can be beneficial for those who may not have powerful computers at their location.

Integration with TPTP added a new first order language capability to Sigma for ontology reading and for export[13]. It also highlighted a limitation of Sigma until that point. Although SUMO has types defined for all relations, the logic itself is not typed. That meant that provers would not necessarily take advantage of type restrictions in limiting their search space, and, in certain cases, could result in incorrect inferences, when inappropriate types were applied in finding solutions to queries. A theorem prover was free to use inappropriate types and then find a contradiction with SUMO's type restrictions, resulting in an inconsistent knowledge base. To solve this problem, we added a pre-processor which adds type restrictions as a new precondition to every rule. These type restrictions are deduced by collecting the most specific type restriction implied by the use of each variable as the argument to a relation in the given axiom.

Combining the automatic generation of type restrictions on axioms with the capability to generate TPTP language versions of SUMO allowed us to use SUMO-based tests in the yearly CASC competition[14,15], stretching theorem prover developers to work on high performance results in a large new category of problems in which inferences of modest difficulty must be done on a very large knowledge base, where only a small number of axioms are relevant to a given query. A key recent innovation is the SUMO Inference Engine (SInE) [16]. which selects only the subset of axioms likely to be relevant for a given query.

Another recent innovation is in translating SUMO to a typed higher order form[18] for use by true higher order theorem provers [17].The goal of this work is to better support higher order aspects in SUMO, in particular, embedded formulas and modal operators.

At the boundary of diagnostics and inference we have the general case of using theorem proving to find contradictions. Because first order proving is not guaranteed to find all problems that may exist, Sigma includes a consistency check function that leads the theorem prover to consider each axiom in a knowledge base. Each axiom is loaded one by one starting with an empty knowledge base. For each axiom, the prover is asked to computer whether the knowledge base contradicts the axiom, or is redundant with it. If the axiom doesn't create a contradiction, it is asserted to the knowledge base and the next axiom is considered. A contradiction will stop processing, since once a contradiction is found, any further results may be nonsensical. Redundancies are collected and reported once processing finishes.

Similar to the CASC competition, but on a much smaller scale, Sigma has the capability to run a series of SUMO-based tests for any theorem prover it supports, reporting success or failure and the time taken on each test.

6 MAPPING, MERGING and TRANSLATION

In addition to SUO-KIF and TPTP Sigma can also read and write OWL format [21]. Since many lightweight ontologies are currently being created in OWL, this feature opens up the use of Sigma to a large community, and provides a straightforward migration path to use of a stronger logic and more sophisticated inference. It also opens up the use of SUMO to a community that wishes to have simple and fast inference, since SUMO can be (and is) exported with a lossy translation to an OWL version. While the bulk of the SUMO axioms are not directly expressible in OWL, they can serve as informative comments (and in fact are exported as human-readable comments) that serve to better define terms for the human user than if they were simply omitted.

We should note that a general philosophy during the construction of SUMO was not to limit it to the theorem provers or techniques available at the time of knowledge engineering. If something needed to be stated to capture the semantics of a concept, we used a logic expressive enough to state it. The idea was that any statement too complicated for reasoning could at least be used as a formal concept. It's always possible to leave out complex statements in order to comply with the need for faster or decidable inference. It is not possible, obviously, to automatically create knowledge base content that does not exist, once better inference capabilities become available. This approach is paying off now that serious work is underway on practical higher order reasoning.

Sigma also includes an export of facts in Prolog form. Once Sigma generates a TPTP version of an ontology, the TPTPWorld tools also handle a translation to Prolog that supports horn clause rules. There is also a simple prototype capability for exporting SQL statements for database creation and population from Sigma.

The growing availability and coverage of lightweight taxonomies that cover domain specific knowledge, and the corresponding phenomenon of "linked data" as a community objective has encouraged the addition of an ontology mapping and merging capability to Sigma. It is based on earlier work on a stand-alone tool [23]. In mapping SUMO to simple taxonomies there is often very little information for the machine to use to determine what matches might exist. The principal problem appears to be massive numbers of false positive matches. A simple algorithm appears to do as well in practice as a more sophisticated one, since the bulk of effort is still spent by a human in selecting accurate matches. Having a simple and easy user interface appears to provide more leverage than an incrementally better matching algorithm. The Sigma matching tool has been used to create an initial alignment with the lightweight Open Biomedical Ontologies (OBO) [24] that consist mostly of taxonomic relations, with no rules and few axioms besides class membership.

7 SUMMARY and CONCLUSIONS

Sigma has served two main purposes. It is a practical tool that has supported the development of the SUMO. It is also a toolkit and testbed that is used to support experiments in ontology application and logical reasoning. Sigma has co-evolved with SUMO with each becoming more sophisticated and extensive as they progressed. The regular open source release of both products has and will continue to form a unique resource for academic and commercial researchers and practitioners engaged in ontology, natural language understanding and formal reasoning.

REFERENCES

[1] Sigma web site http://sigmakee.sourceforge.net

[2] Pease, A., (2003). The Sigma Ontology Development Environment, in Working Notes of the IJCAI-2003 Workshop on Ontology and Distributed Systems. Volume 71 of CEUR Workshop Proceeding series.

[3] Niles, I., & Pease, A., (2001), Toward a Standard Upper Ontology, in Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001), Chris Welty and Barry Smith, eds., pp2-9.

[4] de Melo, G., Suchanek, F., and Pease, A., (2008). Integrating YAGO into the Suggested Upper Merged Ontology. In Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008). IEEE Computer Society, Los Alamitos, CA, USA.

[5] Niles, I., and Pease, A., (2003). Linking Lexicons and Ontologies: Mapping WordNet to the Suggested Upper Merged Ontology, Proceedings of the IEEE International Conference on Information and Knowledge Engineering, pp 412-416.

[6] Pease, A., and Fellbaum, C., (2010) Formal Ontology as Interlingua: The SUMO and WordNet Linking Project and GlobalWordNet, In: Huang, C. R. et al (eds.) Ontologies and Lexical Resources. Cambridge: Cambridge University Press, ISBN-13: 9780521886598.

[7] Kucera and Francis, W.N. (1967). Computational Analysis of Present-Day American English. Providence: Brown University Press.

[8] Landes S., Leacock C., and Tengi, R.I. (1998) “Building semantic concordances”. In Fellbaum, C. (ed.) (1998) WordNet: An Electronic Lexical Database. Cambridge (Mass.): The MIT Press.

[9] Global WordNet web site http://www.globalwordnet.org

[10] SUMO web site http://www.ontologyportal.org

[11] Hayes, P., and Menzel, C., (2001). A Semantics for Knowledge Interchange Format, in Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology.

[12] Trac, S., Sutcliffe, G., and Pease, A., (2008) Integration of the TPTPWorld into SigmaKEE. Proceedings of IJCAR '08 Workshop on Practical Aspects of Automated Reasoning (PAAR-2008). Volume 373 of the CEUR Workshop Proceedings.

[13] Pease, A., and Sutcliffe, G., (2007) First Order Reasoning on a Large Ontology, in Proceedings of the CADE-21 workshop on Empirically Successful Automated Reasoning on Large Theories (ESARLT).

[14] Sutcliffe. G., (2007) TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, Volume 4649/2007, ISBN 978-3-540-74509-9, pp 6-22.

[15] Pease, A., Sutcliffe, G., Siegel, N., and Trac, S., (2010). Large Theory Reasoning with SUMO at CASC, AI Communications, Volume 23, Number 2-3 / 2010, Special issue on Practical Aspects of Automated Reasoning, IOS Press, ISSN 0921-7126, pp 137-144.

[16] Hoder, K. (2008) Automated Reasoning in Large Knowledge Bases, PhD thesis, Charles University, Prague, Czech Republic. See http://is.cuni.cz/eng/studium/dipl_st/index.php? doo=detail&did=49052

[17] Benzmüller, C., and Pease., A., (2010). Progress in Automating Higher Order Ontology Reasoning, Proceedings of the Second International Workshop on Practical Aspects of Automated Reasoning, Boris Konev and Renate A. Schmidt and Stephan Schulz, editors, Edinburgh, UK, July 14, 2010, CEUR Workshop Proceedings.

[18] Sutcliffe, G., and Benzmüller, C., (2010) Automated Reasoning in Higher-Order Logic using the {TPTP THF} Infrastructure, Journal of Formalized Reasoning, vol 3, no.1, pp1-27.

[19] Pease, A., (2009). Standard Upper Ontology Knowledge Interchange Format, dated 6/18/2009. Available at http://sigmakee.cvs.sourceforge.net/*checkout*/sigmakee/sigma/suo-kif.pdf

[20] Genesereth, M., (1991). “Knowledge Interchange Format’’, In Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning, Allen, J., Fikes, R., Sandewall, E. (eds), Morgan Kaufman Publishers, pp 238-249.

[21] Sean Bechhofer, Frank van Harmelen, James A. Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein (Auth.), Mike Dean, Guus Schreiber (Ed.), OWL Web Ontology Language Reference, World Wide Web Consortium, Recommendation REC-owl-ref-20040210, February 2004.

[22] Pease, A., and Benzmüller, C., (2010) Ontology Archaeology: A Decade of Effort on the Suggested Upper Merged Ontology, in Proceeding of The ECAI-10 Workshop on Automated Reasoning about Context and Ontology Evolution (ARCOE-10), A.Bundy and J.Lehmann and G.Qi and I.J.Varzinczak editors, August 16-17, Lisbon, Portugal.

[23] Li, J., (2004) LOM: A Lexicon-based Ontology Mapping Tool, in Proceedings of the Performance Metrics for Intelligent Systems conference (PerMIS).

[24] Smith B, Ashburner M, Rosse C, Bard C, Bug W, Ceusters W, Goldberg L J, Eilbeck K, Ireland A, Mungall C J, The OBI Consortium, Leontis N, Rocca-Serra P, Ruttenberg A, Sansone S-A, Scheuermann R H, Shah N, Whetzel P L and Lewis S (2007). "The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration", Nature Biotechnology 25, 1251 - 1255.

[25] Youn, S., and McLeod, D. Ontology Development Tools for Ontology-Based Knowledge Management. Encyclopedia of E-Commerce, E-Government and Mobile Commerce, Idea Group Inc., 2006.

[26] David Aspinall. Proof General: A Generic Tool for Proof Development. Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000, LNCS 1785.

Characterization of configuration knowledge bases

Juha Tiihonen
Department of Computer Science and Engineering
Aalto University
Juha.Tiihonen@tkk.fi

Abstract

Characterization of configuration problems has remained limited. This work characterizes 26 configuration models with numerous indicators of size and degree of application of modeling mechanisms including inheritance and application of advanced compositional structure. The original goal of modeling was to evaluate and demonstrate the applicability of WeCoTin configurator and PCML modeling language to industrial configuration problems. The main contribution of the paper is in providing probably the first multi-faceted detailed characterization of a relatively large number of configuration problems. Additionally, aspects for characterizing configuration models were identified.

1 Introduction

Due to active research in the configuration domain, several formalisms for representing configuration knowledge have been proposed, and configurators are used to support day-to-day business in many companies. However, characterization of practical configuration problems has remained limited in the literature. Of course, a number of individual cases have been documented thoroughly, most importantly the R1/XCON [Barker, et al. 1989] and the VT/Sisyphus elevator configuration problem [Schreiber, et al. 1996], not forgetting characterizing the domain and configuration models when describing configurator implementations, e.g. [Fleischanderl, et al. 1998]. Further, classification of configuration tasks has been proposed [Wielinga, et al. 1997, Haag. 2008], which requires characterization of the tasks as a basis for classifications.

There are several reasons why configuration tasks should be characterized. These include enabling evaluation of effectiveness of specific representation formalisms and modeling constructs, gaining understanding on the nature of different configuration tasks, which in turn could enable supporting tools that better match practical problems and facilitates development of benchmarks, and enabling classification of configuration tasks. Configuration models, related configuration tasks and their IT support could be characterized from several perspectives. These include the size of the models, computational performance, and complexity, effectiveness of specific modeling techniques related to specific configuration tasks as defined by the company offering. Less technical views cover aspects of practical interest such as the proportion of cases covered by the configuration models, completeness of the models in terms of business requirements, usability and different aspects of utility provided and sacrifices required.

This work reports a part of evaluation of the WeCoTin configurator [Tiihonen, et al. 2003] and especially its modeling capabilities. 26 configuration models were created to evaluate and demonstrate the applicability of WeCoTin and PCML to real industrial configuration problems, which demonstrates the high-level efficacy of the constructs. The models will be characterized with numerous indicators of size and degree of application of different modeling constructs. Five tables provide characterizations models.

This paper is structured as follows. In Section 2 the applied modeling language will be described. Section 3 gives an overview of the configuration models and their background. Section 4 describes component type hierarchy, overall configuration model size, and price modeling. Section 5 details the compositional structure of the models, Section 6 attributes, and Section 7 constraints. Finally, discussion, future work and conclusions are presented in Section 8.

2 Product Configuration Modeling Language

The configuration models have been modeled with ***PCML*** (Product Configuration Modeling Language) [Tiihonen, et al. 2003, Peltonen, et al. 2001]. PCML is object-oriented, declarative and it has formal implementation-independent semantics. The semantics of PCML is provided by mapping it to weight constraint rules [Soininen, et al. 2001]. The basic idea is to treat the sentences of the modeling language as short hand notations for a set of sentences in the weight constraint rule language (***WCRL***) (Soininen et al. 1998). PCML covers a practically important subset of a synthesized ontology of configuration knowledge [Soininen, et al. 1998].

The main concepts of PCML are ***component types***, their ***compositional structure***, ***properties*** of components, and ***constraints***. Component types define the parts and

Model	Company & Description Status and validation
1 Compressor FM	Gardner Denver. Compressor family FM series. 18-40kW compressors. Internal view for sales persons. Demonstrated integration to automatic manufacturing completion (EDMS2), and e-commerce site Intershop 4. Delivery time calculation. Complete model. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Company workers had a few dozen configuration sessions over the web. Model used in empirical performance testing. Manually analyzed the number of possible configurations, which matches the number of answer sets (configurations) generated by the inference engine.
2Compr FM sc	Gardner Denver. Model 1 Compressor FM above augmented with 13 pre-selection packages to represent agreed-on customer standards with their defaults, usually enforced with soft constraints. Behaves as intended.
3 Compr FS	Gardner Denver compressor family, 11-18 kW FS series compressors. Internal view for sales persons. Complete model. Demonstrations to company with expert and focus group validation on matching company view of product configurability. In addition, company workers had configuration sessions over the web. Model used in empirical performance testing. The manually analyzed number of possible configurations matches the number generated by the inference engine.
4 Compr FX	Gardner Denver compressor family, 4-10kW FX series compressors. Internal view for sales persons. Complete model. Demonstrations to company with expert and focus group validation on matching company view of product configurability. In addition, company workers had configuration sessions over the web. Model used in empirical performance testing. The manually analyzed number of possible configurations matches the number generated by the inference engine.
5Compr FL	Gardner Denver compressor family, 45-80 kW FL series compressors. Internal view for sales persons. Complete model, seems to work, no validation in company.
6Compr M	Gardner Denver compressor family. 75-160kW M series compressors. Internal view for sales persons. Complete model, seems to work, no validation in company.
7 KONE old	KONE maintenance service contracts, older company offering. Some additional options that can be specified after contract made (billing options, e-notification). Some one-time service extras like long-term maintenance plan, condition check. Extensive on-line help texts for salespersons developed together with company, shown with the description mechanism. Complete model with extras. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Installed to product manager computer for test use.
8 KONE new	KONE maintenance service contracts. Options of newer maintenance service contracts. Additional options that can be specified after contract made (billing options, extensive e-services). Some one-time service extras like long-term maintenance plan, condition check. Extensive on-line help texts for sales persons. Complete model with extras. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Installed to product manager computer for test use.
9 Bed	Not public 1. Real hospital bed product line based on order forms and additional information. Complete model. Demonstration to company and focus group on matching company view of product configurability.
10Fireplace	Not public 2. Modular fireplace. Technology demonstration with CAD vendor. Simple product. Integrated with 3D CAD to visualize configuration changes. Complete model. Validation with CAD vendor.
11 Patria Pasi	Patria Vehicles. Military vehicle. For company internal systematic documentation of available productized options. Complete model with respect to standardized offering. Demonstrations to company with expert and focus group validation on matching company view of product configurability. Configuration model validated in internal test use in company.
12 Dental	Not public 3. Real integrated dental unit and patient chair. Most difficult to configure product of the company. Complete model. Demonstration to company, brief focus group, expert validation.
13 X-ray	Not public 3. Real X-ray unit for dentists. Product designed with ease of configuration in mind. Complete model. Demonstration to company, brief focus group, expert validation.
14 Vehicle	Not public 4. Self-moving machine industry product. Real product based on order forms and interviews. Partial model for demonstration purposes representing about half of the sales view of the product. Numerous optional parts and some simple constraints were excluded. Despite omissions the model reflects quite well the nature of sales configuration of this vehicle. Test-used by company stakeholders over the web. Functionality found satisfying "better than our commercial product". The manually analyzed number of possible configurations matches the number generated by the inference engine.
15 Insur 1	Tapiola group. Insurance coverage for families (persons, travel, car, home, cottage) as a combination of insurance products. Demonstration model with a subset of the whole offering, Discretized large domain specification attributes. Demonstrations to company.
16 Insur 2	Tapiola group. Comprehensible insurance coverage for family's person related risks. Non-traditional risk-oriented (not insurance product oriented) way of asking which coverage is desired. These selections are satisfiable with a combination of real products. Mapping to products not performed in model. Demonstration model. Demonstration to company and focus group.
17 Insur 3	Tapiola group. Experimented 4-world model objects-world questions. Solution-world with detailed model of real offered car-related insurance coverage. HUT internal, no company validation.
18 Insur 4	Tapiola group. Comprehensive insurance coverage for families and property. Tested some 4-worlds model ideas. Objects: persons, home, leisure-time apartment, vehicles (cars, motorcycle, boat), forest, domestic animals (dogs, horses, cats). Solutions corresponding real insurance products and their availability, modeled in detail. However, risks coverage for home and cottage is selectable with excessively small granularity. HUT internal, no company validation.

19 Mob Subscr 1	Elisa (Radio-linja). Demonstration on configuring mobile subscription and its value-added services while taking into account phone capabilities. Model covers only aspects considered interesting for the demonstration. Information acquired from public company web-site. Demonstration model, demonstrated to company stakeholders.
20 Mob Subscr 2	Elisa. Real mobile subscription + phone bundle as basis. Reverse-engineered from company web site + added needs analysis questions to identify suitable product options with soft constraints HUT internal validation against offering., presentation in an open seminar for the Finnish industry.
21 Mob Subscr 3	Telia-Sonera. Demonstration on selecting mobile subscription and some value-added services based on usage characteristics. Implemented with hard and soft constraints. Based on public information from company website. No validation. Behaves as intended by the modeler.
22 Broadband	Telia-Sonera. Real broadband subscription product line. 4 worlds model demonstration. Re-engineered from company website, added customer and needs analysis questions, and aspects of delivery process that are configured based on selected options. Soft constraints warn when selections do not correspond to needs. Complete model with extras. Demonstration to the company.
23 Linux	Debian Linux Familiar distribution configuration model over 6 points of time and several software versions. Information gathered from Debian Linux version compatibility lists, configuration model generated via script by mapping package descriptions into PCML [Kojo, et al. 2003]. A newer version than that in Kojo et al. was characterized. No validation, "seems and behaves right", although with slow performance.
24 Iced	None. Demonstration: minimal fictive car model for ICED conference article describing WeCoTin configurator. No validation. Behaves as intended by the modeler.
25Wecotin car	BMW. Demonstration model based on a subset of a real car, identified configuration rules from company website No validation. Behaves as intended by the modeler.
26 CarDiss	BMW. Demonstration model based on 25 WeCoTin Car, with some extra fictive features to demonstrate higher cardinality. No validation. Behaves as intended by the modeler.

Table 1. Identification, description, status and validation of the configuration models.

properties of their ***individuals*** that can appear in a configuration. A component type defines its compositional structure through a set of ***part definitions***. A part definition specifies a ***part name***, a non-empty set of ***possible part types*** (***allowed types*** for brevity) and a ***cardinality*** indicating the possible number of parts. A component type may define properties that parametrize or otherwise characterize the type. A ***property definition*** consists of a ***property name***, a ***property value type*** and ***a necessity definition*** indicating if the property must be given a value in a complete configuration. Component types are organized in a ***class hierarchy*** where a ***subtype*** inherits the property and part definitions of its ***supertypes*** in the usual manner. When a type inherits data from a supertype, the type can use the inherited data "as such" or it can modify the inherited data by means of ***refinement***. Refinement is semantically based on the notion that the set of potential valid individuals directly of the subtype is smaller than the set of valid individuals directly of the supertype. A component type is either ***abstract*** or ***concrete***. Only an individual directly of a concrete type can be used in a configuration. Constraints associated with component types define conditions that a correct configuration must satisfy. A constraint expression is constructed from references to parts and properties of components and constants such as integers. These can be combined into complex expressions using relational operators and Boolean connectives.

3 Model background, identification, characterization, status and validation

PCML and WeCoTin have been used to model and configure the complete sales view of 14 real products or services, and partial sales view of 8 products or services. In the complete sales views all known configurable options of the products or services have been modeled. Configuration models of some products or services contained extra features that are not normally taken into account during sales configuration. Three additional demonstration models are included in the characterizations. Some configuration models were created in early phases of WeCoTin construction, some after completion of the development project.

In most cases, order forms, brochures, and other documentation were used as a basis for modeling, and company representatives were contacted for additional information before showing the results as demonstrations. In some cases it was possible to re-engineer configuration model information from company websites.

The WeCoTin modeling tool was instrumented to provide the characterizing metrics based on static configuration model analysis presented in Tables 2-5. The configuration models come from the following domains:

- Eight models are from machine industry and come from three companies (5 compressors (1 twice), an undisclosed vehicle, and one military vehicle).
- Three models from two companies are from healthcare domain (2 dentist equipment product families, a hospital bed family.)
- Four models from two companies are from telecommunications domain (3 mobile and 1 broadband subscriptions).
- Three models from one company are from insurance domain.
- Two models from one company represent two generations of maintenance contracts of elevators.

- One model is software configuration (Debian Linux Familiar with package versions).
- One model demonstrates configuration of a modular fireplace.
- Three models are pure demonstration models – two are based on a subset of a real car, and one is fictional.

Table 1 identifies the models and briefly characterizes the domain of each configuration model. Each configuration model is identified with a unique numeric identifier that remains the same in each table. Model names have been abbreviated in later tables due to space constraints.

Table 2 details the degree of configuration model completeness and model validation status. Five models from three companies have been test-used by company representatives. In addition, configuration demonstrations and immediately following focus groups have been used for validation of additional seven configuration models.

4 Taxonomy, model size, and pricing

Table 2 provides characterization of component type hierarchy used in the models, overview of model size, and .information on price modeling.

One model (23, Linux) was significantly larger than others and semi-automatically generated. Discussions on model characterizations exclude this model, but averages and totals are calculated with and without it.

Numbers of abstract, concrete, and total component types contribute to the size of a configuration model, and are shown in corresponding columns of Table 2. The total number of component types varied from one 1 to 626, the median was 9 and average was 18 (42 with Linux). The number of direct subtypes of abstract types (other than root of component type hierarchy `Component`) ("Subtypes") characterizes the number of component types organized in a type hierarchy. Interpreted as a percentage "% as subtypes", the figure varied from 0% to 100%, with average without Linux being 59% and median 46%.

Each selectable attribute or part of a component individual being configured generates a ***question*** during a configuration process. The number of questions in a configuration model ("Questions") roughly characterizes the size of each configuration model and the related configuration task. In a typical configuration model without redundant concrete component types, each question might have to be answered while configuring a product. All possible questions may not be asked in a configuration session, because an individual of a specific type is not necessarily selected into a configuration, or if some attributes or parts are defined to be invisible to the user or to have a fixed value. On the other hand, if several individuals of a component type are in a configuration, the number of questions may be multiplied. The average was 61 questions per configuration model, and roughly 5.4 questions per concrete type (excluding Linux).

Model	Total types	Abstract types	Concrete types	Subtypes	% as subtypes	Questions	% questions in root	Constraints	Price
1 C FM	9	2	7	4	44	31	58	17	adv
2 CFm sc	9	2	7	4	44	31	58	17	adv
3 C FS	3	0	3	0	0	24	88	14	adv
4 C FX	1	0	1	0	0	20	100	23	adv
5 C FL	9	2	7	4	44	28	64	13	no
6 C M	3	0	3	0	0	23	91	14	no
7 KO old	5	0	5	0	0	28	79	13	no
8 Ko new	15	3	12	7	47	77	4	1	no
9 Bed	31	8	23	27	87	34	76	10	basic
10 Firepl	7	1	6	4	57	4	75	0	no
11 Pasi	5	1	4	2	40	79	95	13	no
12 Dental	64	11	53	43	67	109	3	36	no
13 X-ray	11	2	9	4	36	37	41	3	no
14 Vehicl	28	4	24	9	32	24	75	7	basic
15 Ins 1	8	2	6	5	63	30	20	4	no
16 Ins 2	62	13	49	56	90	49	20	0	no
17 Ins 3	11	3	8	5	45	41	29	14	no
18 Ins 4	37	11	26	34	92	242	5	84	no
19 Mob 1	4	0	4	0	0	18	56	6	basic
20 Mob 2	39	9	30	38	97	65	25	28	basic
21 Mob 3	5	1	4	3	60	21	38	6	no
22 Broad	66	15	51	64	97	485	1	43	no
23 Linux	626	1	625	624	100	4369	14	2380	no
24 Iced	8	2	6	5	63	4	75	3	basic
25 Wcar	6	1	5	2	33	10	60	3	basic
26CarDis	10	2	8	5	50	12	58	3	basic
Total	1082	96	986	949		5985		2755	
Total no Linux	456	95	361	325		1526		375	
Average	42	4	38	37	48	227	50	106	
Avg no Linux	18	4	14	13	59	61	52	15	
Median	9	2	7	5	46	31	58	13	
Min	1	0	1	0	0	4	1	0	
Max	626	15	625	624	100	4369	100	2380	

Table 2. Use of pricing mechanisms, company look, and component type hierarchy in the configuration models.

Especially simpler models were often centered on the configuration type that is the root of the compositional hierarchy. The degree of such concentration is characterized by the proportion of questions defined in the configuration type. Column "% questions in root" specifies this proportion. On the average about half (50%), and median 58% of questions were in the root component type, with a large scale of variation.

The total number of constraints ("Constraints") specified in the component types of each configuration model varied largely, but usually remained in a few dozen at maximum. More details will be given in Section 2.5.

Column “Price” indicates which of two price calculation mechanisms, if any, was used. The “basic” mechanism considers additive prices: each component individual can have a base price determined by its type, and each attribute value can specify additional price. The sum of component individual and their attribute value prices is the price of the configuration. Four real products and three demonstration models applied this pricing mechanism.

A more advanced calculation mechanism (“adv”) [Nurmilaakso. 2004] performs definable calculations as function of the current configuration, configuration model, and external data. Values are provided to calculations when condition expressions examining a configuration evaluate to true. Three products were priced with this mechanism.

Prices were often omitted either due to indicated sensitivity or to constrain resource usage. The simple mechanism would have been sufficient for other products except compressors and insurance products.

5 Compositional structure

Table 3 exhibits details of applying compositional structure in the modeled cases. An indication on the number of parts in a configuration model is given by the number effective parts in concrete types of the model. The number of effective parts (“Effective parts”) is the sum of inherited and locally defined parts in concrete types. The average number of effective parts per concrete component type (“Eff. parts / concrete”) characterizes the breath of the configuration tree and average application of the compositional structure as a modeling mechanism. On the average, 10 parts were defined in each configuration model. Median was 4. The average of 0.6 effective parts in each concrete type indicates moderate usage of the compositional structure.

Inheritance in the compositional structure was used on the average less frequently than direct part definitions in concrete types: Eight models applied inheritance of parts whereas parts were introduced in 25 models. On the average, seven part definitions were defined in concrete types (“Part def in concrete”), and one part per configuration model was defined in abstract types (“Part def in abstract”). Four of the effective 10 part definitions were inherited. Three inherited part definitions were applied as such (“Non-refined inherited”), and one was refined, e.g. to restrict the set of allowed types (“Refined inherited”). Some models applied part inheritance significantly more. For example, in model 12 Dental, 26 (79%) of 33 effective part definitions were inherited (“% inherited parts”). Out of these 6 were refined. The eight models applying part inheritance had 31% (80) of their 257 effective part definitions inherited.

Many configuration models concentrated part definitions on the configuration type. An average configuration type contained 4 part definitions (“Part def in conftype”). In 12 models all parts were defined in the configuration type. The average percentage of part definitions in the configuration type was 70% (“% part def in conf type”).

The cardinality of a part definition defines how many component individuals must realize the part in a consistent and complete configuration. On the average, 6 part definitions were optional, that is, with cardinality with 0 to 1 (“0 to 1 cardinality”), and 2 part definitions were obligatory with cardinality 1 to 1 (“1 to 1 cardinality”). Only one demonstration model contained one part definition with a larger maximum cardinality than one (“max cardinality 2+”).

Model	Effective parts	Eff. parts / concrete	Part def in concrete	Part def in abstract	Part def in conf type	% part def in conftype	Non-refined inherited	Refined inherited	% inherited parts	0 to 1 cardinality	1 to 1 cardinality'	max cardinality 2+	Enumerated allowed	Effective allowed	% allowed saved	Max allowed
1 C FM	4	0.6	2	1	2	50	1	1	50	0	3	0	6	6	0	2
2 C Fm sc	4	0.6	2	1	2	50	1	1	50	0	3	0	6	6	0	2
3 Com FS	1	0.3	1	0	1	100	0	0	0	0	1	0	2	2	0	2
4 com FX	0	0.0	0	0	0	-	0	0	-	0	0	0	0	0	-	0
5 com FL	4	0.6	2	1	2	50	1	1	50	0	3	0	6	6	0	2
6 com M	1	0.3	1	0	1	100	0	0	0	0	1	0	2	2	0	2
7 KO old	2	0.4	2	0	2	100	0	0	0	1	1	0	4	4	0	2
8 Ko new	19	1.6	10	3	2	11	5	4	47	11	2	0	14	17	18	3
9 Bed	3	0.1	3	0	3	100	0	0	0	0	3	0	5	32	84	12
10 Firepla	2	0.3	2	0	2	100	0	0	0	1	1	0	2	5	60	4
11 Pasi	2	0.5	2	0	2	100	0	0	0	1	1	0	3	3	0	2
12 Dental	33	0.6	7	13	1	3	20	6	79	18	2	0	33	80	59	8
13 X-ray	5	0.6	3	1	2	40	0	2	40	3	1	0	8	8	0	3
14 Vehicl	16	0.7	16	0	12	75	0	0	0	12	4	0	20	24	17	3
15 Insur 1	10	1.7	10	0	6	60	0	0	0	9	1	0	10	10	0	1
16 Insur 2	30	0.6	22	4	10	33	8	0	27	24	2	0	26	29	10	4
17 Insur 3	12	1.5	12	0	11	92	0	0	0	10	2	0	13	13	0	2
18 Insur 4	53	2.0	30	5	12	23	27	0	51	35	0	0	35	46	24	4
19 Mob 1	3	0.8	3	0	3	100	0	0	0	3	0	0	3	3	0	1
20 Mob 2	13	0.4	13	0	12	92	0	0	0	8	5	0	15	27	44	5
21 Mob 3	1	0.3	1	0	1	100	0	0	0	1	0	0	3	3	0	3
22 Broad	32	0.6	30	1	4	13	2	0	6	10	21	0	32	53	40	15
23 Linux	624	1.0	624	0	624	100	0	0	0	624	0	0	624	624	0	1
24 Iced	2	0.3	2	0	2	100	0	0	0	0	2	0	2	5	60	3
25 Car	2	0.4	2	0	2	100	0	0	0	1	1	0	3	4	25	2
26 CarDis	3	0.4	3	0	3	100	0	0	0	1	1	1	4	7	43	3
Total	881		805	30	724		65	15		773	61	1	881	1019		
Total no Linux	257		181	30	100		65	15		149	61	1	257	395		
Average	34	0.7	31	1	28	72	3	1	16	30	2	0	34	39	19	4
Avg no li	10	0.6	7	1	4	70	3	1	17	6	2	0	10	16	20	4
Median	4	0.6	3	0	2	92	0	0	0	1	1	0	6	7	0	3
Minimum	0	0.0	0	0	0	3	0	0	0	0	0	0	0	0	0	0
Maximum	624	2.0	624	13	624	100	27	6	79	624	21	1	624	624	84	15

Table 3. Compositional structure of the configuration models.

A part is realized with individuals(s) of allowed types. A part definition explicitly enumerates the allowed types. The number of directly enumerated allowed types (“Enumerated allowed”) is often smaller than the number of effectively allowed component types (“Effective allowed”), because specifying a supertype as an allowed type effectively specifies the concrete subtypes as allowed types. An average configuration model directly specified 10 and effectively 16 component types in the average 8 part definitions. The average percentage of “savings” was 20% (“% allowed saved”). The relatively low number of allowed types is partially explained by relatively often occurring optional parts with only one effective allowed type. The maximum number of effective allowed types in a part definition (“Max allowed”) was on the average and median 4 types, and maximally 15.

6 Attributes

Table 4 exhibits details of applying attributes in the modeled cases. A rough indication on the number of attributes in a configuration model is given by the number effective attributes of the configuration model. The number of effective attributes (“Effective attributes”) is the sum of inherited and locally defined attributes in concrete types. Concrete types had a total of 1269 effective attributes. Average without the Linux model was 51 effective attributes and the median was 25. The average of average number of effective attributes per concrete component type (“Effective / concrete”) was 4.8, and median of averages was 3.7.

Inheritance of attributes was applied more frequently than inheritance of parts, but less frequently than direct attribute definitions in concrete types: 16 models applied inheritance of attributes whereas all 26 models defined attributes.

On the average a model contained 26 attribute definitions in concrete types (“Defs in concrete”). The 5 attribute definitions in abstract types (“Defs in abstract”) expanded to an average of 25 effective attributes in concrete types. The average percentage of inherited attributes in concrete types (“% Inherited”) was 23%. Some models applied attribute inheritance more significantly, e.g. 44 - 89% of effective attributes were inherited in some larger models.

Of the 1269 effective attributes, 51% (642) were defined locally (“Defs in Concrete”), and the remaining 49% (627) were inherited. 122 attribute definitions in abstract types (“Defs in abstract”) were inherited as such into 537 attributes in subtypes (“Non-refined inherited”), and into 168 attributes in refined form (“Refined inherited”), a total of 705 inherited attributes.[1] Often attribute definitions were concentrated on the configuration type, 54% (14) of the 26 models specified at least 50% of effective attribute definitions there. An average configuration type defined 11 attributes (“Def in config type”). The average percentage of attribute definitions in the configuration type was 46%. (“% part def in conf type”).

Model	Effective attributes	Effective / concrete	% Inherited	Attr. definitions	Defs in concrete	Defs in abstract	Def in config type	Boolean	Enumerated string	Integer	Refined inherited	Non-refined inherited	Optional attr. defs	Maximum domain	Domain 1	Domain 2 to 3	Domain4 to 10	Domain 11+
1 FM	27	3.9	22	24	21	3	16	7	13	4	2	4	0	61	0	17	6	1
2 Fm sc	27	3.9	22	24	21	3	16	7	13	4	2	4	0	61	0	17	6	1
3 FS	23	7.7	0	23	23	0	20	5	13	5	0	0	0	51	0	16	6	1
4 FX	20	20.0	0	20	20	0	20	5	10	5	0	0	0	44	0	12	7	1
5 FL	24	3.4	17	22	20	2	16	7	12	3	2	2	0	20	0	13	7	1
6 M	22	7.3	0	22	22	0	20	5	14	3	0	0	0	15	0	11	8	1
7 K old	26	5.2	0	26	26	0	20	8	9	4	0	0	0	10	0	14	7	0
8 k new	58	4.8	81	29	11	18	1	12	7	2	23	24	0	4	0	18	3	0
9 Bed	31	1.3	0	31	31	0	23	17	13	1	0	0	2	7	0	26	5	0
10 Fire	2	0.3	0	2	2	0	1	0	2	0	0	0	0	2	0	2	0	0
11 Pasi	77	19.3	3	76	75	1	73	39	37	0	0	2	0	5	0	67	9	0
12 dent	76	1.4	70	48	23	25	2	28	19	1	3	50	8	10	0	41	7	0
13 xray	32	3.6	44	25	18	7	13	10	7	8	2	12	2	11	0	16	1	8
14Vehi	8	0.3	0	8	8	0	6	3	4	1	0	0	0	22	0	5	2	1
15 Ins1	20	3.3	10	19	18	1	0	4	6	9	0	2	0	11	1	9	8	1
16 Ins2	19	0.4	58	12	8	4	0	4	2	5	0	11	0	11	0	7	3	1
17 Ins3	29	3.6	0	29	29	0	1	17	6	4	0	0	3	122	0	21	4	2
18 Ins4	189	7.3	26	159	140	19	0	144	3	2	0	49	1	101	0	146	1	2
19 Mo1	15	3.8	0	15	15	0	7	10	2	3	0	0	2	13	0	10	2	3
20 Mo2	52	1.7	29	40	37	3	4	25	10	2	15	0	1	5	0	35	2	0
21 Mo3	20	5.0	60	12	8	4	7	12	0	0	0	12	0	2	0	12	0	0
22 Broa	453	8.9	89	81	51	30	0	51	12	3	119	361	1	436	4	58	1	3
23 Linu	3745	6.0	67	1253	1249	4	1	0	1248	3	1248	1248	0	6	551	691	9	0
24 Iced	2	0.3	0	2	2	0	1	0	1	1	0	0	0	2	0	2	0	0
25Wcar	8	1.6	25	7	6	1	4	4	3	0	0	2	0	5	0	6	2	0
26 Diss	9	1.1	22	8	7	1	4	5	3	0	2	2	0	5	0	6	2	0
Total	5014			2017	1891	126	276	429	1469	73	1416	1785	20		556	1278	108	27
Tot. no Linux	1269			764	642	122	275	429	221	70	168	537	20		5	587	99	27
Average	193	4.8	25	78	73	5	11	17	57	3	54	69	1	40	21	49	4	1
Avg no Linux	51	4.8	23	31	26	5	11	17	9	3	7	21	1	41	0	23	4	1
Median	25	3.7	19	24	21	1	5	7	8	3	0	2	0	11	0	15	4	1
Minimum	2	0.3	0	2	2	0	0	0	0	0	0	0	0	2	0	2	0	0
Maximum	3745	20.0	89	1253	1249	30	73	144	1248	9	1248	1248	8	436	551	691	9	8

Table 4. Attributes in the configuration models.

Attribute value types were distributed as follows: of average 31 attribute definitions per configuration model, 17 were Boolean (“Boolean”), 9 were enumerated strings (“”Enumerated string”), 3 integers (“Integer”), and 2 unconstrained strings. In total there were 56% (429)

[1] The 705 inherited attributes includes 78 (705-627) attributes inherited to abstract types.

Boolean, 29% (221) enumerated string, 9% (70) integer, and 6% (44) unconstrained string attribute definitions. Unconstrained strings specified additional details such as customer names, addresses, etc. aspects that do not require inference.

Attribute domain sizes remained quite small. A domain of at least 11 possible values ("Domain 11+") was present in about 4% (27) of 717 attribute definitions with a fixed domain. The maximum domain size ("Maximum domain") varied significantly – the largest domain was 436 possible values, in this case enumerated string values. The most common domain size was 2 to 3 possible values ("Domain 2 to 3") in 82% (587) of attribute definitions. 14% (108) of attribute domains were of size 4-10 ("Domain 4 to 10"). Domain of size 1 ("Domain 1") was encountered in 1% (5) cases.

20 attributes were defined as ***optional***, ("Optional attr. defs") meaning that it is possible to specify in a complete configuration that no value will be assigned to the attribute.

7 Constraints

The number of constraints varied significantly from 0 to 84 (2380 with Linux), an average of 15 per model. The median was 13. On the average, two of the constraints were soft, and the rest were hard. Totally 44 constraints were defined in abstract component types, of these 40 were hard and 4 soft. As with other modeling constructs, definition of a constraint in a supertype causes inheritance to subtypes.

It is not trivial to characterize complexity of constraints. A simple syntactic metric based on parse tree complexity of the resulting constraint expression was calculated. For example, consider the following a constraint:

```
Active_Cruise_Control_Requires_BiXenon
( Cruise_control = true ) implies
($config.Headlights individual of BiXenon)
```

The "complexity" of the example constraint is seven (7). Complexity of a literal, a constant, a variable, a component type reference, element access, an ID-expression, or an element reference in the expression is one. Each operator application counts one plus complexity of each argument.

Typical constraints were small, almost half (45%) of the constraints were of roughly the same complexity as the example constraint above, and 36% a bit more complex.

- 12% (45) of constraints had complexity 0-5
- 45% (170) of constraints had complexity 6-10
- 36% (135) of constraints had complexity 11-20
- 4% (14) of constraints had complexity 21-50
- 1% (4) of constraints had complexity 51-100
- 1% (4) of constraints had complexity 101-1000
- 1% (3) of constraints had complexity over 1000

Maximum constraint complexity varied significantly. The median was as low as 13, and average without Linux was 235. The maximum complexity was 1319. All the compressor models had a large table constraint specifying feasible combinations of values of 5 attributes, each with a relatively large number of rows, which explains the high average.

8 Discussion, future work, and conclusions

8.1 Limitations

Modeling and evaluation of modeling mechanisms contains author bias. All modeling was performed by researchers who were involved in development of the system.

The companies whose products were modelled were either existing or potential research partners. In other words, the sample of companies was not selected e.g. to cover most challenging cases such as telecommunications networks.

8.2 Modeling mechanisms

Some partial models were created due to resource constraints, or when the purpose of modeling was attainable with partial modeling. In other words, capabilities of PCML or WeCoTin did not limit the scope of modeling. However, Floats, fixed point numbers or integers with very large domain would have been useful in the insurance and compressor domains. In the insurance case some specification variables such as a desired amount of monetary coverage were specifiable with arbitrary monetary amounts, which can lead to very large domains. Apartment size and desired coverage were discretized in model "16 Insur 1". In compressor models, the company had calculated and validated combinations of specification variable values that produce a specific nominal capacity, represented in a table constraint. Further, a specific percentage of capacity loss is encountered in high altitude use environments. Calculating this would have been more convenient with floating or fixed point arithmetic.

Application of the compositional structure was important but less frequent than anticipated. A partial explanation is that it was often considered more practical to model alternative or optional components as enumeration or Boolean attributes rather than as a part, if there was no need to configure details of the selected component individuals.

Part definitions with cardinality were useful: the mechanism provides a convenient way to model selecting at most one or exactly one component individual to a role in product structure out of several alternatives. This capability prevents the need for a number of extra constraints. For example, some commercial systems require that each alternative is specified as optional, and a mutual exclusivity constraint is required [Damiani, et al. 2001]. However, sometimes the mechanism was a bit clumsy: in case of an optional part (cardinality 0 to 1), and exactly one allowed type, it was difficult to invent a name for the component type and for the part. A bit surprisingly, large cardinalities were not needed in these configuration models.

Applying inheritance saved modeling effort in larger models significantly. Almost half (49%) of effective attributes were inherited, and one definition in a supertype created in average 4.4 effective attributes. Refinement of inherited attributes and parts was a useful mechanism for limiting the domain of allowed values or allowed types. created through statistics. Refinement facilitated the application of inheritance also in cases where some subtypes have a narrower range of allowed values or types. Inheritance related to compositional structure was also useful, but was applied only in about 31% of the models. This mechanism was generally used in larger models, where almost a third of part definitions were inherited.

There was no need for explicit resource balancing or satisfaction in the modeled cases. There was no need for topological modeling, e.g. ports in our modeled cases. However, when modeling services and their delivery processes [Tiihonen, et al. 2007], there was a need to assign different stakeholders as resources that participate in different service activities. This assignment can be somewhat clumsily modeled with attributes. However, allocation of responsibilities to different, dynamically defined stakeholders could be more naturally modeled as connections between the activities and stakeholders.

8.3 Future work

The amount of work required to create a configuration model depended to a large extent on the knowledge acquisition and validation work. Collecting reliable statistics on total effort of creating and maintaining configuration models remains future work.

Performance evaluations are important characterization of configuration models. In previous work, performance of some of the models has been evaluated, and was found satisfactory[Tiihonen, et al. 2002]. However, performance should be tested with a larger and more representative set of configuration models.

8.4 Conclusions

The main contribution of this paper is providing, to our knowledge, the first multi-case in-depth characterization of configuration models and analysis of utility of modeling mechanisms. A combination of taxonomic hierarchy with inheritance and strict refinement, compositional structure with the concept of part definitions, attributes, and constraints for expressing consistency requirements of a configuration seem to be able to effortlessly capture a significant subset of sales configuration problems. The utility of inheritance in configuration was shown through significant application of the mechanism, especially when related to attributes, and to a lesser but still significant extent to parts.

In addition, this work provides an initial proposal for a framework for characterizing configuration models.

Acknowledgements

We thank A. Anderson, A. Martio, J. Elfström, K. Sartinko, M. Heiskala, M. Pasanen, and T. Kojo for modeling and knowledge acquisition; A. Martio and R. Sulonen for acquisition of the cases; Gardner Denver Finland, KONE, Patria,Tapiola Group for sharing product information; and TEKES for funding WeCoTin, ConSerWe, and Cosmos

References

Barker, V. E., O'Connor, D. E., Bachant, J., & Soloway, E. (1989). Expert systems for configuration at digital: XCON and beyond. Communications of the ACM, 32(3), 298-318.

Damiani, S. R., Brand, T., Sawtelle, M., & Shanzer, H. (2001). Oracle configurator developer User’s guide, release 11i Oracle Corporation.

Fleischanderl, G., Friedrich, G., Haselbock, A., Schreiner, H., & Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. Intelligent Systems and their Applications, IEEE [See also IEEE Intelligent Systems], 13(4), 59-68.

Haag, A. (2008). What makes product configuration viable in a business? Proceedings of ECAI 2008 Workshop on Configuration Systems, Patras, Greece. 53-54.

Kojo, T., Männisto, T. And Soininen, T. (2003). Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. In Software Configuration Management (ICSE Workshops SCM 2001 and SCM 2003 Selected Papers)**,** 86-101.

Nurmilaakso, J. (2004). WeCotin.calc documentation. Unpublished manuscript.

Peltonen, H., Tiihonen, J., & Anderson, A. (2001). Configurator tool concepts and model definition language. Unpublished manuscript.

Schreiber, A. T., & Birmingham, W. P. (1996). Editorial: The Sisyphus-VT initiative. International Journal of Human-Computer Studies, 44(3-4), 275-280.

Soininen, T., Niemelä, I., Tiihonen, J., & Sulonen, R. (2001). Representing configuration knowledge with weight constraint rules. Proceedings of the AAAI Spring Symp.on Answer Set Programming: Towards Efficient and Scalable Knowledge, , 195–201.

Soininen, T., Tiihonen, J., Mannisto, T., & Sulonen, R. (1998). Towards a general ontology of configuration. AI EDAM, 12(04), 357-372.

Tiihonen, J., Heiskala, M., Paloheimo, K., & Anderson, A. (2007). Applying the configuration paradigm to mass-customize contract based services. Paper presented at the Extreme Customization: Proceedings of the MCPC 2007 World Conference on Mass Customization & Personalization, Massachusetts Institute of Technology, MA, USA. paper ID MCPC-134-2007, section 7.5.3.

Tiihonen, J., Soininen, T., Niemelä, I., & Sulonen, R. (2003). A practical tool for mass-customising configurable products. Paper presented at the Proceedings of the 14th International Conference on Engineering Design, Stockholm, Sweden. Paper 1290.

Tiihonen, J., Soininen, T., Niemelä, I., & Sulonen, R. (2002). Empirical testing of a weight constraint rule based configurator. Proceedings of the Configuration Workshop, 15th European Conference on Artificial Intelligence, Lyon, France, 2002. 17–22-17-22.

Wielinga, B., & Schreiber, G. (1997). Configuration-design problem solving. Expert, IEEE [See also IEEE Intelligent Systems and their Applications], 12(2), 49-56.

Efficient Explanations for Inconsistent Constraint Sets

Alexander Felfernig[1] and **Monika Schubert**[1] and
Monika Mandl[1] and **Gerhard Friedrich**[2] and **Erich Teppan**[2]

Abstract. Constraint sets can become inconsistent in different contexts. For example, during a configuration session the set of customer requirements can become inconsistent with the configuration knowledge base. Another example is the engineering phase of a configuration knowledge base where the underlying constraints can become inconsistent with a set of test cases. In such situations we are in the need of techniques that support the identification of minimal sets of constraints that have to be adapted or deleted in order to restore consistency. In this paper we introduce a divide-and-conquer based diagnosis algorithm (FASTDIAG) which identifies minimal sets of faulty constraints in an over-constrained problem. This algorithm is specifically applicable in scenarios where the efficient identification of leading (preferred) diagnoses is crucial. We compare the performance of FASTDIAG with the conflict-directed calculation of hitting sets and present an in-depth performance analysis that shows the advantages of our approach.

1 Introduction

Constraint technologies [19] are applied in different areas such as configuration [11, 15, 18], recommendation [9], and scheduling [3]. There are many scenarios where the underlying constraint sets can become over-constrained. For example, when implementing a configuration knowledge base, constraints can become inconsistent with a set of test cases [8]. Alternatively, when interacting with a configurator application [9, 16], the given set of customer requirements (represented as constraints) can become inconsistent with the configuration knowledge base. In both situations there is a need of an intelligent assistance that actively supports users of a constraint-based application (end users or knowledge engineers). A wide-spread approach to support users in the identification of minimal sets of faulty constraints is to combine conflict detection (see, e.g., [13]) with a corresponding hitting set algorithm [6, 17]. In their original form these algorithms are applied for the calculation of *minimal diagnoses* which are typically determined with breadth-first search. Further diagnosis algorithms have been developed that follow a best-first search regime where the expansion of the hitting set search tree is guided by failure probabilities of components [5]. Another example for such an approach is presented in [9] where similarity metrics are used to guide the (best-first) search for a preferred (plausible) minimal diagnosis (including repairs).

Both, simple breadth-first search and best-first search diagnosis approaches are predominantly relying on the calculation of conflict sets [13]. In this context, the determination of a minimal diagnosis of cardinality *n* requires the identification of at least *n* minimal conflict sets. In this paper we introduce a diagnosis algorithm (FASTDIAG) that allows to determine *one minimal diagnosis at a time* with the same computational effort related to the calculation of *one conflict set at a time*. The algorithm supports the identification of preferred diagnoses given predefined preferences regarding a set of decision alternatives. FASTDIAG is boosting the applicability of diagnosis methods in scenarios such as online configuration & reconfiguration [8], recommendation of products & services [9], and (more generally) in scenarios where the efficient calculation of preferred (leading) diagnoses is crucial [5]. FASTDIAG is not restricted to constraint-based systems but it is also applicable, for example, in the context of SAT solving [14] and description logics reasoning [12].

The remainder of this paper is organized as follows. In Section 2 we introduce a simple example configuration task from the automotive domain. In Section 3 we discuss the basic hitting set based approach to the calculation of diagnoses. In Section 4 we introduce an algorithm (FASTDIAG) for calculating preferred diagnoses for a given over-constrained problem. In Section 5 we present a detailed evaluation of FASTDIAG which clearly outperforms standard hitting set based algorithms in the calculation of the *topmost-n* preferred diagnoses. With Section 6 we provide an overview of related work in the field. The paper is concluded with Section 7.

2 Example Domain: Car Configuration

Car configuration will serve as a working example throughout this paper. Since we exploit configuration problems for the discussion of our diagnosis algorithm, we first introduce a formal definition of a configuration task. This definition is based on [8] but is given in the context of a constraint satisfaction problem (CSP) [19].

Definition 1 (Configuration Task). A configuration task can be defined as a CSP (V, D, C). V = $\{v_1, v_2, \ldots, v_n\}$ represents a set of finite domain variables. D = $\{dom(v_1), dom(v_2), \ldots, dom(v_n)\}$ represents a set of variable domains dom(v_k) where dom(v_k) represents the domain of variable v_k. C = $C_{KB} \cup C_R$ where $C_{KB} = \{c_1, c_2, \ldots, c_q\}$ is a set of domain specific constraints (the configuration knowledge base) that restrict the possible combinations of values assigned to the variables in V. $C_R = \{c_{q+1}, c_{q+2}, \ldots, c_t\}$ is a set of customer requirements also represented as constraints.

A simplified example of a configuration task in the automotive domain is the following. In this example, *type* represents the car type, *pdc* is the parc distance control functionality, *fuel* represents the fuel consumption per 100 kilometers, a *skibag* allows the ski stowage inside the car, and *4-wheel* represents the corresponding actuation type. These variables describe the potential set of requirements that can be specified by the user (customer). The possible combinations of these requirements are defined by a set of constraints which are denoted as *configuration knowledge base*, $C_{KB} = \{c_1, c_2, c_3, c_4\}$. Furthermore, we assume the set of *customer requirements* $C_R = \{c_5, c_6, c_7\}$.

[1] TU Graz, Austria, email: {felfernig, schubert, mandl}@ist.tugraz.at
[2] University of Klagenfurt, Austria, email: {friedrich, teppan}@uni-klu.ac.at

- V = {***type***, ***pdc***, ***fuel***, ***skibag***, ***4-wheel***}
- D = {dom(***type***)={*city*, *limo*, *combi*, *xdrive*}, dom(***pdc***)= {*yes*, *no*}, dom(***fuel***) = {*4l*, *6l*, *10l*}, dom(***skibag***)={*yes*, *no*}, dom(***4-wheel***)={*yes*, *no*}
- C_{KB} = {c_1: ***4-wheel*** = *yes* ⇒ ***type*** = *xdrive*, c_2: ***skibag*** = *yes* ⇒ ***type*** ≠ *city*, c_3: ***fuel*** = *4l* ⇒ ***type*** = *city*, c_4: ***fuel*** = *6l* ⇒ ***type*** ≠ *xdrive*}
- C_R = {c_5: ***type*** = *combi*, c_6: ***fuel*** = *4l*, c_7: ***4-wheel*** = yes}

On the basis of this configuration task definition, we can now introduce the definition of a concrete *configuration* (solution for a configuration task).

Definition 2 (Configuration). A configuration for a given configuration task (V, D, C) is an instantiation I = {v_1=ins_1, v_2=ins_2, ..., v_n=ins_n} where $ins_k \in$ dom(v_k).

A configuration is *consistent* if the assignments in I are consistent with the $c_i \in$ C. Furthermore, a configuration is *complete* if all variables in V are instantiated. Finally, a configuration is *valid* if it is consistent and complete.

3 Diagnosing Over-Constrained Problems

For the configuration task introduced in Section 2 we are *not* able to find a solution, for example, a *combi*-type car does not support a fuel consumption of *4l per 100 kilometers*. Consequently, we want to identify minimal sets of constraints ($c_i \in C_R$) which have to be deleted (or adapted) in order to be able to identify a solution (restore the consistency). In the example of Section 2 the set of constraints C_R={c_5, c_6, c_7} is inconsistent with the constraints C_{KB}= {c_1, c_2, c_3, c_4}, i.e., no solution can be found for the underlying configuration task. A standard approach to determine a minimal set of constraints that have to be deleted from an over-constrained problem is to resolve all minimal conflicts contained in the constraint set. The determination of such constraints is based on a conflict detection algorithm (see, e.g., [13]), the derivation of the corresponding diagnoses is based on the calculation of hitting sets [17]. Since both, the notion of a (*minimal*) *conflict* and the notion of a (*minimal*) *diagnosis* will be used in the following sections, we provide the corresponding definitions here.

Definition 3 (Conflict Set). A conflict set is a set CS ⊆ C_R s.t. C_{KB} ∪ CS is inconsistent. A conflict set CS is a *minimal* if there does not exist a conflict set CS' with CS' ⊂ CS.

In our working example we can identify three minimal conflict sets which are CS_1={c_5,c_6}, CS_2={c_5,c_7}, and CS_3={c_6,c_7}.

CS_1, CS_2, CS_3 are conflict sets since $CS_1 \cup C_{KB} \vee CS_2 \cup C_{KB} \vee CS_3 \cup C_{KB}$ is inconsistent. The minimality property is fulfilled since there does not exist a conflict set CS_4 with $CS_4 \subset CS_1$ or $CS_4 \subset CS_2$ or $CS_4 \subset CS_3$. The standard approach to resolve the given conflicts is the construction of a corresponding *hitting set directed acyclic graph* (*HSDAG*) [17] where the resolution of all minimal conflict sets automatically corresponds to the identification of a minimal diagnosis. A minimal diagnosis in our application context is a minimal set of customer requirements contained in the set of car features (C_R) that has to be deleted from C_R (or adapted) in order to make the remaining constraints consistent with C_{KB}. Since we are dealing with the diagnosis of customer requirements, we introduce the definition of a *customer requirements diagnosis problem* (Definition 4). This definition is based on the definition given in [8].

Definition 4 (CR Diagnosis Problem). A customer requirements diagnosis (CR diagnosis) problem is defined as a tuple (C_{KB}, C_R) where C_R is the set of given customer requirements and C_{KB} represents the constraints part of the configuration knowledge base.

The definition of a *CR diagnosis* that corresponds to a given CR Diagnosis Problem is the following (see Definition 5).

Definition 5 (CR Diagnosis). A CR diagnosis for a CR diagnosis problem (C_{KB}, C_R) is a set Δ ⊆ C_R, s.t., C_{KB} ∪ (C_R - Δ) is consistent. Δ is *minimal* if there does not exist a diagnosis Δ' ⊂ Δ s.t. C_{KB} ∪ (C_R - Δ') is consistent.

The HSDAG algorithm for determining minimal diagnoses is discussed in detail in [17]. The concept of this algorithm will be explained on the basis of our working example. It relies on a conflict detection algorithm that is responsible for detecting minimal conflicts in a given set of constraints (in our case in the given customer requirements). One conflict detection algorithm is QUICKXPLAIN [13] which is based on an efficient divide-and-conquer search strategy. For the purposes of our working example let us assume that the first minimal conflict set determined by QUICKXPLAIN is the set CS_1= {c_5, c_6}. Due to the minimality property, we are able to resolve each conflict by simply deleting one element from the set, for example, in the case of CS_1 we have to either delete c_5 or c_6. Each variant to resolve a conflict set is represented by a specific path in the corresponding HSDAG – the HSDAG for our working example is depicted in Figure 1. The deletion of c_5 from CS_1 triggers the calculation of another conflict set CS_3 = {c_6, c_7} since C_R - {c_5} ∪ C_{KB} is inconsistent. If we decide to delete c_6 from CS_1, C_R - {c_6} ∪ C_{KB} remains inconsistent which means that QUICKXPLAIN returns another minimal conflict set which is CS_2 = {c_5, c_7}.

The original HSDAG algorithm [17] follows a strict breadth-first search regime. Following this strategy, the next node to be expanded in our working example is the minimal conflict set CS_3 which has been returned by QUICKXPLAIN for C_R - {c_5} ∪ C_{KB}. In this context, the first option to resolve CS_3 is to delete c_6. This option is a valid one and Δ_1= {c_5, c_6} is the resulting minimal diagnosis. The second option for resolving CS_3 is to delete the constraint c_7. In this case, we have identified the next minimal diagnosis Δ_2 = {c_5, c_7} since C_R - {c_5, c_7} ∪ C_{KB} is consistent. This way we are able to identify all minimal sets of constraints Δ_i that – if deleted from C_R – help to restore the consistency with C_{KB}. If we want to calculate the complete set of diagnoses for our working example, we still have to resolve the conflict set CS_2. The first option to resolve CS_2 is to delete c_5 – since {c_5, c_6} has already been identified as a minimal diagnosis, we can close this node in the HSDAG. The second option to resolve CS_2 is to delete c_7. In this case we have determined the third minimal diagnosis which is Δ_3 = {c_6, c_7}.

In our working example we are able to enumerate all possible diagnoses that help to restore consistency. However, the calculation of all minimal diagnoses is expensive and thus in many cases not practicable for interactive settings. Since users are often interested in a reduced subset of all the potential diagnoses, alternative algorithms are needed that are capable of identifying preferred diagnoses [5, 9, 17]. Such approaches have already been developed [5, 9], however, they are still based on the resolution of conflict sets which is computationally expensive (see Section 5). Our idea presented in this paper is a diagnosis algorithm that helps to determine preferred diagnoses without the need of calculating the corresponding conflict sets. The basic properties of FASTDIAG will be discussed in Section 4.

4 Calculating Preferred Diagnoses with FASTDIAG

Preferred Diagnoses. Users typically prefer to keep the important requirements and to change or delete (if needed) the less important ones [13]. The major goal of (model-based) diagnosis tasks is to identify the preferred (leading) diagnoses which are not necessarily

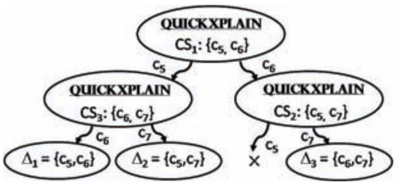


Figure 1. HSDAG (Hitting Set Directed Acyclic Graph) [17] for the CR diagnosis problem (C_R={c_5, c_6, c_7}, C_{KB}={c_1, c_2, c_3, c_4}). The sets {c_5, c_6}, {c_6, c_7}, and {c_5, c_7} are the minimal diagnoses – the conflict sets CS_1, CS_2, and CS_3 are determined on the basis of QUICKXPLAIN [13].

minimal cardinality ones [5]. For the characterization of a preferred diagnosis we will rely on the definition of a total ordering of the given set of constraints in C (respectively C_R). Such a total ordering can be achieved, for example, by *directly asking* the customer regarding the preferences, by applying *multi-attribute utility theory* [1, 20] where the determined interest dimensions correspond with the attributes of C_R or by applying the rankings determined by *conjoint analysis* [2]. The following definition of a *lexicographical ordering* (Definition 6) is based on total orderings for constraints that has been applied in [13] for the determination of *preferred conflict sets*.

Definition 6 (Total Lexicographical Ordering). Given a total order < on C, we enumerate the constraints in C in increasing < order c_1.. c_n starting with the *least important constraints* (i.e., c_i < c_j ⇒ i < j). We compare two subsets X and Y of C lexicographically:

$$X >_{lex} Y \text{ iff}$$

$$\exists k\text{: } c_k \in Y - X \text{ and}$$

$$X \cap \{c_{k+1}, ..., c_t\} = Y \cap \{c_{k+1}, ..., c_t\}.$$

Based on this definition of a lexicographical ordering, we can now introduce the definition of a *preferred diagnosis*.

Definition 7 (Preferred Diagnosis). A minimal diagnosis Δ for a given CR diagnosis problem (C_R, C_{KB}) is a preferred diagnosis for (C_R, C_{KB}) iff there does not exist another minimal diagnosis Δ′ with $\Delta' >_{lex} \Delta$.

In our working example we assumed the lexicographical ordering (c_5 < c_6 < c_7), i.e., the most important customer requirement is c_7 (the 4-wheel functionality). If we assume that $X = \{c_5, c_7\}$ and $Y = \{c_6, c_7\}$ then Y-X = $\{c_6\}$ and $X \cap \{c_7\}$ = $Y \cap \{c_7\}$. Intuitively, $\{c_5, c_7\}$ is a preferred diagnosis compared to $\{c_6, c_7\}$ since both diagnoses include c_7 but c_5 is less important than c_6. If we change the ordering to (c_7 < c_6 < c_5), FASTDIAG would then determine {c_6, c_7} as the preferred minimal diagnosis.

FASTDIAG Approach. For the following discussions we introduce the set AC = $C_{KB} \cup C_R$ which represents the union of customer requirements (C_R) and the configuration knowledge base (C_{KB}). The basic idea of the FASTDIAG algorithm (Algorithm 1) is the following.[3] In our working example, the set of customer requirements C_R = {c_5, c_6, c_7} includes at least one minimal diagnosis since C_{KB} is consistent and $C_{KB} \cup C_R$ is inconsistent. In the worst case C_R itself represents the minimal diagnosis which would mean that *all* constraints in C_R are part of the diagnosis, i.e., each $c_i \in C_R$ represents a singleton conflict. In our case C_R obviously does not represent a minimal diagnosis – the set of diagnoses in our working example is {Δ_1 = {c_5, c_6}, Δ_2 = {c_5, c_7}, Δ_3 = {c_6, c_7}} (see Section 3). The next step in Algorithm 1 is to divide the set of customer requirements C_R = {c_5, c_6, c_7} into the two sets C_1 = {c_5} and C_2 = {c_6, c_7} and to check whether AC - C_1 is already consistent. If this is the case, we can omit the set C_2 since at least one minimal diagnosis can already be identified in C_1. In our case, AC - {c_5} is inconsistent, which means that we have to consider further elements from C_2. Therefore, C_2 = {c_6, c_7} is divided into the sets {c_6} and {c_7}. In the next step we can check whether AC – (C_1 ∪ {c_6}) is consistent – this is the case which means that we do not have to further take into account {c_7} for determining the diagnosis. Since {c_5} does not include a diagnosis but {c_5} ∪ {c_6} includes a diagnosis, we can deduce that {c_6} must be part of the diagnosis. The final step is to check whether AC – {c_6} leads to a diagnosis without including {c_5}. We see that AC – {c_6} is inconsistent, i.e., Δ = {c_5, c_6} is a minimal diagnosis for the CR diagnosis problem (C_R = {c_5, c_6, c_7}, C_{KB} = {c_1, . . . , c_4}). An execution trace of the FASTDIAG algorithm in the context of our working example is shown in Figure 2.

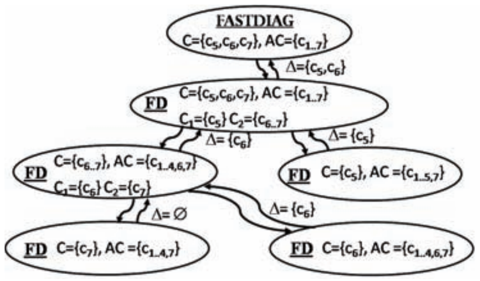

FASTDIAG
C={c$_5$,c$_6$,c$_7$}, AC={c$_{1..7}$}
Δ={c$_5$,c$_6$}
FD
C={c$_5$,c$_6$,c$_7$}, AC ={c$_{1..7}$}
C$_1$={c$_5$} C$_2$={c$_{6..7}$}
Δ= {c$_6$}
Δ= {c$_5$}
FD
C={c$_{6..7}$}, AC ={c$_{1..4,6,7}$}
C$_1$={c$_6$} C$_2$={c$_7$}
FD
C={c$_5$}, AC ={c$_{1..5,7}$}
Δ= ∅
Δ= {c$_6$}
FD
C={c$_7$}, AC ={c$_{1..4,7}$}
FD
C={c$_6$}, AC ={c$_{1..4,6,7}$}

Figure 2. FASTDIAG execution trace for the CR diagnosis problem (C_R={c_5, c_6, c_7}, C_{KB}={c_1, c_2, c_3, c_4}).

Algorithm 1 – FASTDIAG

```
func FASTDIAG($C \subseteq AC, AC = \{c_1..c_t\}$) : $diagnosis\ \Delta$
if $isEmpty(C)\ or\ inconsistent(AC - C)\ return\ \emptyset$
else $return$ FD($C, AC$);

func FD($C = \{c_1..c_q\}, AC$) : $diagnosis\ \Delta$
if $consistent(AC)\ return\ \emptyset$;
if $singleton(C)\ return\ C$;
$k = \frac{n}{2}$;
$C_1 = \{c_1..c_k\}; C_2 = \{c_{k+1}..c_q\}$;
$D_1$ = FD($C_2, AC - C_1$);
$D_2$ = FD($C_1, AC - D_1$);
$return(D_1 \cup D_2)$;
```

Calculating n>1 Diagnoses. In order to be able to calculate *n*>1 diagnoses[4] with FASTDIAG we have to adopt the HSDAG construction introduced in [17] by substituting the resolution of conflicts (see Figure 1) with the deletion of elements c_i from C_R (C) (see Figure 3). In this case, a path in the HSDAG is closed if no further diagnoses can be identified for this path or the elements of the current path are a superset of an already closed path (containment check). Conform to the HSDAG presented in [17], we expand the search tree

[3] In Algorithm 1 we use the set C instead of C_R since the application of the algorithm is not restricted to inconsistent sets of customer requirements.

[4] Typically a CR diagnosis problem has more than one related diagnosis.

in a *breadth-first* manner. In our working example, we can delete $\{c_5\}$ (one element of the first diagnosis Δ_1 = $\{c_5, c_6\}$) from the set C_R of diagnosable elements and restart the algorithm for finding another minimal diagnosis for the CR diagnosis problem ($\{c_6, c_7\}$, C_{KB}). Since AC - $\{c_5\}$ is inconsistent, we can conclude that C_R = $\{c_6, c_7\}$ includes another minimal diagnosis (Δ_2 = $\{c_6, c_7\}$) which is determined by FASTDIAG for the CR diagnosis problem (C_R - $\{c_5\}$, C_{KB}). Finally, we have to check whether the CR diagnosis problem ($\{c_5, c_7\}$, C_{KB}) leads to another minimal diagnosis. This is the case, i.e., we have identified the last minimal diagnosis which is Δ_3 = $\{c_5, c_7\}$. The calculation of all diagnoses in our working example on the basis of FASTDIAG is depicted in Figure 3.

Note that for a given set of constraints (C) FASTDIAG always calculates the preferred diagnosis in terms of Definition 7. If Δ_1 is the diagnosis returned by FASTDIAG and we delete one element from Δ_1 (e.g., c_5), then FASTDIAG returns the preferred diagnosis for the CR diagnosis problem ($\{c_5$, c_6, $c_7\}$-$\{c_5\}$, $\{c_1$, ..., $c_7\}$) which is Δ_2 in our example case, i.e., $\Delta_1 >_{lex} \Delta_2$. Consequently, diagnoses part of one path in the search tree (such as Δ_1 and Δ_2 in Figure 3) are in a strict preference ordering. However, there is only a *partial order* between individual diagnoses in the search tree in the sense that a diagnosis at level *k* is not necessarily preferable to a diagnosis at level *k*+1.

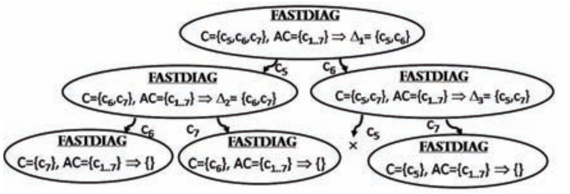


Figure 3. FASTDIAG: calculating the complete set of *minimal diagnoses*.

FASTDIAG Properties. A detailed listing of the basic operations of FASTDIAG is shown in Algorithm 1. First, the algorithm checks whether the constraints in C contain a diagnosis, i.e., whether AC - C is consistent – the function assumes that it is activated in the case that AC is inconsistent. If AC - C is inconsistent or C = ∅, FASTDIAG returns the empty set as result (no solution can be found). If at least one diagnosis is contained in the set of constraints C, FASTDIAG activates the FD function which is in charge of retrieving a preferred diagnosis. FASTDIAG follows a divide-and-conquer strategy where the recursive function FD divides the set of constraints (in our case the elements of C_R) into two different subsets (C_1 and C_2) and tries to figure out whether C_1 already contains a diagnosis. If this is the case, FASTDIAG does not further take into account the constraints in C_2. If only one element is remaining in the current set of constraints C and the current set of constraints in AC is still inconsistent, then the element in C is part of a minimal diagnosis. FASTDIAG is *complete* in the sense that if C contains exactly one minimal diagnosis then FD will find it. If there are multiple minimal diagnoses then one of them (the preferred one – see Definition 7) is returned. The recursive function FD is triggered if AC-C is consistent and C consists of at least one constraint. In such a situation a corresponding minimal diagnosis can be identified. If we assume the existence of a minimal diagnosis Δ that can not be identified by FASTDIAG, this would mean that there exists at least one constraint c_a in C which is part of the diagnosis but not returned by FD. The only way in which elements can be deleted from C (i.e., not included in a diagnosis) is by the return ∅ statement in FD and ∅ is only returned in the case that AC is consistent which means that the elements of C_2 (C_1) from the previous FD incarnation are not part of the preferred diagnosis. Consequently, it is not possible to delete elements from C which are part of the diagnosis. FASTDIAG computes only *minimal diagnoses* in the sense of Definition 5. If we assume the existence of a non-minimal diagnosis Δ calculated by FASTDIAG, this would mean that there exists at least one constraint c_a with Δ - $\{c_a\}$ is still a diagnosis. The only situation in which elements of C are added to a diagnosis Δ is if C itself contains exactly one element. If C contains only one element (let us assume c_a) and AC is inconsistent (in the function FD) then c_a is the only element that can be deleted from AC, i.e., c_a must be part of the diagnosis.

5 Evaluation

Performance of FASTDIAG. In this section we will compare the performance of FASTDIAG with the performance of the hitting set algorithm [17] in combination with the QUICKXPLAIN conflict detection algorithm introduced in [13].

The worst case complexity of FASTDIAG in terms of the number of consistency checks needed for calculating one minimal diagnosis is $2d \cdot \log_2(\frac{n}{d})+2d$, where *d* is the minimal diagnoses set size and *n* is the number of constraints (in C). The best case complexity is $\log_2(\frac{n}{d})+2d$. In the worst case each element of the diagnosis is contained in a different path of the search tree: $\log_2(\frac{n}{d})$ is the depth of the path, $2d$ represents the branching factor and the number of leaf-node consistency checks. In the best case all elements of the diagnosis are contained in one path of the search tree.

The worst case complexity of QUICKXPLAIN in terms of consistency checks needed for calculating one minimal conflict set is $2k \cdot \log_2(\frac{n}{k})+2k$ where *k* is the minimal conflicts set size and *n* is again the number of constraints (in C) [13]. The best case complexity of QUICKXPLAIN in terms of the number of consistency checks needed is $\log_2(\frac{n}{k})+2k$ [13]. Consequently, the number of consistency checks per conflict set (QUICKXPLAIN) and the number of consistency checks per diagnosis (FASTDIAG) fall into a logarithmic complexity class.

Let n_{cs} be the number of minimal conflict sets in a constraint set and n_{diag} be the number of minimal diagnoses, then we need n_{diag} FD calls (see Algorithm 1) plus n_{cs} additional consistency checks and n_{cs} activations of QUICKXPLAIN with n_{diag} additional consistency checks for determining *all* diagnoses. The results of a performance evaluation of FASTDIAG are depicted in the Figures 4–7. The basis for these evaluations were generated constraint sets (t = 100 constraints with a randomized lexicographical ordering and n = 100 variables) with a varying number of conflict sets (of cardinality 1–4) and corresponding diagnoses (#diagnoses between 3 – 22). The constraint solver used for consistency checking was CHOCO (*choco.emn.fr)* and the tests have been executed on a standard desktop computer (*Intel(R) Core(TM)2 Quad CPU QD9400* CPU with *2.66Ghz* and *2GB* RAM).

Figure 4 shows a comparison between the hitting set based diagnosis approach (denoted as HSDAG) and the FASTDIAG algorithm (denoted as FASTDIAG) in the case that only *one diagnosis* is calculated. FASTDIAG clearly outperforms the HSDAG approach independent of the way in which diagnoses are calculated (breadth-first or best-first). Figure 5 shows the performance evaluation for calculating the *topmost-5 minimal diagnoses*. The result is similar to the one for calculating the first diagnosis, i.e., FASTDIAG outperforms the two HSDAG versions. Our evaluations show that FASTDIAG is very efficient in calculating preferred minimal diagnoses. In contrast to the HSDAG-based best-first search mode FASTDIAG has a performance

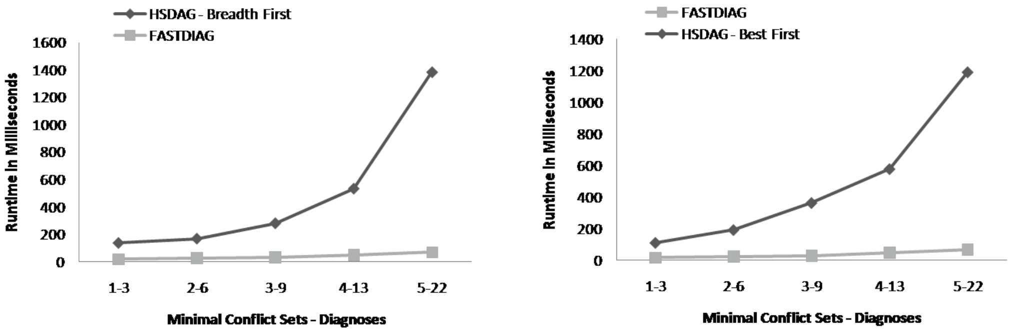


Figure 4. Calculating the *first minimal diagnosis* with FastDiag vs. hitting set based diagnosis on the basis of QuickXPlain.

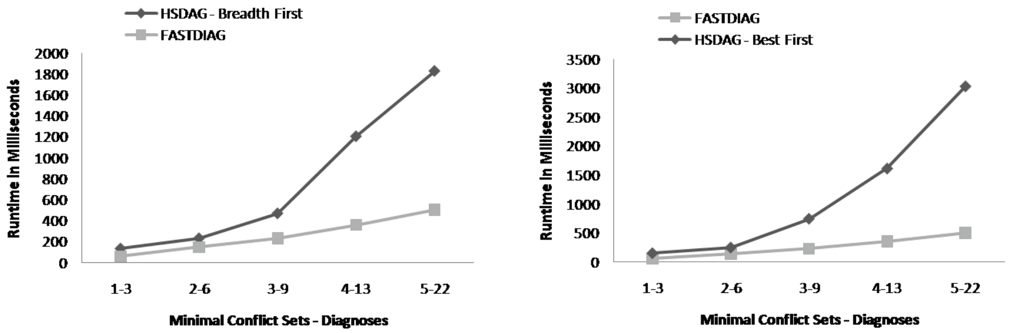


Figure 5. Calculating the *topmost-5 minimal diagnoses* with FastDiag vs. hitting set based diagnosis on the basis of QuickXPlain.

that makes it an excellent choice for interactive settings.

Empirical Evaluation. Based on a computer configuration dataset of the Graz University of Technology (N = 415 configurations) we evaluated the three presented approaches w.r.t. their capability of predicting diagnoses that are acceptable for the user (diagnoses leading to selected configurations). Each entry of the dataset consists of a set of initial *user requirements* C_R inconsistent with the configuration knowledge base C_{KB} and the *configuration* which had been finally selected by the user. Since the original requirements stored in the dataset are inconsistent with the configuration knowledge base, we could determine those diagnoses that indicated which minimal sets of requirements have to be deleted or adapted in order to be able to find a solution.

We evaluated the *prediction accuracy* of the three diagnosis approaches (*HSDAG breadth-first*, FastDiag, and *HSDAG best-first*). First, we measured the distance between the *predicted position* of a diagnosis leading to a selected configuration and the *expected position of the diagnosis* (which is 1). This distance was measured in terms of the *root mean square deviation* – RMSD (see Formula 1). Table 1 depicts the results of this first analysis. An important result is that FastDiag has the lowest RMSD value (0.95). Best-first HSDAG has a similar prediction quality (RMSD = 0.97). Finally, breadth-first HSDAG has the worst prediction quality (RMSD = 1.64).[5]

$$RMSD = \sqrt{\frac{1}{n}\sum_{1}^{n}(predicted\,position - 1)^2} \qquad (1)$$

breadth-first (HSDAG)	FastDiag	best-first (HSDAG)
1.64	0.95	0.97

Table 1. Root Mean Square Deviation (RMSD) of the diagnosis approaches.

6 Related Work

The authors of [8] introduce an algorithm for the automated debugging of configuration knowledge bases. The idea is to combine a conflict detection algorithm such as QuickXPlain [13] with the hitting set algorithm used in model-based diagnosis (MBD) [17] for the calculation of minimal diagnoses. In this context, conflicts are induced by test cases (examples) that, for example, are stemming

[5] A more detailed analysis of the prediction quality of the algorithm will be given in an extended version of this paper.

from previous configuration sessions, have been automatically generated, or have been explicitly defined by domain experts. Further applications of MBD in constraint set debugging are introduced in [7] where diagnosis concepts are used to identify minimal sets of faulty transition conditions in state charts and in [10] where MBD is applied for the identification of faulty utility constraint sets in the context of knowledge-based recommendation. In contrast to [7, 8, 10], our work provides an algorithm that allows to directly determine diagnoses without the need to determine corresponding conflict sets. FASTDIAG can be applied in knowledge engineering scenarios for calculating preferred diagnoses for faulty knowledge bases given that we are able to determine reasonable ordering for the given set of constraints – this could be achieved, for example, by the application of corresponding complexity metrics [4].

In contrast to the algorithm presented in this paper, calculating diagnoses for inconsistent requirements typically relies on the existence of (minimal) conflict sets. A well-known algorithm with a logarithmic number of consistency checks depending on the number of constraints in the knowledge base and the cardinality of the minimal conflicts – QUICKXPLAIN [13] – has made a major contribution to more efficient interactive constraint-based applications. QUICKXPLAIN is based on a divide-and-conquer strategy. FASTDIAG relies on the same principle of divide-and-conquer but with a different focus, namely the determination of minimal diagnoses. QUICKXPLAIN calculates minimal conflict sets based on the assumption of a linear preference ordering among the constraints. Similarly – if we assume a linear preference ordering of the constraints in C – FASTDIAG calculates preferred diagnoses.

The authors of [16] focus on interactive settings where users of constraint-based applications are confronted with situations where no solution can be found. In this context, [16] introduce the concept of minimal exclusion sets which correspond to the concept of minimal diagnoses as defined in [17]. As mentioned, the major focus of [16] are interactive settings where the proposed algorithm supports users in the identification of acceptable exclusion sets. The authors propose an algorithm (representative explanations) that helps to improve the quality of the presented exclusion set and thus increases the probability of finding an acceptable exclusion set for the user. Our diagnosis approach calculates preferred diagnoses in terms of a predefined ordering of the constraint set. Thus – compared to the work of [16] – we follow a different approach in terms of focusing more on preferences than on the degree of representativeness.

Many of the existing diagnosis approaches do not take into account the need for personalizing the set of diagnoses to be presented to a user. Identifying diagnoses of interest in an efficient manner is a clear surplus regarding the acceptance of the underlying application. A first step towards the application of personalization concepts in the context of knowledge-based recommendation is presented in [9]. The authors introduce an approach that calculates leading diagnoses on the basis of similarity measures used for determining n-nearest neighbors. A general approach to the identification of preferred diagnoses is introduced in [5] where probability estimates are used to determine the leading diagnoses with the overall goal to minimize the number of measurements needed for identifying a malfunctioning device. We see our work as a major contribution in this context since FASTDIAG helps to identify leading diagnoses more efficiently – further empirical studies in different application contexts are within the major focus of our future work.

7 Conclusion

In this paper we have introduced a new diagnosis algorithm (FASTDIAG) which allows the efficient calculation of one diagnosis at a time with logarithmic complexity in terms of the number of consistency checks. Thus, the computational complexity for the calculation of one minimal diagnosis is equal to the calculation of one minimal conflict set in hitting set based diagnosis approaches. The algorithm is especially applicable in settings where the number of conflict sets is equal to or larger than the number of diagnoses, or in settings where preferred (leading) diagnoses are needed.

References

[1] L. Ardissono, A. Felfernig, G. Friedrich, D. Jannach, G. Petrone, R. Schäfer, and M. Zanker. A Framework for the development of personalized, distributed web-based configuration systems. *AI Magazine*, 24(3):93–108, 2003.

[2] F. Belanger. A conjoint analysis of online consumer satisfaction. *Journal of Electronic Commerce Research*, 6:95–111, 2005.

[3] L. Castillo, D. Borrajo, and M. Salido. *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice*. IOS Press, 2005.

[4] Z. Chen and C. Suen. Measuring the complexity of rule-based expert systems. *Expert Systems with Applications*, 7(4):467–481, 2003.

[5] J. DeKleer. Using crude probability estimates to guide diagnosis. *AI Journal*, 45(3):381–391, 1990.

[6] J. DeKleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *AI Journal*, 56(2–3):197–222, 1992.

[7] A. Felfernig, G. Friedrich, K. Isak, K. Shchekotykhin, E. Teppan, and D. Jannach. Automated debugging of recommender user interface descriptions. *Journal of Applied Intelligence*, 31(1):1–14, 2007.

[8] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *AI Journal*, 152(2):213–234, 2004.

[9] A. Felfernig, G. Friedrich, M. Schubert, M. Mandl, M. Mairitsch, and E. Teppan. Plausible repairs for inconsistent requirements. In *21st International Joint Conference on Artifical Intelligence (IJCAI'09)*, pages 791–796, Pasadena, CA, 2009.

[10] A. Felfernig, G. Friedrich, E. Teppan, and K. Isak. Intelligent debugging and repair of utility constraint sets in knowledge-based recommender applications. In *13th ACM International Conference on Intelligent User Interfaces (IUI'08)*, pages 218–226, Canary Islands, Spain, 2008.

[11] G. Fleischanderl, G. Friedrich, A. Haselboeck, H. Schreiner, and M. Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.

[12] G. Friedrich and K. Shchekotykhin. A general diagnosis method for ontologies. In *4th International Semantic Web Conference (ISWC'05)*, number 3729 in Lecture Notes in Computer Science, pages 232–246, Galway, Ireland, 2005. Springer.

[13] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *19th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, San Jose, CA, 2004.

[14] J. Marques-Silva and K. Sakallah. Grasp: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, Santa Clara, CA, 1996.

[15] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *11th International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1395–1401, Detroit, MI, 1989.

[16] Barry O'Sullivan, A. Papdopoulos, B. Faltings, and P. Pu. Representative explanations for over-constrained problems. In *22nd National Conference on Artificial Intelligence (AAAI'07)*, pages 323–328, Vancouver, Canada, 2007.

[17] R. Reiter. A theory of diagnosis from first principles. *AI Journal*, 23(1):57–95, 1987.

[18] C. Sinz and A. Haag. Configuration. *IEEE Intelligent Systems*, 22(1):78–90, 2007.

[19] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[20] D. Winterfeldt and W. Edwards. Decision analysis and behavioral research. *Cambridge University Press*, 1986.

On Classification and Modeling Issues in Distributed Model-based Diagnosis

Franz Wotawa and **Ingo Pill**[1,2]

Abstract. With model-based diagnosis, diagnoses for occurring faults can be directly computed from given observations and a system model. Model-based diagnosis has been successfully accommodated to several purposes, including for instance the diagnosis of space probes and configuration knowledge bases. Recent research includes also extensions for distributed systems, motivated by the ever-growing system complexity and inherently distributed domains like service-oriented architectures. Previous work in this respect lacks however a detailed analysis and classification approach for distributed diagnosis, that considers essential underlying issues like diagnosis architecture, utilized models, and abstract requirements that might stem from the application domain. In this paper, we will show an analysis of distributed system diagnosis and a classification in the three dimensions mentioned.

1 INTRODUCTION

Model-based diagnosis [9, 10, 29] is a diagnosis approach that allows deriving possible root causes for certain misbehavior from a system's model and actual observations. The concept of model-based diagnosis has gained a lot of attention during the past decades, and several applications have been reported. These include software debugging [23, 24], as well as the diagnosis of space probes [36, 28], cars [21, 22], and configuration knowledge bases [11]. In "classical" model-based diagnosis, a diagnosis algorithm, for example Reiter's hitting set algorithm [29, 14] or GDE [10], takes the system model as well as available observations about actual system behavior, and derives possible diagnoses with respect to optimization criteria like diagnosis probability and size. Even in the case of distributed systems with subsystems connected via communication channels, all observations, as well as the system model, have to be stored and processed centralizedly.

The drawbacks of such a centralized approach have been the subject of academic discussion, e.g. [34]. For instance, as the (single) considered state space is usually aggregated from those of its subsystems, the considered space is substantially larger than when considering components locally. Since, in the worst case, diagnosis computation is exponential in the size of considered components, this is a severe computational issue. Inherently, also the robustness of the diagnostic reasoning process is weaker. A failure in the centralized diagnoser leads to a complete crash of the diagnostic reasoning. Moreover, a centralized approach might also suffer from poor scalability. Changes in the structure might require the centralized diagnoser itself to be changed, where it is even possible that the whole diagnoser has to be rebuilt. However, depending on the situation, a centralized diagnoser might not always be a bad choice. If the system is not physically distributed, or if there are special requirements like storing all failure-related data in one place, there might be no reason to use a distributed diagnosis approach.

Despite the varying needs for diagnostic reasoning in different distributed settings, to our knowledge, there is no systematic study regarding an analysis and classification of approaches considering essential features like architecture, utilized models, and application related issues. Even worse, the term "distributed diagnosis" is overloaded and refers to very different scenarios as illustrated in the following.

Distributed diagnosis might be used for example for diagnosing a system that comprises independent but interconnected subsystems, where each one has a corresponding local diagnoser. While the application seems to be almost the same as the diagnosis of a team of collaborating robots, there are subtle but essential differences. These differences lead to the fact that distributed diagnosis algorithms and systems that suit one domain, cannot be used for the other; but let us discuss an example for such an issue in detail.

In the case of the distributed system with channels for subsystem communication, original observations about system and environment attributes are most likely to be local to the corresponding subsystem (but are communicated afterwards). Hence, even in the case of distributing an observation by communication, it could not conflict with other (original) measurements of the same attribute. A corresponding application area could be that of industrial transportation systems as diagnosed centralizedly in [26].

This is different from the case of a team of autonomous robots that are equipped with sensors measuring the same or similar physical attributes. In this case, one robot might measure ambient temperature to be too high, while another one measures it to be in normal range. With independent consideration by each robot, this might lead to differing behavior that impacts overall performance. Aggregating local diagnoses might lead to deductions that stem from local diagnoses that are not compatible. Thus, an essential difference between the two domain variants is that in one case there are no contradictions between available (possibly communicated) observations, while in the other, local observations might contradict each other. In the latter case, thus (optional) inconsistency of local observations has to be taken into account in order to handle such cases correctly. Such essential differences illustrate the need for a deep analysis and classification of distributed diagnosis, as well as the necessity of defining the term distributed diagnosis and its aspects in detail.

Besides those already mentioned, there is a multitude of applications for distributed diagnosis, including settings that make use of service-oriented architectures (SOAs). SOAs aggregate numerous instances of web services, brokers, message busses, mediators, moni-

[1] Graz University of Technology, Austria, email: {wotawa,pill}@ist.tugraz.at
[2] Authors are listed in reverse alphabetical order.

tors, and other components that might be physically distributed over vast distances. In such settings a distributed diagnosis approach has significant advantages in respect of intellectual property disclosure and dynamic reconfiguration of services. While a company might be willing to provide an interface offering diagnostic data for a service, due to business and security reasons they might not want to disclose all the internal details necessary for customers to develop effective models on their own. Dynamic reconfiguration of services can be accommodated by corresponding exchanges of the local diagnosers. Another application is suggested by our example of autonomous robot teams. Settings with teams of collaborating autonomous systems might make use of distributed knowledge bases [31, 32]. Considering for instance data communication bandwidth or robustness, a distributed approach for the diagnosis of the underlying distributed knowledge base might be of advantage in such settings.

In Figure 1, we depict the architecture of a spatially distributed diagnosis system. It comprises two or more local diagnosis systems, each of which has a diagnosis engine DE_i, a model SD_i of the underlying (sub-)system, and a set of observations OBS_i. The latter represents grounded facts that are obtained from sensor information after filtering and symbol grounding. It is worth noting that, due to sensing failures and sensor noise, sensed observations might not reflect reality. However, they are the only information available for diagnostic reasoning. Another part of the discussed distributed diagnosis architecture is the global diagnosis engine. This engine is used for combining the local diagnoses and might be used also for communicating observations between the local diagnosis engines (if required). Note that in general, there is no requirement to have a global diagnosis engine (or global diagnoser) in the context of distributed diagnosis. Computing global diagnoses is also possible via communicating local diagnoses among the local diagnosis engines. In this case, the global diagnosis engine in Fig. 1 might be reduced to a communication backbone for the exchange of diagnosis results.

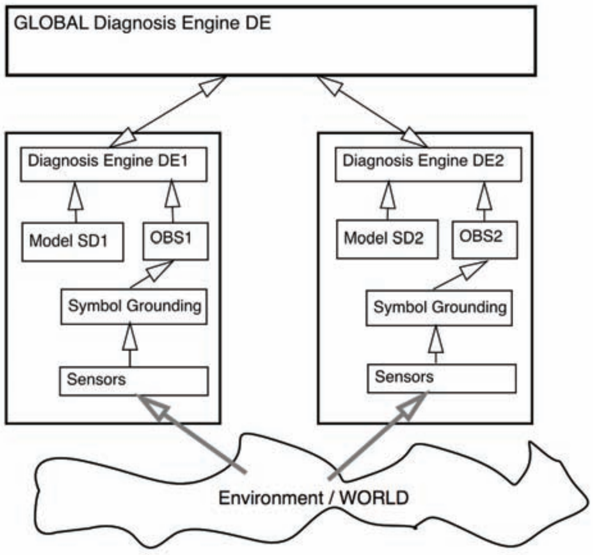


Figure 1. The distributed diagnosis architecture

In contrast to the single system model diagnosis approach, all distributed diagnosis systems have in common that they make use of local models and local diagnosis engines. Specifically, all these approaches aim at computing global diagnoses from local ones without relying on a global model. Hence, we define a distributed diagnosis system as a system aggregating local diagnosis engines (or local diagnosers), using local models and (not necessarily local) observations for the computation of local diagnoses. Subsequently, these local diagnoses are used to derive global ones. In this paper we will characterize such distributed diagnosis systems according to the underlying architecture, utilized models, and the relationship between the local models. The motivation of this paper is to analyze previous work in the domain of distributed diagnosis, the discussion of related modeling issues, and outlining a characterization of distributed diagnosis systems with respect to several essential aspects. The remainder of this document is structured as follows. In Section 2, we discuss related research, followed by an analysis of related modeling issues presented in Section 3. Section 4 contains our analysis of the distributed diagnosis problem itself and related definitions. In Section 5 we discuss our characterization of distributed diagnosis systems, while in Section 6 we draw our conclusion and give directions for future work.

2 RELATED RESEARCH

In [19], Kurien et al. introduced a basic formalism for distributed diagnosis that is based on information interchange between local diagnosers, where the intersection between the local component sets is empty. Local diagnoses with their corresponding value assignments are communicated to negotiate diagnoses that all local diagnosers agree on. In this process, inconsistencies between diagnoses from different local diagnosers are used to reduce the set of possible assignments. Thus, their approach can be classified as a distributed and decentralized approach that is based on local diagnosis systems interacting via shared connections and sharing global observations. The content of Kurien et al.'s paper is most closely to the one discussed in this paper. Similar work includes Wonham et al. [34] and Diagle et al. [8], the latter tackling the task of diagnosis in the context of mobile robots which is a very interesting domain as discussed in the introduction.

Other work in the domain of mobile robot diagnosis, specifically in the context of mobile robot interaction for achieving a common goal, includes [17]. In this paper, Kalech et al. describe an approach using distributed algorithms for the localization of faults in the team coordination. While their work is mainly focused on their particular problem domain, we are interested in classifying and solving distributed diagnosis in general.

Some colleagues outlined distributed diagnosis in the context of discrete event systems [20]. In [3], Baroni et al. extended their previous work on diagnosis of active systems [2] to the distributed case. Guillou et al. introduced in [15] the use of chronicles for distributed diagnosis, based on previous work of the same group [7]. The latter paper shows how incremental and decentralized diagnosis can be implemented effectively. Most recently, Ribot et al. discussed in [30] the use of design requirements that enable the diagnosability of discrete event systems. This work is of particular interest regarding practical applications, because it addresses the issue of how to construct system models so that the system can be diagnosed afterwards. In our work we follow very closely Reiter's model-based diagnosis theory [29], with the objective of extending it to the distributed case. For previous work in this respect, we refer the interested reader to

[37] where we discussed challenges regarding distributed diagnosis and an extension of Reiter's theory.

Practical issues like enabling diagnosis under real-time requirements have also been discussed in literature. In [5], Chung and Barrett presented the distributed on-line diagnosis of spacecrafts under real-time constraints. Their approach combines model-based diagnosis with rule-based systems, where the underlying idea is to compile models into rules that can be used on-line. The advantage is that in this case, diagnosis time can be estimated and guaranteed, so that these rules can be used efficiently in critical real-time systems.

3 MODELING

In this section, we discuss modeling issues relevant for model-based diagnosis. Although there is more than one approach for system behavior modeling, most of them rely on the well known component-connection modeling paradigm (CCMP) for practical reasons. CCMP requires the definition of the system's structure and behavior, where system behavior is defined by the behavior of each (sub-)component and interconnecting ports for information exchange between components. Usually components with the same behavior are defined as instances of a component class, which eases maintenance.

According to Reiter's original theory of diagnosis [29], only the correct behavior of components has to be given in a model. Such an approach comes with the advantage of not requiring knowledge about possible faults and their consequences. And indeed, there are many applications where either no fault models are available, or where there are too many possible fault modes. However, as Struss et al. indicate in [33], there are cases where, if consistency-based diagnosis considers only definitions of correct behavior, this leads to the computation of diagnoses with no representation in the physical world. Hence, computed diagnoses, although correct with respect to the used model, are incorrect when considering reality. In order to solve this problem, Struss et al. suggested to introduce fault models. Console et al. discussed in [6] the close relationship between abduction and deduction. In particular, using [6] we can state that consistency-based diagnosis using fault models is computationally equivalent to abduction-based diagnosis.

Motivated by the impact on computational complexity that is entailed by introducing fault models, Gottlob et al. suggested to add physical impossibilities to models [13]. A physical impossibility is a behavioral rule that has to be fulfilled in all circumstances, and describes the relationship between connection values and the status of components. For example, let us consider an electronic circuit that comprises a battery and two parallel bulbs. If we observe that one bulb shines while the other does not, we are able to conclude that (a) the battery works correctly and that (b) the bulb not shining is broken. This results from the fact that the battery has to be working, as without power no bulb could shine. Lacking corresponding fault models, we can use physical impossibilities to obtain the same effect. We state that it cannot be the case that the battery is not working (or not providing voltage) if a bulb is shining. From a more general perspective, physical impossibilities are similar to invariants, e.g. loop invariants or class invariants as used in verification. We follow this more general term of invariants and consider physical impossibilities as system invariants. Such system invariants have only little influence on the computational complexity, but contradict CCMP as they are not assigned to a component but to the system itself.

An extension to CCMP is the concept of hierarchical models. A model is called hierarchical, if a component model itself is implemented using CCMP. Hence, if CCMP is used recursively to aggregate a system from components, subcomponents, ..., as well as necessary connections. Igor Mozetič was the first who published the idea of hierarchical models [25]. Autio and Reiter [1] introduced a formal definition of hierarchical models including results regarding diagnosis capabilities. It is worth noting that hierarchical models are used for two reasons. The first is related to the modeling process itself. Using hierarchical models, components can be modeled from available simpler components, taking advantage of all the possibilities entailed with modularization and related libraries. Hence, modeling becomes easier and more convenient. The other reason concerns computational issues. On the top level, diagnosis has to consider only a smaller set of components, due to the abstraction performed when subsuming several components to form one hierarchical component. Since, in the worst case, the computation of diagnoses is exponential in the number of components, structuring systems in a hierarchic way saves computation time.

Academic literature offers some modeling languages for model-based diagnosis. In [12, 16] the authors introduce the modeling language AD2L that is based on CCMP, and allows describing system models including fault models and system invariants. More recently, Provan and Wang [27, 35] suggested a benchmark generator and later on a language for sharing models to be used for performance evaluation of diagnosis algorithms.

In distributed diagnosis, (local) models follow the same principles like models used in the non-distributed case. Hence, the models most likely rely on CCMP and (not necessarily) implement fault models, invariants, or hierarchical components. Having a closer look at the underlying formal modeling methods, we see that they vary from discrete event systems [20] to logic [29] and bond graphs [8]. The reason for choosing a certain format might stem from the requirements regarding temporal aspects. If temporal reasoning is not necessary at all, the modeling format can be simpler. In this case, even simple horn-clause propositional logic might be sufficient.

4 DISTRIBUTED DIAGNOSIS

In this section, we will introduce several definitions which we will use for the discussion of classification criteria for distributed diagnosis. In [29] Reiter introduced several formal definitions for model-based diagnosis. While we will rely on those definitions in principle, we will extend them to accommodate the scenario of distributed diagnosis.

Definition 1 (Diagnosis System) *A diagnosis system DS is a tuple* $(SD, COMP, CONN, PE, \Psi)$ *where SD is a set of logical sentences describing the system's behavior, COMP is the set of components, CONN the set of connections, PE the set of physical entities, and* Ψ *is a function mapping connections to their corresponding physical entities. For simplicity, we assume the presence of functions* $comp(DS)$ *and* $conn(DS)$ *that, given a diagnosis system, return the corresponding sets of components and connections respectively.*

The reason for separating connections from physical entities is that this allows us to differentiate between entities at model level and real (e.g. physical) system entities. This separation is necessary, so that we can reason about scenarios, where two system parts measure the same physical entity (e.g. ambient temperature), but do this in a different way. Hence, in this case there might be inconsistencies in the observations that have to be dealt with.

According to [29], a diagnosis problem comprises a diagnosis system and a set of observations, where in the context of this paper, an

observation is an assignment of a value to a connection. In practice, such observations are either stated by some user, or are derived from sensors via symbol grounding. For computing the diagnoses, it is assumed that observations and model are correct and reflect the real world scenario. As without further knowledge the incorrectness of observations cannot be stated, this is a useful assumption. However, as we will see later, this might not be the case for distributed diagnosis systems where there might be more than one measurement of the same physical attribute.
Regarding the definition of an actual diagnosis, compared to [29] we give a slightly modified version in order to be self-compliant with Definition 1.

Definition 2 (Diagnosis) *Let $(SD, COMP, CONN, PE, \Psi)$ be a diagnosis system and OBS be a set of observations. A set $\Delta \subseteq COMP$ is a diagnosis iff $SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$ is satisfiable.*

For our distributed scenario, we define a distributed diagnosis system to be a set of local diagnosis systems that belong to a global diagnosis system.

Definition 3 (Distributed Diagnosis System (DDS)) *Let $(SD, COMP, CONN, PE, \Psi)$ be a global diagnosis system. A set $\{DS_i = (SD_i, COMP_i, CONN_i, PE, \Psi) | i \in \{1, \ldots, n\}\}$ is a distributed diagnosis system (DSS) comprising n local diagnosis systems iff the following conditions are satisfied:*

1. $\forall_{i=1}^{n} : SD_i \subseteq SD$
2. $(\cup_{i=1}^{n} COMP_i) = COMP$
3. $(\cup_{i=1}^{n} CONN_i) = CONN$

The reason for binding the DDS to a global diagnosis system is that this allows us to define the correctness and completeness of distributed diagnosis algorithms. It is worth noting that we use a global set of physical entities for the local systems on purpose. This emphasizes the fact that some $p \in PE$ might be shared, a situation that might lead to problems with observations as discussed before. Furthermore, for obvious reasons this set is equal to the set of physical entities for the global diagnosis system.

In order to apply a diagnosis algorithm to a distributed diagnosis problem, we first have to define this problem. Following the definition of a diagnosis problem, we state a distributed diagnosis problem as follows:

Definition 4 (Distributed Diagnosis Problem) *A distributed diagnosis problem comprises a DDS $\{DS_i | i \in \{1, \ldots, n\}\}$ and a set of observations $\{OBS_i | i \in \{1, \ldots, n\}\}$ where OBS_i are the observations for the corresponding local diagnosis system DS_i.*

One characterization of distributed diagnosis is that the local diagnosis systems from a DDS, and their corresponding observations, are used to compute local diagnoses following Definition 2. Subsequently, these local diagnoses are combined to obtain global ones. The correctness and completeness of such an algorithm depends on the integration mechanism utilized. Formally, we are able to define the correctness and completeness with respect to global and local models as well as global and local observations. Note that in general, due to possible inconsistencies, the set of global observations cannot be obtained by taking the union of local observations. Simple approaches like majority vote or using fault probabilities defined statically by a sensor's characteristics or dynamically by fault history, as well as more elaborate approaches like [18] provide the means to resolve such inconsistencies [4].

Definition 5 (Correctness, Completeness) *Let $(SD, COMP, CONN, PE, \Psi)$ be a global diagnosis system, $\{DS_i | i \in \{1, \ldots, n\}\}$ its corresponding DDS, OBS the global observations, and $\{OBS_i | i \in \{1, \ldots, n\}\}$ the local observations. A distributed diagnosis algorithm DD is correct iff all computed diagnoses using the DDS and the local observations are also diagnoses of the global diagnosis problem. A distributed diagnosis algorithm DD is complete iff all global diagnoses are computed.*

In Definition 5, correctness and completeness are defined using the global diagnosis system as reference. The correctness and completeness of already published distributed diagnosis algorithms usually requires additional assumptions. For example, it is often assumed that DDS do not share components and that local observations are never in contradiction. While these assumptions are appropriate for several domains, there are applications, like autonomous mobile robot teams, where these assumptions are invalid. Consider, for instance, our example of a team of robots working on the same task. Each robot is perceiving the world via its sensors, and each robot's diagnosis system is relying on its sensor information to reflect the state of the real world. While the robots are perceiving the same physical entities, due to symbol grounding they might use different corresponding observations. Therefore, they do not share observations, and as a consequence, local observations might be in contradiction. Moreover, the robots may compute diagnoses that reflect their individually perceived correctness or incorrectness of real world entities. When each robot uses the same model, the intersection of considered components is not empty. Hence, these assumptions are not valid in this application domain, which has consequences on the choice regarding the use of a specific distributed diagnosis algorithm.

In order to let us characterize DDS and thus corresponding diagnosis algorithms, we further partition the space of DDS into subclasses. We start with a DDS where the intersection of the local components is empty.

Definition 6 (Partitioned DDS) *A DDS $\{DS_i | i \in \{1, \ldots, n\}\}$ is a partitioned DDS iff*
$\forall i, j \in \{1, \ldots, n\} : i \neq j \rightarrow comp(DS_i) \cap comp(DS_j) = \emptyset.$

In a partitioned DDS, the local diagnosis systems do not share any component. In order to further divide the DDS space, we introduce structural independent DDSs, where even no connections are shared.

Definition 7 (Structural Independent DDS) *A partitioned DDS $\{DS_i | i \in \{1, \ldots, n\}\}$ is a structural independent DDS iff*
$\forall i, j \in \{1, \ldots, n\} : i \neq j \rightarrow conn(DS_i) \cap conn(DS_j) = \emptyset.$

The definition of structural independence is not enough to ensure that local observations are not in contradiction, due to different sensor measurements used by the local diagnosis systems. To ensure real independence in a distributed setting, we have to apply further restrictions on the measured physical entities.

Definition 8 (Independent DDS) *A structural independent DDS $\{DS_i | i \in \{1, \ldots, n\}\}$ is an independent DDS iff*
$$\forall i, j \in \{1, \ldots, n\} : i \neq j \rightarrow \{\Psi(c) | c \in conn(DS_i)\} \cap \{\Psi(d) | d \in conn(DS_j)\} = \emptyset.$$

In the case of an independent DDS, it is ensured that local observations cannot intersect. Hence, a simple algorithm that computes local diagnoses and puts them together by computing all possible combinations is correct and complete. For a structural independent DDS, such a simple algorithm can only be used if the local observations are never in contradiction.

A different branch of DDS, which occurs often in practice, is that where a global diagnosis system is partitioned into connected subsystems. Many of todays systems like power supplies or telecommunication networks fall into this category.

Definition 9 (Connected DDS) *A partitioned DDS* $\{DS_i | i \in \{1, \ldots, n\}\}$ *is a connected DDS iff*

$$\forall i \in \{1, \ldots, n\} : \exists j \in \{1, \ldots, n\} : i \neq j \wedge conn(DS_i) \cap conn(DS_j) \neq \emptyset.$$

Let $\mathbf{DDS}$ be the set of all DDS, $\mathbf{PDDS}$ the set of partitioned DDS, $\mathbf{SIDSS}$ the set of structural independent DDS, $\mathbf{IDSS}$ the set of independent DDS, and $\mathbf{CDSS}$ the set of connected DDS. Then, from the definitions above, we are able to easily identify the following relationships:

Corollary 1 *The following relationships hold for the different classes of DDS:*

- $\mathbf{DSS} \supset \mathbf{PDDS} \supset \mathbf{SIDSS} \supset \mathbf{IDSS}$
- $\mathbf{PDDS} \supset \mathbf{CDSS}$
- $\mathbf{CDSS} \cap \mathbf{SIDSS} = \emptyset$

Note that, as the relationships directly follow from the definitions, proofs are omitted.

5 CHARACTERIZING DISTRIBUTED DIAGNOSIS SYSTEMS

After discussing modeling aspects in the context of model-based diagnosis, introducing the notions of the distributed diagnosis problems and related distributed diagnosis systems along several definitions for a sensible classification, we are now able to characterize distributed diagnosis. In the following, we will discuss this characterization along three dimensions; modeling, classification of DDS, and used architecture.

- **Model:** There are many modeling paradigms and modeling languages used in practice. One might use discrete event systems, finite state machines, or a simple non-temporal logic for describing the behavior of a system. The underlying models might follow the component-connection modeling paradigm, with or without making use of physical impossibilities, fault models, or modeling hierarchies. The different modeling paradigms and styles can be used to characterize diagnosis systems and therefore also DDS. In case of DDS, the choice of the modeling language and paradigm has an impact on the partitioning of the local diagnosis models. However, the choice has no direct influence on the combination of local diagnoses in order to obtain global ones, if the local diagnosis engines only return subsets of components as diagnosis candidates (as stated in definition of diagnosis: Def. 2). Nevertheless, the underlying modeling language and paradigm have an impact on the choice of the underlying local diagnosis algorithm and theorem provers used. Hence, a characterization of DDS according to the model is useful to select the right local diagnosis engine and theorem provers.
- **DDS class:** A distributed diagnosis problem that belongs to a certain application area can be characterized according to the DDS classes introduced in this paper. This characterization heavily depends on the application scenario. In case of distributed autonomous systems, a general DDS or a connected DDS might be used because further restrictions do not apply. In case of networks a clear separation of components is possible and observations can be exchanged over the sharing connections only. Hence, a connected DDS might be sufficient. Once a characterization of diagnosis algorithms according to DDS is available, such a classification would allow to select correct and complete algorithms for different problems. However, even when such a classification of algorithms is not available, the characterization into DDS classes helps to understand the problems that have to be tackled when developing an algorithm. For example, in case of a structural independent DDS, someone has to be aware that there might be contradicting observations, which have to be handled in the right way.
- **Architecture:** Regarding the implementation of DDS there are two general architectures one can follow. The centralized distributed diagnosis is characterized by a central global diagnosis engine that takes the local diagnosis results and combines them to form global diagnoses. Such a centralized diagnosis engine is also capable of distributing observations and performing measurement selection. The advantage is that obtaining global diagnoses is easier from the algorithmic point of view and that communication can be controlled globally, which might lead to less messages necessary to obtain global diagnoses. The disadvantage is that the approach is less robust. In case of a fault in the global diagnosis engine, there is no way of coming up with a global diagnostic view.

  In the decentralized architecture, the local diagnosis engines communicate their results and compute a global view without using a central diagnosis engine. The advantage is that a decentralized approach is more robust and that faults in one part should have only local influence. However, there is a communication overhead because all necessary information has to be communicated to all connected subsystems.

6 CONCLUSION

In this paper, we discussed modeling aspects in the context of model-based diagnosis, analyzed the distributed diagnosis problem, introduced corresponding definitions, suggest a specific classification of DDS, and propose a characterization of competing approaches regarding essential problem aspects. The three dimensions proposed are modeling, classification of DDS, and used architecture. While a lot of applications in the broad context of distributed diagnosis seem to be similar, we showed by simple examples that there are significant differences that have to be taken into account when selecting an appropriate algorithm for a specific problem. With our characterization, we enable the selection of the right diagnosis algorithm and architecture, given a distributed diagnosis problem in a certain application area. Currently missing is a classification of already published distributed diagnosis algorithms using the classifications given in this paper. However, even without such a classification, the characterization into DDS classes helps in understanding the problems that have to be tackled when developing an algorithm. For example, in the case of a structural independent DDS, we are made aware that possibly contradicting observations have to be taken care of.

Future work will include classifying previous DDS approaches using our classification framework. Furthermore, we are interested in providing algorithms including correctness and completeness proofs, as well as in identifying specific algorithmic areas and aspects that should be covered by future research. An important problem we will tackle in future research is that of solving diagnosis problems for autonomous mobile robots in a distributed way.

ACKNOWLEDGEMENTS

The work presented in the paper has been funded by the Austrian Science Fund (FWF) under contract number P20199-N15. The authors would also like to thank the reviewers for their valuable comments.

REFERENCES

[1] K. Autio and R. Reiter, 'Structural abstraction in model-based diagnosis', in *13th European Conference on Artificial Intelligence (ECAI)*, pp. 269–273, (1998).

[2] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella, 'Diagnosis of large active systems', *Artificial Intelligence*, **110**, 135–183, (1999).

[3] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella, 'Diagnosis of a class of distributed discrete-event systems', *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, **30**(6), 731–752, (2000).

[4] R. R. Brooks and S.S. Iyengar, *Multi-sensor fusion: fundamentals and applications with software*, 1998. ISBN 0-13-901653-8.

[5] S. H. Chung and A. Barrett, 'Distributed real-time model-based diagnosis', in *IEEE Aerospace Conference*, (2003).

[6] L. Console, D. T. Dupré, and P. Torasso, 'On the relationship between abduction and deduction', *Journal of Logic and Computation*, **1**(5), 661–690, (1991).

[7] M. O. Cordier and A. Grastien, 'Exploiting independence in a decentralised and incremental approach of diagnosis', in *17th International Workshop on Principles of Diagnosis (DX)*, (2006).

[8] M. Daigle, X. Koutsoukos, and G. Biswas, 'Distributed diagnosis of coupled mobile robots', in *IEEE International Conference on Robotics and Automation*, pp. 3787–3794, (2006).

[9] R. Davis, 'Diagnostic reasoning based on structure and behavior', *Artificial Intelligence*, **24**, 347–410, (1984).

[10] J. de Kleer and B. C. Williams, 'Diagnosing multiple faults', *Artificial Intelligence*, **32**(1), 97–130, (1987).

[11] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, 'Consistency-based diagnosis of configuration knowledge bases', *Artificial Intelligence*, **152**(2), 213–234, (2004).

[12] G. Fleischanderl, H. Schreiner, T. Havelka, M. Stumptner, and F. Wotawa, 'An environment and language for industrial use of model-based diagnosis', in *ECAI Workshop on Knowledge-Based Engineering*, (2000).

[13] G. Friedrich, G. Gottlob, and W. Nejdl, 'Physical impossibility instead of fault models', in *8th AAAI Conference on Artificial Intelligence*, pp. 331–336, (1990). Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann, 1992).

[14] R. Greiner, B. A. Smith, and R. W. Wilkerson, 'A correction to the algorithm in Reiter's theory of diagnosis', *Artificial Intelligence*, **41**(1), 79–88, (1989).

[15] X. Le Guillou, M. O. Cordier, S. Robin, and L. Rozé, 'Chronicles for on-line diagnosis of distributed systems', in *18th European Conference on Artificial Intelligence (ECAI)*, pp. 194–198, (2008).

[16] Th. Havelka, M. Stumptner, and F. Wotawa, 'AD2L- A Programming Language for Model-Based Systems (Preliminary Report)', in *11th International Workshop on Principles of Diagnosis (DX)*, (2000).

[17] M. Kalech and G. A. Kaminka, 'On the design of coordination diagnosis algorithms for teams of situated agents', *Artificial Intelligence*, **171**(8–9), 491–513, (2007).

[18] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli, 'On-line fault detection of sensor measurements', in *IEEE Sensors*, pp. 974–979, (2003). ISBN 0-7803-8133-5.

[19] J. Kurien, X. Koutsoukos, and F. Zhao, 'Distributed diagnosis of networked hybrid systems', in *AAAI Spring Symposium on Information Refinement and Revision for Decision Making: Modeling for Diagnostics, Prognostics, and Prediction*, pp. 37–44, (2002).

[20] G. Lamperti and M. Zanella, *Diagnosis of Active Systems*, 2003. ISBN 978-1-4020-7487-5.

[21] A. Malik, P. Struss, and M. Sachenbacher, 'Qualitative modeling is the key – a successful feasibility study in automated generation of diagnosis guidelines and failuer mode and effects analysis for mechatronic car subsystems', in *6th International Workshop on Principles of Diagnosis (DX)*, (1995).

[22] A. Malik, P. Struss, and M. Sachenbacher, 'Case studies in model-based diagnosis and fault analysis of car-subsystems', in *12th European Conference on Artificial Intelligence (ECAI)*, (1996).

[23] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, 'Can AI help to improve debugging substantially? debugging experiences with value-based models', in *15th Eureopean Conference on Artificial Intelligence (ECAI)*, pp. 417–421, (2002).

[24] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, 'Towards an integrated debugging environment', in *15th Eureopean Conference on Artificial Intelligence (ECAI)*, pp. 422–426, (2002).

[25] I. Mozetič, 'Hierarchical model-based diagnosis', *International Journal of Man-Machine Studies*, **35**, 329–362, (1991).

[26] I. Pill, G. Steinbauer, and F. Wotawa, 'A practical approach for the online diagnosis of industrial transportation systems', in *7th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*, pp. 1318–1323, (2009).

[27] G. M. Provan and J. Wang, 'Automated benchmark model generators for model-based diagnostic inference', in *20th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 513–518, (2007).

[28] K. Rajan, D. E. Bernard, G. Dorais, E. B. Gamble, B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. P. Nayak, N. F. Rouquette, B. D. Smith, W. Taylor, and Y. Tung, 'Remote agent: An autonomous control system for the new millennium', in *14th European Conference on Artificial Intelligence (ECAI)*, pp. 726–730, (2000).

[29] R. Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57–95, (1987).

[30] P. Ribot, Y. Pencolé, and M. Combacau, 'Design requirements for the diagnosability of distributed discrete event systems', in *19th International Workshop on Principles of Diagnosis (DX)*, (2008).

[31] J. Schweiger, K. Ghandri, and A. Koller, 'Concepts of a distributed real-time knowledge base for teams of autonomous systems', in *International Conference on Intelligent Robots and Systems*, pp. 1508–1515, (1994).

[32] S. Stoehr, 'Using a distributed knowledge base to coordinate autonomous mobile systems', in *7th International Workshop on Database and Expert Systems Applications (DEXA)*, pp. 172–177, (1996).

[33] P. Struss and O. Dressler, 'Physical negation — Integrating fault models into the general diagnostic engine', in *11th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1318–1323, (1989).

[34] R. Su, W.M. Wonham, J. Kurien, and X. Koutsoukos, 'Distributed diagnosis for qualitative systems', in *6th International Workshop on Discrete Event Systems (WODES)*, (2002).

[35] J. Wang and G. M. Provan, 'Generating application-specific benchmark models for complex systems', in *23rd AAAI conference on Artificial intelligence*, pp. 566–571, (2008).

[36] B. C. Williams and P. P. Nayak, 'Immobile robots – AI in the new millennium', *AI Magazine*, 16–35, (1996).

[37] F. Wotawa and J. Weber, 'Challenges of distributed model-based diagnosis', in *23rd International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA/AIE)*, (2010).

Diagnosis discrimination for ontology debugging

Kostyantyn Shchekotykhin and **Gerhard Friedrich** [1]

Abstract. Debugging is an important prerequisite for the widespread application of ontologies, especially in areas that rely upon everyday users to create and maintain knowledge bases, such as the Semantic Web. Most recent approaches use diagnosis methods to identify sources of inconsistency. However, in most debugging cases these methods return many alternative diagnoses, thus placing the burden of fault localization on the user. This paper demonstrates how the target diagnosis can be identified by performing a sequence of observations, that is, by querying an oracle about entailments of the target ontology. We exploit probabilities of typical user errors to formulate information theoretic concepts for query selection. Our evaluation showed that the suggested method reduces the number of required observations compared to myopic strategies.

1 Introduction

The application of semantic systems, including the Semantic Web technology, is largely based on the assumption that the development of ontologies can be accomplished efficiently even by every day users. For such users and also for experienced knowledge-engineers the identification and correction of erroneous ontological definitions can be an extremely hard task. Ontology debugging tools simplify the development of ontologies by localizing a set of axioms that should be modified in order to formulate the intended target ontology.

To debug an ontology a user must specify some requirements such as coherence and/or consistency. Additionally, one can provide test cases [3] which must be fulfilled by the target ontology $\mathcal{O}_t$. A number of diagnosis methods have been developed [11, 6, 3] to pinpoint alternative sets of possibly faulty axioms (called a set of diagnoses). A user has to change at least all of the axioms of one diagnosis in order to satisfy all of the requirements and test cases.

However, the diagnosis methods can return many alternative diagnoses for a given set of test cases and requirements. In such cases it is unclear how to identify the target diagnosis. A possible solution would be to introduce an ordering based on some preference criteria. For instance, Kalyanpur et al. [7] suggest measures to rank the axioms of a diagnosis depending on their structure, occurrence in test cases, etc. Only the top ranking diagnoses are then presented to the user. Of course this set of diagnoses will contain the target one only in the case where a faulty ontology, the given requirements and test cases provide enough information for appropriate heuristics. However, in most debugging sessions a user has to provide additional information (e.g. in the form of tests) to identify the target diagnosis.

In this paper we present an approach to acquire additional information by generating a sequence of queries that are answered by some oracle such a user, an information extraction system, etc. Each answer to a query reduces the set of diagnoses until finally the target diagnosis is identified. In order to construct queries we exploit the property that different diagnoses imply unequal sets of axioms. Consequently, we can differentiate between diagnoses by asking the oracle if the target ontology should imply an axiom or not. These axioms can be generated by classification and realization services provided in description logic reasoning systems [12, 4]. In particular, the classification process computes a subsumption hierarchy (sometimes also called "inheritance hierarchy" of parents and children) for each concept name mentioned in a TBox. For each individual mentioned in an ABox, realization computes the atomic concepts (or concept names) of which the individual is an instance [12].

In order to generate the most informative query we exploit the fact that some diagnoses are more likely than others because of typical user errors. The probabilities of these errors can be used to estimate the change in entropy of the set of diagnoses if a particular query is answered. We select those queries which minimize the expected entropy, i.e. maximize the information gain. An oracle should answer these queries until a diagnosis is identified whose probability is significantly higher than those of all other diagnoses. This diagnosis is the most likely to be the target one.

We compare our entropy-based method with a greedy approach that selects those queries which try to cut the number of diagnoses in half. The evaluation shows that on average the suggested entropy-based approach is at least 50% better than the greedy one.

The remainder of the paper is organized as follows: Section 2 presents two introductory examples as well as the basic concepts. The details of the entropy-based query selection method are given in Section 3. Section 4 describes the implementation of the approach and is followed by evaluation results in Section 5. The paper concludes with an overview of related work.

2 Motivating examples and basic concepts

In order to explain the fundamentals of our approach let us introduce two examples.

Example 1 Consider a simple ontology $\mathcal{O}$ with the terminology $\mathcal{T}$:

$$ax_1 : A \sqsubseteq B \quad ax_2 : B \sqsubseteq C \quad ax_3 : C \sqsubseteq Q \quad ax_4 : Q \sqsubseteq R$$

and the background theory $\mathcal{A} : \{A(w), \neg R(w)\}$. Let the user explicitly define that the two assertional axioms should be considered as correct.

The ontology $\mathcal{O}$ is inconsistent and the only irreducible set of axioms (minimal conflict set) that preserves the inconsistency is $CS : \{\langle ax_1, ax_2, ax_3, ax_4\rangle\}$. That is one has to modify or remove at least one of the following axioms $\{\{ax_1\}, \{ax_2\}, \{ax_3\}, \{ax_4\}\}$ to restore the consistency of the ontology. However it is unclear, which ontology from a set of consistent ontologies (diagnoses) $\mathbf{D} : \{\mathcal{D}_1 \ldots \mathcal{D}_4\}$, where $\mathcal{D}_i = \mathcal{O} \setminus \{ax_i\}$, corresponds to the target one.

$$\mathcal{D}_1 : [ax_1] \quad \mathcal{D}_2 : [ax_2] \quad \mathcal{D}_3 : [ax_3] \quad \mathcal{D}_4 : [ax_4]$$

In order to focus on the essentials of our approach we employ the following simplified definition of diagnosis without limiting its generality. A more detailed version can be found in [3].

We allow the user to define of a background theory (represented as a set of axioms) which is considered to be correct, a set of logical

[1] University Klagenfurt, Austria, email: firstname.lastname@uni-klu.ac.at

sentences which must be implied by the target ontology and a set of logical sentences which must *not* be implied by the target ontology. Δ is a set of axioms which are assumed to be faulty.

Definition 1: *Given a diagnosis problem* $\left\langle \mathcal{O}, B, T^{\models}, T^{\not\models} \right\rangle$ *where* $\mathcal{O}$ *is an ontology,* B *a background theory,* $T^{\models}$ *a set of logical sentences which must be implied by the target ontology* $\mathcal{O}_t$*, and* $T^{\not\models}$ *a set of logical sentences which must* not *be implied by* $\mathcal{O}_t$*.*

A diagnosis is a partition of $\mathcal{O}$ *in two disjoint sets* $\mathcal{D}$ *and* Δ *(*$\mathcal{D} = \mathcal{O} \setminus \Delta$*) s.t.* $\mathcal{D}$ *can be extended by a logical description* EX *and* $\mathcal{D} \cup B \cup EX \models t^{\models}$ *for all* $t^{\models} \in T^{\models}$ *and* $\mathcal{D} \cup B \cup EX \not\models t^{\not\models}$ *for all* $t^{\not\models} \in T^{\not\models}$*.*

A diagnosis $(\mathcal{D}, \Delta)$ is minimal if there is no proper subset of the faulty axioms $\Delta' \subset \Delta$ such that $(\mathcal{D}', \Delta')$ is a diagnosis. The following proposition allows us to characterize diagnoses without the extension EX. The idea is to use the sentences which must be implied to approximate EX.

Corollary 1: *Given a diagnosis problem* $\left\langle \mathcal{O}, B, T^{\models}, T^{\not\models} \right\rangle$*, a partition of* $\mathcal{O}$ *in two disjoint sets* $\mathcal{D}$ *and* Δ *is a diagnosis iff* $\mathcal{D} \cup B \cup \{\bigwedge_{t^{\models} \in T^{\models}} t^{\models}\} \cup \neg t^{\not\models}$ *consistent for all* $t^{\not\models} \in T^{\not\models}$*.*

In the following we assume that a diagnosis always exists under the (reasonable) condition that the background theory together with the axioms in $T^{\models}$ and the negation of axioms in $T^{\not\models}$ are mutually consistent. For the computation of diagnoses the set of conflicts is usually employed.

Definition 2: *Given a diagnosis problem* $\left\langle \mathcal{O}, B, T^{\models}, T^{\not\models} \right\rangle$*, a conflict* CS *is a subset of* $\mathcal{O}$ *s.t. there is a* $t^{\not\models} \in T^{\not\models}$ *and* $CS \cup B \cup \{\bigwedge_{t^{\models} \in T^{\models}} t^{\models}\} \cup \neg t^{\not\models}$ *is inconsistent.*

A conflict is the part of the ontology that preserves the inconsistency. Note, incoherence is just a special case of inconsistency enforced by background axioms or recognized by built-in reasoning services. A minimal conflict CS has no proper subset which is a conflict. $(\mathcal{D}, \Delta)$ is a (minimal) diagnosis iff Δ is a (minimal) hitting set of all (minimal) conflict sets [10]. In the following we represent a diagnosis by the set of axioms $\mathcal{D}$ assumed to be correct.

In order to differentiate between the minimal diagnoses $\{\mathcal{D}_1 \dots \mathcal{D}_4\}$ an oracle can be queried for information about the entailments of the target ontology. For instance, in our example the diagnoses have the following entailments provided by the realization of the ontology: $\mathcal{D}_1 : \emptyset$, $\mathcal{D}_2 : \{B(w)\}$, $\mathcal{D}_3 : \{B(w), C(w)\}$, and $\mathcal{D}_4 : \{B(w), C(w), Q(w)\}$. Based on these entailments we can ask the oracle whether the target ontology has to entail $Q(w)$ or not ($\mathcal{O}_t \not\models Q(w)$). If the answer is *yes* (which we model with the boolean value 1), then $Q(w)$ is added to $T^{\models}$ and $\mathcal{D}_4$ is the target diagnosis. All other diagnoses are rejected because $\mathcal{D}_i \cup B \cup \{Q(w)\}$ for $i = 1, 2, 3$ is inconsistent. If the answer is *no* (which we model with the boolean value 0), then $Q(w)$ is added to $T^{\not\models}$ and $\mathcal{D}_4$ is rejected as $\mathcal{D}_4 \cup B \models Q(w)$ (rsp. $\mathcal{D}_4 \cup B \cup \neg Q(w)$ is inconsistent) and we have to ask the oracle another question.

Property 1: *Given a diagnosis problem* $\left\langle \mathcal{O}, B, T^{\models}, T^{\not\models} \right\rangle$*, a set of diagnoses* $\mathbf{D}$*, and a set of logical sentences* X *representing the query* $\mathcal{O}_t \models X$ *? :*

If the oracle gives the answer 1 *then every diagnosis* $\mathcal{D}_i \in \mathbf{D}$ *is a diagnosis for* $T^{\models} \cup X$ *iff* $\mathcal{D}_i \cup B \cup \{\bigwedge_{t^{\models} \in T^{\models}} t^{\models}\} \cup \{X\} \cup \neg t^{\not\models}$ *is consistent for all* $t^{\not\models} \in T^{\not\models}$*.*

If the oracle gives the answer 0 *then every diagnosis* $\mathcal{D}_i \in \mathbf{D}$ *is a diagnosis for* $T^{\not\models} \cup \{X\}$ *iff* $\mathcal{D}_i \cup B \cup \{\bigwedge_{t^{\models} \in T^{\models}} t^{\models}\} \cup \neg X$ *is consistent.*

Note, a set X corresponds to a logical sentence where all elements of X are connected by $\wedge$. This defines the semantics of $\neg X$.

As possible queries we consider sets of entailed concept definitions provided by a classification service and sets of individual assertions provided by realization. In fact, the intention of classification is that a model for a specific application domain can be verified by exploiting the subsumption hierarchy [1].

Table 1. Possible queries in Example 1

Query	$\mathbf{D^X}$	$\mathbf{D^{\neg X}}$	$\mathbf{D^{\emptyset}}$
$X_1 : \{B(w)\}$	$\{\mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4\}$	$\{\mathcal{D}_1\}$	$\emptyset$
$X_2 : \{C(w)\}$	$\{\mathcal{D}_3, \mathcal{D}_4\}$	$\{\mathcal{D}_1, \mathcal{D}_2\}$	$\emptyset$
$X_3 : \{Q(w)\}$	$\{\mathcal{D}_4\}$	$\{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3\}$	$\emptyset$

One can use different methods to select the best query in order to minimize the number of questions asked to the oracle. For instance, a simple "split-in-half" heuristic prefers queries which remove half of the diagnoses from the set $\mathbf{D}$. To apply this heuristic it is essential to compute the set of diagnoses that can be rejected depending on the query outcome. For a query X the set of diagnoses $\mathbf{D}$ can be partitioned in sets of diagnoses $\mathbf{D^X}$, $\mathbf{D^{\neg X}}$ and $\mathbf{D^{\emptyset}}$ where

- for each $\mathcal{D}_i \in \mathbf{D^X}$ it holds that $\mathcal{D}_i \cup B \cup \{\bigwedge_{t^{\models} \in T^{\models}} t^{\models}\} \models X$
- for each $\mathcal{D}_i \in \mathbf{D^{\neg X}}$ it holds that $\mathcal{D}_i \cup B \cup \{\bigwedge_{t^{\models} \in T^{\models}} t^{\models}\} \models \neg X$
- $\mathbf{D^{\emptyset}} = \mathbf{D} \setminus (\mathbf{D^X} \cup \mathbf{D^{\neg X}})$

Given a diagnosis problem we say that the diagnoses in $\mathbf{D^X}$ predict 1 as a result of the query X, diagnoses in $\mathbf{D^{\neg X}}$ predict 0, and diagnoses in $\mathbf{D^{\emptyset}}$ do not make any predictions.

Property 2: *Given a diagnosis problem* $\left\langle \mathcal{O}, B, T^{\models}, T^{\not\models} \right\rangle$*, a set of diagnoses* $\mathbf{D}$*, and a query* X*:*

If the oracle gives the answer 1 *then the set of rejected diagnoses is* $\mathbf{D^{\neg X}}$ *and the set of remaining diagnoses is* $\mathbf{D^X} \cup \mathbf{D^{\emptyset}}$*.*

If the oracle gives the answer 0 *then the set of rejected diagnoses is* $\mathbf{D^X}$ *and the set of remaining diagnoses is* $\mathbf{D^{\neg X}} \cup \mathbf{D^{\emptyset}}$*.*

For our first example let us consider three possible queries X_1, X_2 and X_3. For each query we can partition a set of diagnoses $\mathbf{D}$ into three sets $\mathbf{D^X}$, $\mathbf{D^{\neg X}}$ and $\mathbf{D^{\emptyset}}$ (see Table1). Using this data and the heuristic given above we can determine that asking the oracle if $\mathcal{O}_t \models C(w)$? is the best query, as two diagnoses from the set $\mathbf{D}$ are removed regardless of the answer.

Let us assume that $\mathcal{D}_1$ is the target diagnosis, then an oracle will answer 0 to our question (i.e. $\mathcal{O}_t \not\models C(w)$). Given this feedback we can decide that $\mathcal{O}_t \models B(w)$? is the next best query, which is also answered with 0 by the oracle. Consequently, we identified that $\mathcal{D}_1$ is the only remaining minimal diagnosis. More generally, if n is the number of diagnoses and we can split the set of diagnoses in half by each query then the minimum number of queries is $log_2 n$. However, if the probabilities of diagnoses are known we can reduce this number of queries by using two effects: (1) We can exploit diagnoses probabilities to asses the probabilities of answers and the change in information content after an answer is given. (2) Even if there are multiple diagnoses in the set of remaining diagnoses we can stop further query generation if one diagnosis is highly probable and all other remaining diagnoses are highly improbable.

Example 2 Consider an ontology $\mathcal{O}$ with the terminology $\mathcal{T}$:

$$\begin{array}{ll} ax_1 : A_1 \sqsubseteq A_2 \sqcap M_1 \sqcap M_2 & ax_4 : M_2 \sqsubseteq \forall s.A \sqcap C \\ ax_2 : A_2 \sqsubseteq \neg \exists s.M_3 \sqcap \exists s.M_2 & ax_5 : M_3 \equiv B \sqcup C \\ ax_3 : M_1 \sqsubseteq \neg A \sqcap B & \end{array}$$

and the background theory $\mathcal{A} : \{A_1(w), A_1(u), s(u, w)\}$. The ontology is inconsistent and includes two minimal conflicts: $\{\langle ax_1, ax_3, ax_4 \rangle, \langle ax_1, ax_2, ax_3, ax_5 \rangle\}$. To restore consistency, the user should modify all axioms of at least one minimal diagnosis:

$$\mathcal{D}_1 : [ax_1] \quad \mathcal{D}_2 : [ax_3] \quad \mathcal{D}_3 : [ax_4, ax_5] \quad \mathcal{D}_4 : [ax_4, ax_2]$$

Following the same approach as in the first example, we compute entailments for each minimal diagnosis $\mathcal{D}_i \in \mathbf{D}$. To construct a query we select a $\mathbf{D^X} \subset \mathbf{D}$ and determine the set X

Table 2. Possible queries in Example 2

Query	$\mathbf{D^X}$	$\mathbf{D^{\neg X}}$	$\mathbf{D^\emptyset}$
$X_1 : \{B \sqsubseteq M_3\}$	$\{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4\}$	$\{\mathcal{D}_3\}$	$\emptyset$
$X_2 : \{B(w)\}$	$\{\mathcal{D}_3, \mathcal{D}_4\}$	$\{\mathcal{D}_2\}$	$\{\mathcal{D}_1\}$
$X_3 : \{M_1 \sqsubseteq B\}$	$\{\mathcal{D}_1, \mathcal{D}_3, \mathcal{D}_4\}$	$\{\mathcal{D}_2\}$	$\emptyset$
$X_4 : \{M_1(w), M_2(u)\}$	$\{\mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4\}$	$\{\mathcal{D}_1\}$	$\emptyset$
$X_5 : \{A(w)\}$	$\{\mathcal{D}_2\}$	$\{\mathcal{D}_3, \mathcal{D}_4\}$	$\{\mathcal{D}_1\}$
$X_6 : \{M_2 \sqsubseteq D\}$	$\{\mathcal{D}_1, \mathcal{D}_2\}$	$\emptyset$	$\{\mathcal{D}_3, \mathcal{D}_4\}$
$X_7 : \{M_3(u)\}$	$\{\mathcal{D}_4\}$	$\emptyset$	$\{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3\}$

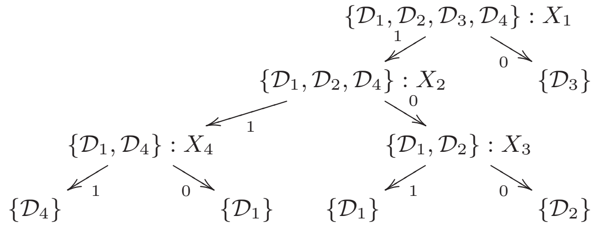


Figure 1. Greedy algorithm

of common entailed concept instantiations and concept subsumption axioms for all $\mathcal{D}_i \in \mathbf{D^X}$. If the set X is empty, the query is rejected. For instance, diagnoses $\mathcal{D}_2$, $\mathcal{D}_3$ and $\mathcal{D}_4$ have the following set of common entailments $X'_4 : \{A_1 \sqsubseteq A_2, A_1 \sqsubseteq M_1, A_1 \sqsubseteq M_2, A_2(u), M_1(u), M_2(u), A_2(w), M_1(w)\}$. However, a query need not include all of these axioms. Note, a query X partitions the set of diagnoses into $\mathbf{D^X}$, $\mathbf{D^{\neg X}}$ and $\mathbf{D^\emptyset}$. It is sufficient to query an irreducible subset of X which preserves the partition. In our example, the set X'_4 can be reduced to its subset $X_4 : \{M_1(w), M_2(u)\}$. If there are multiple subsets that preserve the partition we select one with minimal cardinality. For query generation we investigate all possible subsets of $\mathbf{D}$. This is feasible since we consider only the n most probable minimal diagnoses (e.g. $n = 12$) during query generation and selection.

The possible queries presented in Table 2 partition the set of diagnoses $\mathbf{D}$ in a way that makes the application of myopic strategies, such as split-in-half, inefficient. A greedy algorithm based on such a heuristic would select the first query X_1 as the next query, since there is no query that cuts the set of diagnoses in half. If $\mathcal{D}_4$ is the target diagnosis then X_1 will be positively evaluated by an oracle (see Figure 1). On the next iteration the algorithm would also choose a suboptimal query since there is no partition that divides the diagnoses $\mathcal{D}_1, \mathcal{D}_2$, and $\mathcal{D}_4$ into two equal groups. Consequently, it selects the first untried query X_2. The oracle answers positively, and the algorithm identifies query X_4 to differentiate between $\mathcal{D}_1$ and $\mathcal{D}_4$.

However, in real-world settings the assumption that all axioms fail with the same probability is rarely the case. For example, Rector at al. [9] report that in most cases inconsistent ontologies were created because users (a) mix up $\forall r.S$ and $\exists r.S$, (b) mix up $\neg\exists r.S$ and $\exists r.\neg S$, (c) mix up $\sqcup$ and $\sqcap$, (d) wrongly assume that classes are disjoint by default, (e) wrongly apply negation. Using this information one might find that axioms ax_2 and ax_4 are significantly more likely to be faulty than ax_3 (because of the use of quantifiers), whereas ax_3 is significantly more likely to be faulty than ax_5 and ax_1 (because of the use of negation). Therefore, diagnosis $\mathcal{D}_2$ is the most probable one, followed closely by $\mathcal{D}_4$ although it is a double fault diagnosis. $\mathcal{D}_1$ and $\mathcal{D}_3$ are significantly less probable because ax_1 and ax_5 have a significantly lower fault probability than ax_3. A detailed justification based on probability is given in the next section.

Taking into account the information about user faults given above, it is almost useless to ask query X_1 because it is highly probable that the target diagnosis is either $\mathcal{D}_2$ or $\mathcal{D}_4$ and therefore it is highly probable that the oracle will respond with 1. Instead, asking X_3 is more informative because given any possible answer we can exclude one of the highly probable diagnoses, i.e. either $\mathcal{D}_2$ or $\mathcal{D}_4$. If the oracle responds to X_3 with 0 then $\mathcal{D}_2$ is the only remaining diagnosis. However, if the oracle responds with 1, diagnoses $\mathcal{D}_4$, $\mathcal{D}_3$, and $\mathcal{D}_1$ remain, where $\mathcal{D}_4$ is significantly more probable compared to diagnoses $\mathcal{D}_3$ and $\mathcal{D}_1$. We can stop, since the difference between the probabilities of the diagnoses is high enough such that $\mathcal{D}_1$ can be accepted as the target diagnosis. In other situations additional questions may be required. This strategy can lead to a substantial reduction in the number of queries compared to myopic approaches as we will show in our evaluation.

3 Entropy-based query selection

To select the best query we make the assumption that knowledge is available about the a-priori failure probabilities in specifying axioms. Such probabilities can be estimated either by studies like Rector et al. [9] or can be personalized by observing the typical failures of specific users working with an ontology development tool. Using observations about typical failures we can calculate the initial probability of each axiom $p(ax_i)$ containing a failure using the probability addition rule for non-mutually exclusive events. If no information about failures is available then the debugger can initialize all probabilities $p(ax_i)$ with some small number.

Given the failure probabilities $p(ax_i)$ of axioms, the diagnosis algorithm first calculates the a-priori probability $p(\mathcal{D}_j)$ that $\mathcal{D}_j$ is the target diagnosis. Since all axioms fail independently, this probability can be computed as [2]:

$$p(\mathcal{D}_j) = \prod_{ax_n \not\in \mathcal{D}_j} p(ax_n) \prod_{ax_m \in \mathcal{D}_j} 1 - p(ax_m) \qquad (1)$$

The prior probabilities for diagnoses are then used to initialize an iterative algorithm that includes two main steps: (a) selection of the best query and (b) update of the diagnoses probabilities given the query feedback.

According to information theory the best query is the one that, given the answer of an oracle, minimizes the expected entropy of a the set of diagnoses [2]. Let $p(X_i = v_{ik})$ where $v_{i0} = 0$ and $v_{i1} = 1$ be the probability that query X_i is answered with either 0 or 1. Let $p(\mathcal{D}_j | X_i = v_{ik})$ be the probability of diagnosis $\mathcal{D}_j$ after the oracle answers $X_i = v_{ik}$. The expected entropy after querying X_i is:

$$\begin{aligned} H_e(X_i) = &\sum_{k=0}^{1} p(X_i = v_{ik}) \times \\ &- \sum_{\mathcal{D}_j \in \mathbf{D}} p(\mathcal{D}_j | X_i = v_{ik}) \log_2 p(\mathcal{D}_j | X_i = v_{ik}) \end{aligned}$$

The query which minimizes the expected entropy is the best one based on a one-step-look-ahead information theoretic measure. This formula can be simplified to the following score function [2] which we use to evaluate all available queries and select the one with the minimum score to maximize information gain:

$$sc(X_i) = \sum_{k=0}^{1} p(X_i = v_{ik}) \log_2 p(X_i = v_{ik}) + p(\mathbf{D_i^\emptyset}) + 1 \qquad (2)$$

where $\mathbf{D_i^\emptyset}$ is the set of diagnoses which do not make any predictions for X_i. Since, for a query X_i the set of diagnoses $\mathbf{D}$ can be partitioned into the sets $\mathbf{D^{X_i}}$, $\mathbf{D^{\neg X_i}}$ and $\mathbf{D_i^\emptyset}$, the probability that an oracle will answer a query X_i with either 1 or 0 can be computed as:

$$p(X_i = v_{ik}) = p(\mathbf{S_{ik}}) + p(\mathbf{D_i^\emptyset})/2 \qquad (3)$$

where $\mathbf{S_{ik}}$ corresponds to the set of diagnoses that predicts the outcome of a query, e.g. $\mathbf{S_{i0}} = \mathbf{D^{\neg X_i}}$ for $X_i = 0$ and $\mathbf{S_{i1}} = \mathbf{D^{X_i}}$ in the other case. $p(\mathbf{D_i^\emptyset})$ is the total probability of the diagnoses that predict no value for the query X_i. Under the assumption that *both outcomes are equally likely* the probability that a set of diagnoses $\mathbf{D_i^\emptyset}$ predicts $X_i = v_{ik}$ is $p(\mathbf{D_i^\emptyset})/2$.

Since all diagnoses are statistically independent the probabilities of their sets can be calculated as:

$$p(\mathbf{D_i^\emptyset}) = \sum_{\mathcal{D}_j \in \mathbf{D_i^\emptyset}} p(\mathcal{D}_j) \qquad p(\mathbf{S_{ik}}) = \sum_{\mathcal{D}_j \in \mathbf{S_{ik}}} p(\mathcal{D}_j)$$

Given the feedback v of an oracle to the selected query X_s, i.e. $X_s = v$ we have to update the probabilities of the diagnoses to take the new information into account. The update is made using Bayes' rule for each $\mathcal{D}_j \in \mathbf{D}$:

$$p(\mathcal{D}_j | X_s = v) = \frac{p(X_s = v | \mathcal{D}_j) p(\mathcal{D}_j)}{p(X_s = v)} \tag{4}$$

where the denominator $p(X_s = v)$ is known from the query selection step (Equation 3) and $p(\mathcal{D}_j)$ is either a prior probability (Equation 1) or is a probability calculated using Equation 4 during the previous iteration of the debugging algorithm. We assign $p(X_s = v|\mathcal{D}_j)$ as follows:

$$p(X_s = v|\mathcal{D}_j) = \begin{cases} 1, & \text{if } \mathcal{D}_j \text{ predicted } X_s = v; \\ 0, & \text{if } \mathcal{D}_j \text{ is rejected by } X_s = v; \\ \frac{1}{2}, & \text{if } \mathcal{D}_j \in \mathbf{D_s^\emptyset} \end{cases}$$

Example 1 (continued) Suppose that the debugger is not provided with any information about possible failures and therefore it assumes that all axioms fail with the same probability $p(ax_i) = 0.01$. Using Equation 1 we can calculate probabilities for each diagnosis. For instance, $\mathcal{D}_1$ suggests that only one axiom ax_1 should be modified by the user. Hence, the probability of diagnosis D_1, $p(\mathcal{D}_1) = p(ax_1)(1 - p(ax_2))(1 - p(ax_3))(1 - p(ax_4)) = 0.0097$. All other minimal diagnoses have the same probability, since every other minimal diagnosis suggests the modification of one axiom. To simplify the discussion we only consider minimal diagnoses for the query selection. Therefore, the prior probabilities of the diagnoses can be normalized to $p(\mathcal{D}_j) = p(\mathcal{D}_j)/\sum_{\mathcal{D}_j \in \mathbf{D}} p(\mathcal{D}_j)$ and are equal to 0.25.

Given the prior probabilities of the diagnoses and a set of queries (see Table 1) we evaluate the score function (Equation 2) for each query. E.g. for the first query $X_1 : \{B(w)\}$ the probability $p(\mathbf{D^\emptyset}) = 0$ and the probabilities of both the positive and negative outcomes are: $p(X_1 = 1) = p(\mathcal{D}_2) + p(\mathcal{D}_3) + p(\mathcal{D}_4) = 0.75$ and $p(X_1 = 0) = p(\mathcal{D}_1) = 0.25$. Therefore the query score is $sc(X_1) = 0.1887$.

The scores computed during the initial stage (see Table 3) suggest that X_2 is the best query. Taking into account that $\mathcal{D}_1$ is the target diagnosis the oracle answers 0 to the query. The additional information obtained from the answer is then used to update the probabilities of diagnoses using the Equation 4. Since $\mathcal{D}_1$ and $\mathcal{D}_2$ predicted this answer, their probabilities are updated, $p(\mathcal{D}_1) = p(\mathcal{D}_2) = 1/p(X_2 = 1) = 0.5$. The probabilities of diagnoses $\mathcal{D}_3$ and $\mathcal{D}_4$ which are rejected by the outcome are also updated, $p(\mathcal{D}_3) = p(\mathcal{D}_4) = 0$.

Table 3. Expected scores for queries ($p(ax_i) = 0.01$)

Query	Initial	$(X_2 = 1)$
$X_1 : \{B(w)\}$	0.1887	**0**
$X_2 : \{C(w)\}$	**0**	1
$X_3 : \{Q(w)\}$	0.1887	1

On the next iteration the algorithm recomputes the scores using the updated probabilities. The results show that X_1 is the best query. The other two queries X_2 and X_3 are irrelevant since no information will be gained if they are performed. Given the negative feedback of an oracle to X_1, we update the probabilities $p(\mathcal{D}_1) = 1$ and $p(\mathcal{D}_2) = 0$. In this case the target diagnosis $\mathcal{D}_1$ was identified using the same number of steps as the split-in-half heuristic.

Table 4. Expected scores for queries ($p(ax_1) = 0.025$, $p(ax_2) = p(ax_3) = p(ax_4) = 0.01$)

Query	Initial score
$X_1 : \{B(w)\}$	**0.250**
$X_2 : \{C(w)\}$	0.408
$X_3 : \{Q(w)\}$	0.629

However, if the first axiom is more likely to fail, e.g. $p(ax_1) = 0.025$, then the first query will be $X_1 : \{B(w)\}$ (see Table 4). The recalculation of the probabilities given the negative outcome $X_1 = 0$ sets $p(\mathcal{D}_1) = 1$ and $p(\mathcal{D}_2) = p(\mathcal{D}_3) = p(\mathcal{D}_4) = 0$. Therefore the debugger identifies the target diagnosis only in one step.

Example 2 (continued) Suppose that in ax_4 the user specified $\forall s.A$ instead of $\exists s.A$ and $\neg\exists s.M_3$ instead of $\exists s.\neg M_3$ in ax_2. Therefore $\mathcal{D}_4$ is the target diagnosis. Moreover, the debugger is provided with observations of three types of failures: (1) conjunction/disjunction occurs with probability $p_1 = 0.001$, (2) negation $p_2 = 0.01$, and (3) restrictions $p_3 = 0.05$. Using the probability addition rule for non-mutually exclusive events we can calculate the probability of the axioms containing an error: $p(ax_1) = 0.0019$, $p(ax_2) = 0.1074$, $p(ax_3) = 0.012$, $p(ax_4) = 0.051$, and $p(ax_5) = 0.001$. These probabilities are exploited to calculate the prior probabilities of the diagnoses (see Table 5) and to initialize the query selection process.

On the first iteration the algorithm determines that X_3 is the best query and asks an oracle whether $\mathcal{O}_t \models M_1 \sqsubseteq B$ is true or not (see Table 6). The obtained information is then used to recalculate the probabilities of the diagnoses and to compute the next best query X_4, and so on. The query process stops after the third query, since $\mathcal{D}_4$ is the only diagnosis that has the probability $p(\mathcal{D}_4) > 0$.

Given the feedback of the oracle $X_4 = 1$ for the second query, the updated probabilities of the diagnoses show that the target diagnosis has a probability of $p(\mathcal{D}_4) = 0.9918$ whereas $p(\mathcal{D}_3)$ is only 0.0082. In order to reduce the number of queries a user can specify a threshold, e.g. $\sigma = 0.95$. If the probability of some diagnosis is greater than this threshold, the query process stops and returns the most probable diagnosis. Note, that even after the first answer $X_3 = 1$ the most probable diagnosis $\mathcal{D}_3$ is three times more likely than the second most probable diagnosis D_1. Given such a great difference we could suggest to stop the query process after the first answer. Thus, in this example the debugger requires less queries than the split-in-half heuristic.

Table 5. Probabilities of diagnoses after answers

Answers	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
Prior	0.0970	0.5874	0.0026	0.3130
$X_3 = 1$	0.2352	0	0.0063	0.7585
$X_3 = 1, X_4 = 1$	0	0	0.0082	0.9918
$X_3 = 1, X_4 = 1, X_1 = 1$	0	0	0	1

Table 6. Expected scores for queries

Queries	Initial	$X_3 = 1$	$X_3 = 1, X_4 = 1$
$X_1 : \{B \sqsubseteq M_3\}$	0.974	0.945	**0.931**
$X_2 : \{B(w)\}$	0.151	0.713	1
$X_3 : \{M_1 \sqsubseteq B\}$	**0.022**	1	1
$X_4 : \{M_1(w), M_2(u)\}$	0.540	**0.213**	1
$X_5 : \{A(w)\}$	0.151	0.713	1
$X_6 : \{M_2 \sqsubseteq D\}$	0.686	0.805	1
$X_7 : \{M_3(u)\}$	0.759	0.710	0.970

4 Implementation details

The ontology debugger (Algorithm 1) takes an ontology $\mathcal{O}$ as input. Optionally, a user can provide a set of axioms B that are known to be correct, a set FP of fault probabilities for axioms $ax_i \in \mathcal{O}$, a maximum number n of most probable minimal diagnoses that should be considered by the algorithm, and a diagnosis acceptance threshold σ. The fault probabilities of axioms are computed as described by exploiting knowledge about typical user errors. Parameters n and σ are used to speed up the computations. In Algorithm 1 we approximate the set of the n most probable diagnoses with the set of the n most probable *minimal* diagnoses, i.e. we neglect non-minimal diagnoses which are more probable than some minimal ones. This approximation is correct, under a reasonable assumption that probability of each axiom $p(ax_i) < 0.5$. In this case for every non-minimal diagnosis ND, a minimal diagnosis $\mathcal{D} \subset ND$ exists which from Equation 1 is more probable than ND. Consequently the query selection algorithm operates on the set of minimal diagnoses instead of all diagnoses (including non-minimal ones). However, the algorithm can be adapted with moderate effort to also consider non-minimal diagnoses.

We implemented the computation of diagnoses following the approach proposed by Friedrich et al. [3]. The authors employ the combination of two algorithms, QUICKXPLAIN [5] and HS-TREE [10]. The latter is a search algorithm that takes an ontology $\mathcal{O}$, a set of correct axioms, a set of axioms $T^{\not\models}$ which must not be implied by the target ontology, and the maximal number of most probable minimal diagnoses n as an input. HS-TREE implements a breadth-first search strategy to compute a set of minimal hitting sets from the set of all minimal conflicts in $\mathcal{O}$. As suggested in [3] it ignores all branches of the search tree that correspond to hitting sets inconsistent with at least one element of $T^{\not\models}$. HS-TREE terminates if either it identifies the n most probable minimal diagnoses or there are no further diagnoses which are more probable than the already computed ones. Note, HS-TREE often calculates only a small number of minimal conflict sets in order to generate the n most probable minimal hitting sets (i.e. minimal diagnoses), since only a subset of all minimal diagnoses is required.

The search algorithm computes minimal conflicts using QUICKXPLAIN. This algorithm, given a set of axioms AX and a set of correct axioms B returns a minimal conflict set $CS \subseteq AX$, or $\emptyset$ if axioms $AX \cup B$ are consistent. Minimal conflicts are computed on-demand by HS-TREE while exploring the search space.

The set of minimal hitting sets returned by HS-TREE is used by GETDIAGNOSES to create at most n minimal diagnoses $\mathbf{D}$. The function generates diagnoses $\mathcal{D}_i \in \mathbf{D}$ by removing all elements of the corresponding minimal hitting set from $\mathcal{O}$.

The COMPUTEDATASET function uses the set of diagnoses to generate data sets like the ones presented in Tables 1 and 2. For each diagnosis $\mathcal{D}_i \in \mathbf{D}$ the algorithm gets a set of entailments from the reasoner and computes the set of queries. For each query X_i it partitions the set $\mathbf{D}$ into $\mathbf{D^{X_i}}$, $\mathbf{D^{\neg X_i}}$ and $\mathbf{D_i^{\emptyset}}$, as defined in Section 2. Then X_i is iteratively reduced by applying QUICKXPLAIN such that sets $\mathbf{D^{X_i}}$ and $\mathbf{D^{\neg X_i}}$ are preserved.

Algorithm 1: Ontology debugging algorithm

```
Input: ontology $\mathcal{O}$, set of background axioms $B$, set of fault
       probabilities for axioms $FP$, maximum number of most
       probable minimal diagnoses $n$, acceptance threshold $\sigma$
Output: a diagnosis $\mathcal{D}$
$DP \leftarrow \emptyset$; $DS \leftarrow \emptyset$; $T^{\models} \leftarrow \emptyset$; $T^{\not\models} \leftarrow \emptyset$; $\mathbf{D} \leftarrow \emptyset$;
while $true$ do
    $\mathbf{D} \leftarrow$ getDiagnoses(HS-Tree($\mathcal{O}, B \cup T^{\models}, T^{\not\models}, n$));
    $DS \leftarrow$ computeDataSet($DS, \mathbf{D}$);
    $DP \leftarrow$ computePriors($\mathbf{D}, FP$);
    $DP \leftarrow$ uptateProbablities($DP, DS, T^{\models}, T^{\not\models}$);
    $s \leftarrow$ getMinimalScore($DS, DP$);
    if aboveThreshold($DP, \sigma$) $\vee\ s = 1$ then
        return mostProbableDiagnosis($\mathbf{D}, DP$);
    $\langle X, \mathbf{D^X}, \mathbf{D^{\neg X}} \rangle \leftarrow$ selectQuery($DS, s$);
    if getAnswer($\mathcal{O}_t \models X$) then $T^{\models} \leftarrow T^{\models} \cup X$;
    else $T^{\not\models} \leftarrow T^{\not\models} \cup \neg X$;
```

In the next step COMPUTEPRIORS computes prior probabilities for a set of diagnoses given the fault probabilities of the axioms contained in FP. To take past answers into account the algorithm updates the prior probabilities of the diagnoses by evaluating Equation 4 for each diagnosis in $\mathbf{D}$ (UPDATEPROBABILITIES). All data required for the update is stored in sets DS, $T^{\models}$, and $T^{\not\models}$.

The function GETMINIMALSCORE evaluates the scoring function (Equation 2) for each element of DS and returns the minimal score.

The algorithm stops if there is a diagnosis probability above the acceptance threshold σ or if no query can be used to differentiate between the remaining diagnoses (i.e. all scores are 1). The most probable diagnosis is then returned to the user. If it is impossible to differentiate between a number of highly probable minimal diagnoses, the algorithm returns a set that includes all of them.

In the query-selection phase the algorithm selects a set of axioms that should be evaluated by an oracle. SELECTQUERY retrieves a triple $\langle X, \mathbf{D^X}, \mathbf{D^{\neg X}} \rangle \in DS$ that corresponds to the best (minimal) score s. The set of axioms X is then presented to the oracle. If there are multiple queries with a minimal score SELECTQUERY returns the triple where X has the smallest cardinality in order to reduce the answering effort.

Depending on the answer of the oracle, the algorithm extends either set $T^{\models}$ or $T^{\not\models}$. This is done to exclude corresponding diagnoses from the results of HS-TREE in further iterations. Note, the algorithm can be easily extended to allow the oracle to reject a query if the answer is unknown. In this case the algorithm proceeds with the next best query until no further queries are available.

5 Evaluation

The evaluation of our approach was performed using generated examples. We employed generated examples because (1) for published sets of inconsistent/incoherent ontologies, such as those described in [6], the target ontologies are not known and (2) we wanted to perform controlled experiments where the number of minimal diagnoses and their cardinality could be varied to make the identification of the target diagnosis more difficult.

Therefore we created a generator which takes a consistent and coherent ontology, a set of fault patterns together with their probabilities, the required number of minimal diagnoses, and the required minimum cardinality of these minimal diagnoses as inputs. The output was an alteration of the input ontology for which the required number of minimal diagnoses with the required cardinality exist. In order to introduce inconsistencies and incoherences, the generator applied fault patterns randomly to the input ontology depending on their probabilities.

In our experiments we took five fault patterns from a case study reported by Rector at al. [9] and assigned fault probabilities according to their observations of typical user errors. Thus we assumed that in cases (a) and (b) (see Section 2, when an axiom includes some roles (i.e. property assertions), axiom descriptions are faulty with a probability of 0.025, in cases (c) and (d) 0.01 and in case (e) 0.001. In each iteration the generator randomly selected an axiom to be altered and applied a fault pattern to this axiom. Next it selected another axiom using the concept taxonomy and altered it correspondingly to introduce an incoherency/inconsistency. The fault patterns were randomly selected in each step using the probabilities given above.

For instance, given the description of a randomly selected concept

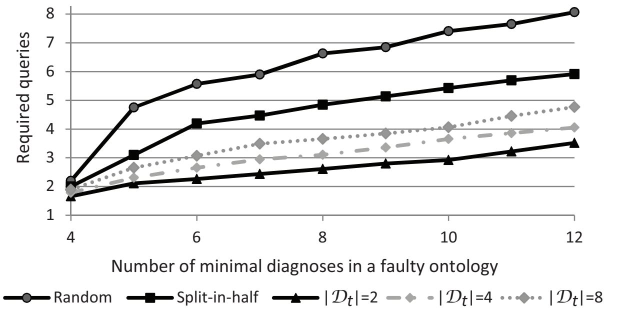


Figure 2. Number of queries required to select the target diagnosis $\mathcal{D}_t$ with threshold $\sigma = 0.95$. Random and "split-in-half" are shown for $|\mathcal{D}_t| = 2$.

A and the fault pattern "misuse of negation", we added the construct $\sqcap\neg X$ to the description of A, where X is a new concept name. Next, we randomly selected concepts B and S such that $S \sqsubseteq A$ and $S \sqsubseteq B$ and added $\sqcap X$ to the description of B.

During the generation process, we applied the HS-TREE algorithm after each introduction of a incoherency/inconsistency to control two parameters: the number of minimal diagnoses in the ontology and their minimum cardinality. The generator continued to introduce incoherences/inconsistencies until the specified parameters were reached. The resulting faulty ontology as well as the fault patterns and their probabilities were inputs for the ontology debugger. The acceptance threshold σ was set to 0.95 and the number of most probable minimal diagnoses n was set to 12. One of the minimal diagnoses with the required cardinality was randomly selected as the target diagnosis. Note, the target ontology is not equal to the original ontology, but rather is a corrected version of the altered one, in which the faulty axioms were repaired by replacing them with their original (correct) versions according to the target diagnosis. The tests were done on ontologies bike2 to bike9, bcs3, galen and galen2 from Racer's benchmark suite[2].

The average results of the evaluation performed on each test suite (depicted in Figure 2) show that the entropy-based approach outperforms the split-in-half method described in Section 2 as well as random query selection by more than 50% for the $|\mathcal{D}_t| = 2$ case due to its ability to estimate the probabilities of diagnoses. On average the algorithm required 8 seconds to generate a query. Figure 2 also shows that the cardinality of the target diagnosis increases as the number of required queries increases. This holds for the random and split-in-half methods (not depicted) as well. However, the entropy-based approach is still better than the split-in-half method even for diagnoses with increasing cardinality. The approach required more queries to discriminate between high cardinality diagnoses because the prior probabilities of these diagnoses tend to converge.

6 Related work

To the best of our knowledge, no entropy-based methods for query generation and selection have been proposed to debug faulty ontologies so far. Diagnosis methods for ontologies are introduced in [11, 6, 3]. Ranking of diagnoses and proposing a target diagnosis is presented in [7]. This method uses a number of measures such as: (a) the frequency with which an axiom appears in conflict sets, (b) impact on an ontology in terms of its "lost" entailments when some axiom is modified or removed, (c) ranking of test cases, (d) provenance information about the axiom, and (e) syntactic relevance. All these measures are evaluated for each axiom in a conflict set. The scores are then combined in a rank value which is associated with the corresponding axiom. These ranks are then used by a modified HS-TREE algorithm that identifies diagnoses with a minimal rank. In this work no query generation and selection strategy is proposed if the target diagnosis cannot be determined reliably with the given a-priori knowledge. In our work additional information is acquired until the target diagnosis can be identified with confidence. In general, the work of [7] can be combined with the one presented in this paper as axiom ranks can be taken into account together with other observations while calculating the prior probabilities of the diagnoses.

The idea of selecting the next best query based on the expected entropy was exploited in the generation of decisions trees [8] and further refined for selecting measurements in the model-based diagnosis of circuits [2]. We extended these methods to query selection in the domain of ontology debugging.

7 Conclusions

In this paper we presented an approach to the sequential diagnosis of ontologies. We showed that the axioms generated by classification and realization can be used to build queries which differentiate between diagnoses. To rank the utility of these queries we employ knowledge about typical user errors in ontology axioms. Based on the likelihood of an ontology axiom containing an error we predict the information gain produced by a query result, enabling us to select the next best query according to a one-step-lookahead entropy-based scoring function. We outlined the implementation of a sequential debugging algorithm and compared our proposed method with a split-in-half strategy. Our experiments showed a significant reduction in the number of queries required to identify the target diagnosis.

REFERENCES

[1] *The Description Logic Handbook*, eds., F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, Cambridge University Press, New York, NY, USA, 2nd edn., 2007.

[2] J. de Kleer and B.C. Williams, 'Diagnosing multiple faults', *Artificial Intelligence*, **32**(1), 97–130, (April 1987).

[3] G. Friedrich and K. Shchekotykhin, 'A General Diagnosis Method for Ontologies', in *Proceedings of the 4th International Semantic Web Conference*, pp. 232–246. Springer, (2005).

[4] V. Haarslev and R. Müller, 'RACER System Description', in *Proceedings of the 1st International Joint Conference on Automated Reasoning*, pp. 701–705, Springer, (2001).

[5] U. Junker, 'QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems.', in *Association for the Advancement of Artificial Intelligence*, pp. 167–172, AAAI, (2004).

[6] A. Kalyanpur, B. Parsia, M. Horridge, and E. Sirin, 'Finding all Justifications of OWL DL Entailments', in *Proceedings of the 6th International Semantic Web Conference*, pp. 267–280, Springer, (2007).

[7] A. Kalyanpur, B. Parsia, E. Sirin, and B. Cuenca-Grau, 'Repairing Unsatisfiable Concepts in OWL Ontologies', in *Proceedings of the 3rd European Semantic Web Conference*, pp. 170–184, Springer, (2006).

[8] J.R. Quinlan, 'Induction of Decision Trees', *Machine Learning*, **1**(1), 81–106, (March 1986).

[9] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe, 'OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns', in *Proceedings of 14th International Conference on Knowledge Engineering and Knowledge Management*, pp. 63–81, Springer, (2004).

[10] R. Reiter, 'A Theory of Diagnosis from First Principles', *Artificial Intelligence*, **23**, 57–95, (1987).

[11] S. Schlobach, Z. Huang, R. Cornet, and F. Harmelen, 'Debugging Incoherent Terminologies', *Journal of Automated Reasoning*, **39**(3), 317–349, (2007).

[12] E. Sirin, B. Parsia, B. Cuenca-Grau, A. Kalyanpur, and Y. Katz, 'Pellet: A practical OWL-DL reasoner', *Web Semantics: Science, Services and Agents on the World Wide Web*, **5**(2), 51–53, (2007).

[2] http://www.racer-systems.com/products/download/benchmark.phtml

On the Way to High-Level Control for Resource-Limited Embedded Systems with Golog

Alexander Ferrein[1] and **Gerald Steinbauer**[2]

Abstract.

In order to allow an autonomous robot to perform non-trivial tasks like to explore a foreign planet the robot has to have deliberative capabilities like reasoning or planning. Logic-based approaches like the programming and planing language Golog and it successors has been successfully used for such decision-making problems. A drawback of this particular programing language is that their interpreter usually are written in Prolog and run on a Prolog back-end. Such back-ends are usually not available or feasible on resource-limited robot systems. In this paper we present our ideas and first results of a re-implementation of the interpreter based on the Lua scripting language which is available on a wide range of systems including small embedded systems.

1 Introduction

In order to allow an autonomous robot to perform non-trivial tasks like to explore a foreign planet the robot has to have deliberative capabilities like reasoning or planning. There are a number of architectures in the literature like the Saphira architecture [14] or Structured Reactive Controllers [1] which enhance a robot with such advanced decision making capabilities. Other approaches, which have been investigated over the last decade, are logic-based approaches. They have been successfully applied to the control of various robot systems. One representative of this sort of decision-making approaches is the programming and planing language Golog [16]. The language is based on the Situation Calculus which allows for reasoning about actions and change. Moreover, its syntax and semantics provides an elegant way to specify the behavior of a system. During the years several dialects of Golog have been developed to integrate more features like dealing with uncertain knowledge, concurrency and sensing [19]. The different Golog derivatives have in common that they are logic-based and their implementation is, so far, usually done in the logic programming language Prolog. The used Prolog back-ends like *SWI* or *Eclipse* usually have the drawback of high demands on memory and computational power. This fact and the fact that probably no Prolog system might be available at all on the target platform, prevents the application of Golog on resource-limited embedded systems. With resource-limited embedded systems we mean systems which have strongly limited resources in terms of memory and computational power. On such systems classical decision making is not able to provide decisions in time, fails to come up with decisions or the methods used are not implementable at all.

The goals of our current ongoing research is to apply such powerful decision-making approaches also to resource-limited systems. In particular we have two target robot systems in mind which we use for our research on autonomous robots. On one hand we use the humanoid robot *Nao* from Aldebaran Robotics. We use the robot for our participation in the competitions of the RoboCup Standard League [7]. Nao is a 21 degrees-of-freedom humanoid robot of the size of about 58 cm. It is equipped with two cameras and a AMD Geode 500 MHz CPU with 256 MB of Memory. The robot is running under embedded Linux where no Prolog interpreter is available nor feasible. The other platform is the Lego Mindstorm NXT system, which we also use for educational robotics on high-school and undergraduate level. The system is also very popular for robot competitions like RoboCupJunior or undergraduate classes in programming. The system has predefined interfaces to various sensors and actuator. The central processing block uses an ARM processor with 64 kB of RAM and 256 kB of flash-memory running at 48 MHz. There are a number of commercial and open source OS for the system. But none of them are coming with a logic back-end. The NXT robot system has also been successfully used to practically teach AI methods from standard textbooks [20, 21]. But the AI methods did not run native on the robot. The related code was executed on an external computer and commands were send to the robot via blue-tooth interface.

In order to make Golog available on such platforms, we started to develop a first prototype implementation of a Golog interpreter in the scripting language Lua [5]. Lua [12] is a fast interpreted scripting language with a small memory footprint that moreover can be executed on resource-limited systems like the Nao robot. In the previous work we ran several standard Golog examples on our Lua interpreter as a proof of concept. In this paper, we discuss our plans to deploy IndiGolog on the Lego Mindstorm NXT platform. IndiGolog is an extension of Golog and allows on-line interpretation sensing and concurrent actions which was not the case in vanilla Golog. A previous attempt to control a Lego robot with IndiGolog was done by Levesque and Pagnucco in [15]. The developed system allowed users to specify the behavior with IndiGolog, the resulting actions were transmitted to the robot via an infra-red link. Recently, the action logic C+ was deployed on a Mindstorm robot in a similar way [3].

In order to be able to use Golog-type languages to control autonomous robots and to directly execute it on the resource-limited system without an external Prolog back-end, we propose to port our previous idea to the two target systems. In the following sections we will explain our idea about the porting the language and the background in more details. In Section 2, we introduce Golog and our previous work on an Golog interpreter in Lua, before we show implementation details and further ideas to deploy the new interpreter on the NXT platform in Section 3. We conclude with Section 4.

[1] Robotics and Agent Research Lab, University of Cape Town, Cape Town, South Africa

[2] Institute for Software Technology, Graz University of Technology, Graz, Austria

(a) Humanoid robot Nao used in RoboCup Standard Platform League

(b) Lego Mindstorm NXT used for educational robotics.

Figure 1. Two of our target platforms.

2 Previous Work on Golog and Lua

In [5], we presented an implementation of the so-called vanilla Golog, this time the chosen implementation language was not Prolog, but Lua. Although only a proof of concept, it showed an alternative implementation for Golog. Especially the implementation language Lua, which was chosen in [5], is interesting, as it is available for a much larger number of systems, including embedded architectures like the Mindstorm NXT. In this section, we briefly go over the basics of the situation calculus and Golog, give a short introduction to Lua, and sketch our previous Lua implementation of Golog.

2.1 Golog

Golog is based on Reiter's variant of the Situation Calculus [17, 19], a second-order language for reasoning about actions and their effects. Changes in the world are only caused by actions so that a situation is completely described by the history of actions starting in some initial situation. Properties of the world are described by fluents, which are situation-dependent predicates and functions. For each fluent the user defines a successor state axiom specifying precisely which value the fluent takes on after performing an action. These, together with precondition axioms for each action, axioms for the initial situation, foundational and unique names axioms, form a so-called basic action theory [19]. Golog emerged to an expressive language over the recent years. It has imperative control constructs such as loops, conditionals [16], and recursive procedures, but also less standard constructs like the non-deterministic choice of actions. Extensions exist for dealing with continuous change [9] and concurrency [4], allowing for exogenous and sensing actions [8] and probabilistic projections into the future [9], or decision-theoretic planning [2] which employs Markov Decision Processes (MDPs). Successful robotics application of Golog and its extensions can be found, for instance, in [6, 15, 18].

2.2 Lua

Lua [12] is a scripting language designed to be fast, lightweight, and embeddable into other applications.The whole binary package takes less then 200 KB of storage. When loaded, it takes only a very small amount of RAM. In an independent comparison Lua has turned out to be one of the fastest interpreted programming languages [13, 22]. Besides that Lua is an elegant, easy-to-learn language [11] that should allow newcomers to start developing behaviors quickly. Another advantage of Lua is that it can interact easily with C/C++. As most basic robot software is written in C/C++, there exists an easy way to make Lua available for a particular control software.

Lua is a dynamically typed language, attaching types to variable values. Eight different types are distinguished: *nil*, *boolean*, *number*, *string*, *table*, *function*, *userdata*, and *thread*. For each variable value, its type can be queried. The central data structure in Lua are tables. Table entries can be addressed by either indices, thus implementing ordinary arrays, or by string names, implementing associative arrays. Table entries can refer to other tables allowing for implementing recursive data types. For example `t["name"] = value1` stores the key-value pair (name, value1) in table t, while `t[9] = value2` stores the value2 at position 9 in array `t`. Special iterators allow access to associative tables and arrays. Note that both index methods can be used for the same table.

Function are first-class types in Lua and can be created at runtime, assigned to a variable, or passed as an argument, or be destroyed. Lua provides proper tail calls and closures to decrease the needed stack size for function calls. Furthermore, Lua offers a special method to modify code at run-time. With the `loadstring()` statement chunks of code (one or more instruction of Lua code is called chunk) can be executed at run-time. This comes in handy to modify code while you are running it.

2.3 The Prototype golog.lua

In [5], we presented a first prototypical implementation of a vanilla Golog interpreter in Lua. The motivation was there as it is here, to make Golog available for a larger number of platforms, as Lua in turn is available for a larger number of embedded systems. We will briefly show, how situations and programs are represented in `golog.lua`.

In Lua, everything is a table, as tables are the central and only data structure. Hence, situation terms, actions, fluents, and programs are also represented as nested tables. The following example code shows a Golog program encoded as a Lua table.

```
prog = {{"a_1", {}}, {"a_2", {"x_1", "x_2"}},
  {"if", {cond}, {"a_3", {}}, {"a_4", {}}}}
local s_2, failure = Do(prog, {})
```

The program above consists of an 0-ary action a_1 in sequence with $a_2(x_1, x_2)$ and a conditional which, depending on the truth value of $cond$, chooses a_3 or a_4, resp. The program is executed with calling the interpreter function Do[3] which takes a program and a situation term, and returns the resulting situation after executing the program, or, if the program trace leads to a failure, i.e. the failure variable is true, s_2 contains the last possible action. Assuming that *cond* holds, the resulting execution trace of the prog will be

```
s_2 =  {{"a_1",{}},
   {"a_2", {"x_1", "x_2"},{"a_3,{}}}}
```
[4]

We use the empty table or *nil* to represent S_0. The above situation term is to be interpreted as $do(a_3,do(a_2(x_1,x_2), do(a_1, S_0)))$. Similarly, we represent logical formulas as tables, with the connectives in prefix notation, i.e. $\{\texttt{and}, \phi, \{\texttt{or}, \psi, \theta\}\}$ represents the formula $\phi \wedge (\psi \vee \theta)$. We refer to [5] for a more concise introduction of the implementation of Golog in Lua.

3 The next step: indigolog.lua on the NXT

In this section, we show the next step after a vanilla Golog interpreter in Lua. We present the corner stones on the way to deploy IndiGolog on the embedded platform Mindstorm NXT. First, we introduce the library pbLua, which allows for running Lua on the NXT and provides a number of Lua functions to access the robot's sensors and actuators. Then, we outline the IndiGolog, the online Golog dialect which we have chosen for this project, and briefly discuss the differences to the fore-mentioned vanilla Golog. Finally, we show the implementation of the main loop of our new IndiGolog interpreter in Lua and introduce a target robotics domain for our approach.

3.1 pbLua

pbLua is a library from the Hempel Design Group [10] which provides a LUA interpreter for the NXT. Moreover, pbLua offers a complete API to access sensors and actuators on the NXT via Lua functions. The following examples are taken from [10].

The first example shows how to access the motor which is plugged into port 1 of the NXT:

```
-- Turn Motor 1 exactly 180 degrees at half speed
port = 1
nxt.OutputSetRegulation(port,SINGLE_MOTOR,USE_BREAK)
nxt.OutputSetSpeed(port,NO_RAMP,50,180)
```

The first statement sets the control option for the motor connected to port 1 (standard control for a single motor using the break of the motor). Then motor 1 is commanded to turn for 180° with 50% of the maximum speed not using a ramp for the acceleration/deceleration.

The second example shows Lua code that returns the values read from the light sensor until the orange button on the NXT is pressed.

```
-- Read light sensor until the orange button is pressed
function LightRead(port,active)
  active = active or 0
  if 0 == active then nxt.InputSetType(port,LS_WI,SV)
                 else nxt.InputSetType(port,LS_WOI,SV)
  end
  repeat
    print( nxt.TimerRead(), nxt.InputGetStatus(port) )
  until( ORANGE_BUTTON == nxt.ButtonRead() )
end
```

First the mode of the sensor on port 1 is configured. The different flags are: LS_WI (light sensor using own illumination), LS_WOI (light sensor using ambient illumination) and SV (provide scaled values in the range of 0 to 100%). Then read and display the value of the sensor until the orange button is pressed.

The pbLua API offers a complete interface to the sensors and actuators of the NXT and is therefore well-suited for our task.

3.2 IndiGolog

The major drawbacks of vanilla Golog [16] is that it does not allow for sensing and can be only interpreted in an offline fashion. These facts obviously limit the value of Golog for the control of autonomous robots which are deployed in a dynamic and uncertain world. In order to overcome this limitations several successors to Golog have been developed. One of them is IndiGolog (incremental deterministic Golog) [8] which allows among others for online interpretation, sensing actions and concurrent actions. We decide to use this Golog dialect for our embedded reasoning system as it provides all features necessary for intelligent robust robot control.

IndiGolog also makes use of a situation calculus basic action theory. The way how IndiGolog interprets programs, is however different. IndiGolog uses a transition semantics. That means that the input program is interpreted in a step-by-step fashion. The interpreter directly commits to actions, i.e. once a primitive action is interpreted and it is possible to execute it, it will be executed in the real world. This is the main difference to the so-called evaluation semantics of vanilla Golog, where the program is first evaluated until the end. In IndiGolog, the program is transformed from one configuration $\langle \sigma, s\rangle$ to a legal successor configuration $\langle \delta, s'\rangle$ by means of a predicate $Trans(\sigma, s, \delta, s')$. Termination conditions of the program are defined with a predicate $Final(\sigma, s)$, which states when the program σ may legally terminate. The semantics of the program statements are hence defined by predicates $Trans$ and $Final$. To give an example, we show the example of the Trans and Final predicates of a primitive action and a loop.

$$
\begin{aligned}
&Trans(\alpha, s, \delta, s') \equiv Poss(a[s], s) \wedge \delta = nil \wedge s' = do(a, s)\\
&Final(\alpha, s) \equiv false\\
&Trans(\mathbf{while}\ \varphi\ \mathbf{do}\ \delta_1\ \mathbf{end}, s, \delta, s') \equiv\\
&\qquad \varphi[s] \wedge \exists\delta'.\delta = (\delta'; \mathbf{while}\ \varphi\ \mathbf{do}\ \delta_1\ \mathbf{end}) \wedge Trans(\delta, s, \delta', s')\\
&Final(\mathbf{while}\ \varphi\ \mathbf{do}\ \sigma\ \mathbf{end}, s) \equiv \varphi[s] \wedge Final(\sigma, s)
\end{aligned}
$$

If the primitive action is possible, i.e. its precondition axiom $Poss(a[s], s)$ holds, the action is executed. The successor situation is the one where the action was executed, i.e. $s' = do(a, s)$, and the successor configuration is $\delta = nil$. The final configuration is obviously not reached, as the program is transformed to the *nil*-program, which in turn defines the final configuration. The loop works as follows. If the condition holds, the successor configuration consists the loop body in sequence with the while loop itself to ensure that another iteration of the loop can be executed in the next transition step. The loop reaches a legal final configuration, when the condition holds and the loop body terminates.

The second advantage of IndiGolog over vanilla Golog is that is has a direct account to sensing. The basic action theory is extended with so-called sensing actions and exogenous actions. These are actions that directly connect a robot's sensor with a fluent. Each time, the sensing action is executed, the program gets an update of the re-

[3] Note that IndiGolog, which we introduce in the next section, does not make use of Do, but uses a transition semantics which is defined with functions trans and final.

[4] Note that all program statements, actions, and fluent names must be given as strings. For reasons of readability, we omit the quotation marks throughout this paper. Note also that Lua supports to return multiple value, the situation term and the failure condition in this case.

spective sensing result. Similarly, exogenous actions are very useful to model interaction between the robot and its environment.

3.3 Some Implementation Details

In the following, we give some implementation details of our IndiGolog implementation in Lua. Basically, we present the interpreter main loop, which shows the respective calls to functions `trans()` and `final()`, which in turn interpret the input program `p` (which we omit here). In line 8 of the function `indigo()` it is checked, whether a transition in the program `p` is possible in the situation `s`. The function `trans:check(p, s)` extracts the first program instruction of `p` and returns the transformed program (as defined in the previous section) in the member variable `trans.delta` (cf. line 21). If `trans:check()` returns `false`, it is checked if a final configuration was reached (line 13), otherwise, the program execution was unsuccessful (line 17).

```
   function indigo(p, s)
      trans.s  = s
      local tv -- did final/trans evaluate to true?
      repeat
5        -- did an exogenous action occur?
         if exog_occurs then exog_action() end

         -- is a transition possible?
         tv = trans:check(p, s)
10
         -- check for termination condition otherwise
         if not tv then
            if final:check(p, s) then
               print("Program terminates successfully\n")
15             return
            else --
               print("Program terminates unsuccessfully\n")
               return
            end
20       end
         p = trans.delta -- remainder program
         s = trans.s -- new situation term
      until false
   end
```

Similar as in [5], the basic action theory has to be specified. Each action is stored as an object with its own metatable, i.e. `act = action:new(name, arg1, ...)`. Then, one has to specify the precondition axiom in form of a function `act.Poss(s,arg1, ...)`. Similarly, a function `act.Execute(arg1, ...)` is required. This defines the interface to the real world and can be used to execute, say, drive actions on the real robot hardware. For each fluent occurring in basic action theory, one has to specify an effect axiom of the form `fluent.action(s, arg1, ...)` together with a function `fluent.initially(arg1, ...)` which defines the initial value of the respective fluent. When a fluent formula is evaluated, fluent regression is applied by subsequently calling the particular effect axioms. Assume we have a fluent f. Moreover, we have two primitive actions a_1 and a_2 which are executed in sequence starting in S_0 leading to a new situation S. In our lua-based implementation the value of the fluent f in situation S is evaluates by the recursive function call `f.a2({a1}, f.a1({}, f.initially({})))`.

Note that we assume a closed world here. More details about this can be found in [5].

3.4 The Target Domain

We plan to use the Lua-based implementation of IndiGolog to solve the following delivery robot domain which is an extension of the original example used in [15]. The robot has to deliver goods between offices. The example domain is shown in Figure 2.

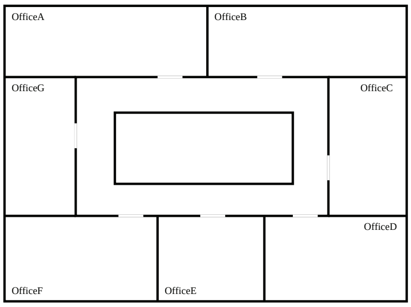


Figure 2. Example domain for the delivery robot.

The domain comprises several offices connected by a corridor. In order to ease the navigation for the Lego Mindstorm NXT based robot in reality there is a black line on the floor connecting the rooms. The robot can simply follow this line. The line may branch on some junctions. In order to allow the robot to detect if it has reached a room or junction there are silver spots on the floor. Please note that branches at junctions are always orthogonal and marked with their direction, e.g., N for the northbound branch. Junctions are always located in front of an office door. Connection lines between rooms and junctions itself need not to be straight. The robot uses four standard Lego Mindstorm NXT sensors: (1) a light sensor for following the black line, (2) a light sensor to detect the silver spots, (3) a touch sensor to detect an obstacle and (4) a touch sensor to detect if a package is currently loaded.

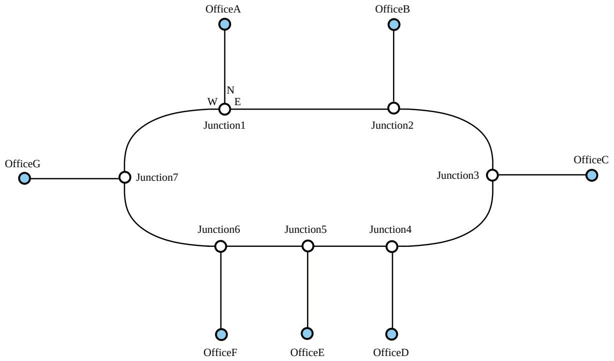


Figure 3. Topological map of the example domain.

The robot uses a graph-based topological map to find its way between the office. The corresponding graph for the example domain is depicted in Figure 3. The robot receives delivery orders to bring goods from one office to another office. In order to be able to fulfill its task the robot needs some planning capabilities in order to decide which order to process next and to plan its path from the origin to the destination. Moreover, paths between junctions and office may be blocked by one or more obstacles. If a path is blocked is initially not known to the robot. Therefore, the robot has to deal with incomplete knowledge and it needs also sensing and re-planning capabilities.

IndiGolog is obviously suitable to control the robot for this task. In order to solve the task with we have to specify fluents, actions, the influence of sensing and domain-dependent facts.

For instance we need the following fluents among others:

- $at(loc)$: this fluent is true if the robot is at the corresponding junction or room *loc*
- $direction(dir)$: the robot faces towards a particular direction *dir* at a junction
- $loaded(package)$: the robot carries *package*
- $blocked(node_1, dir_1, node_2, dir_2)$: the path between the nodes *node₁* and $node_2$ via the directions dir_1 and dir_2 is blocked

Moreover, we need domain-dependent facts:

- $connect(node_1, dir_1, node_2, dir_2)$: the nodes $node_1$ and $node_2$ are connected via the directions dir_1 and dir_2
- $order(origin, destination, package)$: there exists a delivery order for $package$ from $origin$ to $destination$

In order to serve the the different delivery orders we define the following control procedure:

```
control = procedure:new('control', {})
control.body = { {'while', {'some', {o, d, p},
      {'order', {o, d, p}},
      {serve_order', {o, d, p}}}}}}
```

The procedure $control$ continues to chose non-deterministically an order and process it until no open orders exist anymore. We use existentially quantified statement `some`. This leads us to the procedure $serve\_order$:

```
serve_order = procedure:new(serve_order', {o, d, p})
serve_order.body = { {'goto', {o}}, {'pickup', {p}},
  {'goto', {d}}, {'deliver', {p}}}
```

$serve\_order$ execute sequentially the actions go to origin o, picks up the package p, go to destination d and finally deliver the package p. Where the procedure $goto(g)$ implements the navigation towards the goal g.

```
goto = procedure:new('goto', g)
goto.body = { {'while', { {'not', {'at', g}}},
    {'?', {'some', {d}, {'next_dir', d}}},
    {'turn', {d}}, {'drive'}}}
```

As long as the final goal g is not reached, expressed by the formula $\neg at(g)$ as condition, the procedure chooses the next suitable direction on the junction, turns toward the corresponding branch and moves along the branch until it reaches the next node. Please note the predicate $next\_dir(d)$ is true for the next suitable direction towards the goal.

This leads us to the basic primitive actions. For instance the primitive action $turn(d)$ ensures that the robot faces towards the direction d, $d \in \{N, E, S, W\}$. The primitive actions actually control the robot's hardware via the corresponding pbLua commands. Please note that for simplicity the directions are represented internally by the set of integers $\{0, 1, 2, 3\}$.

```
turn = action:new('turn', 1) -- turn action has arity 1

function turn.Poss(s,dir)
   return true
end

function turn.Execute(s,dir)
   local act_dir = GetFluent('direction')
   local dir_diff = dir - act_dir
   if dir_diff < 0 then dir_diff = 4 + dir_diff end
   for i=dir_diff, i>0 do
      repeat
         nxt.OutputSetSpeed(MOTOR_A,NO_RAMP,50)
         nxt.OutputSetSpeed(MOTOR_B,NO_RAMP,-50
      until (nxt.InputGetStatus(SENSOR_1) < BLACK_LIMIT)
      nxt.OutputSetSpeed(MOTOR_A,NO_RAMP,0)
      nxt.OutputSetSpeed(MOTOR_B,NO_RAMP,0)
   end
end
```

The primitive action $turn$ can be executed all time and has therefore a precondition which is always true.

The executable part of the action first calculate how many orthogonal turn it has to perform to reach the desired direction. Once it knows the number of turns it simple repeats to turn on the spot until the line sensor detects the next orthogonal branch. This fact is detected if the returned value of the light sensor falls below the threshold `BLACK_LIMIT`.

The other remaining primitive action are defined in a similar way.

4 Conclusion and Future Work

In this paper, we showed some ongoing work to make the robot programming and plan language Golog and its derivatives available for a larger number of robot platforms. Usually, Golog interpreter are implemented in Prolog, which is straight-forward regarding the fact that the specification of Golog is given in first order logic. However, Prolog might not be available for all robot platforms. Therefore in [5], a first prototypical Lua implementation of a vanilla Golog interpreter was presented. As vanilla Golog is only of limited usability for real robots, we here show the next steps to apply Golog-type languages to real robots. We started to re-implemented IndiGolog in Lua with the goal to deploy it on the platform Lego Mindstorm NXT. IndiGolog features a online execution semantics and is able to integrate sensor actions and exogenous events, which is particularly useful for real robot applications. As this is ongoing research, we are only able to show the preliminary implementation of IndiGolog on a Linux-based Lua interpreter here. Currently, we working towards the actual deployment of the interpreter under pbLua on the Lego Mindstorm NXT.

Due to the very limited memory on the NXT, one challenge is to fit the interpreter, the domain description and the IndiGolog program onto the NXT. One possible way to do this could be to set up a cross-compile chain for the ARM CPU and just deploy the Lua byte-code on the NXT. This is due to further investigation, but by the time of the workshop we are convinced to have first results available. The future work also includes the full integration of sensing, the investigation of the correctness of the implementation as well as giving run-time results of the performance of the interpreter, both on the Mindstorm NXT as well as on the humanoid robot Nao.

In particular the Nao application is interesting. In previous work the Golog dialect ReadyLog [6] was used to control soccer robots of the RoboCup Middle Size League. The complexity of the RoboCup Standard Platform League is comparable to that domain which was modeled using about 100 fluents, 40 primitive actions and additional 15000 line of code for interfacing the robot hardware. How to fit such a control model into a limited system such as the Nao will be a real challenge.

REFERENCES

[1] Michael Beetz, 'Structured Reactive Controllers', *Autonomous Agents and Multi-Agent Systems*, **4**(2), 25–55, (2001).

[2] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun, 'Decision-theoretic, high-level agent programming in the situation calculus', in *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pp. 355–362. AAAI Press, (2000).

[3] Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu, 'Bridging the gap between high-level reasoning and low-level control', in *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pp. 342–354. Springer, (2009).

[4] G. De Giacomo, Y. Lsperance, and H. Levesque, 'ConGolog, A concurrent programming language based on situation calculus', *Artificial Intelligence*, **121**(1–2), 109–169, (2000).

[5] Alexander Ferrein, 'lua.golog: Towards a non-prolog implementation of golog for embedded systems', in *Cognitive Robotics*, eds., Gerhard Lakemeyer, Hector Levesque, and Fiora Pirri, number 100081 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, (2010). to appear.

[6] Alexander Ferrein and Gerhard Lakemeyer, 'Logic-based robot control in highly dynamic domains', *Robotics and Autonomous Systems, Special Issue on Semantic Knowledge in Robotics*, **56**(11), 980–991, (2008).

[7] Alexander Ferrein, Gerald Steinbauer, Graeme McPhillips, and Anet Potgieter, 'RoboCup Standard Platform League - Team Zadeat - An Intercontinental Research Effort', in *International RoboCup Symposium*, Suzhou, China, (2008).

[8] Giuseppe De Giacomo, Yves Lesprance, Hector J. Levesque, and Sebastian Sardina, *Multi-Agent Programming: Languages, Tools and Applications*, chapter IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents, 31–72, Springer, 2009.

[9] Henrik Grosskreutz and Gerhard Lakemeyer, 'ccgolog – A logical language dealing with continuous change.', *Logic Journal of the IGPL*, **11**(2), 179–221, (2003).

[10] Ralph Hempel. pblua – scripting fot the LEGO NXT. http://www.hempeldesigngroup.com/lego/pblua/, 2010. (last visited on May 21, 2010).

[11] Ashwin Hirschi, 'Traveling Light, the Lua Way', *IEEE Software*, **24**(5), 31–38, (2007).

[12] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho, 'Lua - An Extensible Extension Language', *Software: Practice and Experience*, **26**(6), 635 – 652, (Jan 1999).

[13] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho, 'The Evolution of Lua', in *Proceedings of History of Programming Languages III*, pp. 2–1 – 2–26. ACM, (2007).

[14] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti, 'The saphira architecture: A design for autonomy', *Journal of Experimental and Theoretical Artificial Intelligence*, **9**, 215–235, (1997).

[15] Hector J. Levesque and Maurice Pagnucco, 'Legolog: Inexpensive experiments in cognitive robotics', in *Proceedings of the Second International Cognitive Robotics Workshop*, Berlin, Germany, (2000).

[16] Hector J. Levesque, Raymond Reiter, Yves Lesprance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *The Journal of Logic Programming*, **31**(1-3), 59 – 83, (1997). Reasoning about Action and Change.

[17] J. McCarthy, 'Situations, Actions and Causal Laws', Technical report, Stanford University, (1963).

[18] H. Pham, *Applying DTGolog to Large-scale Domains*, Master's thesis, Department of Electrical and Computer Engineering, Ryerson University, Toronot, Canada, 2006.

[19] Raymond Reiter, *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, 2001.

[20] Stuart Russell and Pater Norvig, *Artificial Intelligence: A Modern Approach (Second Edition)*, Prentice Hall, 2003.

[21] Paul Talaga and Jae C. Oh, 'Combining AIMA and LEGO mindstorms in an artificial intelligence course to build realworldrobots', *Journal of Computing Science in Colleges*, **24**(3), 56–64, (2009).

[22] The Debian Project. The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/. retrieved Jan 30th 2009.

A domain language for expressing engineering rules and a remarkable sub-language

Guy A. Narboni [1]

Abstract. We describe and analyse a constraint language for expressing logic rules on variables having a finite integer range. Our rules can involve linear arithmetic operations and comparisons to capture common engineering knowledge. They are capable of bidirectional inferences, using standard constraint reasoning techniques. Further, we show that for a restricted sub-language the basic propagation mechanism is complete. For systems in that class, local consistency implies global consistency.

This noteworthy property can be instrumental in assessing the scalability of an Expert System, which in practice is a cause for concern with the growth of data. When the knowledge base passes a syntactic check, we can guarantee that its solution is search-free and therefore backtrack-free. By way of an illustration, we examine the worst case complexity of a dynamic product configuration example.

1 INTRODUCTORY EXAMPLE

Rules are a natural means for stating general principles, legal or technical. For instance in Europe, any installation distributing electrical power must comply with the IEC standards. A mandatory safety requirement stipulates that: *in a humid environment, the degree of protection provided by enclosures must be rated IPX1*.

Formally, we have a logical implication:

$$\texttt{env} = \texttt{humid} \Rightarrow \texttt{code} = \texttt{IPX1}$$

In that statement, the environment characteristic can only take two values (`dry` or `humid`), whereas the IP code has a wider but still finite set of choices (including the `IP3X`, `IP4X` and `IPX1` values). The above rule therefore pertains to propositional calculus.

With respect to Boolean satisfiability, an equivalent formulation, using binary variables only, is given by the clausal system:

$$\texttt{h} \Rightarrow \texttt{c}_1$$
$$\texttt{c}_3 \vee \texttt{c}_4 \vee \texttt{c}_1$$

where `h` is true when the environment is humid and where $\texttt{c}_1$ (resp., $\texttt{c}_3$, $\texttt{c}_4$) is true when the degree of protection is `IPX1` (resp., `IP3X`, `IP4X`). The additional disjunction ensures that the equipment has indeed an IP code (*i.e.*, $\texttt{c}_3$, $\texttt{c}_4$ and $\texttt{c}_1$ cannot be simultaneously false)[2].

It appears that, although the clause $\texttt{h} \Rightarrow \texttt{c}_1$ is both Krom and Horn (it has exactly 2 literals, and no more than one is positive), the resulting system is neither Krom nor Horn. Now, those are the two best known classes of rules for which the Boolean satisfiability problem can be solved in reasonable time. So, should we fear the worst case behaviour (*i.e.*, a combinatorial explosion) when having to reason with larger rule sets translating conditions of such a simple kind? Until proven otherwise, presumably.

In this paper, we show that for some specific systems (including the above example), such guarantees can be given.

To start with, we define a small constraint language for expressing logic rules on variables having a finite integer range. We observe that forward and backward inferences can be performed rule-wise, by narrowing the variables' domains. Then, assuming that consistency is checked locally, at the rule level, using the general propagation mechanism for constraint solving, we prove that, in a special case, we get global consistency for free. Rules in that remarkable case are easily recognizable from their syntactic structure.

Coming from tractability theory, this result generalizes the Horn propositional case. Despite the relative narrowness of the class of rules identified, we give a full-fledged example to which it is applicable. We thus conclude that it should be of practical interest for the handling of vast knowledge bases, in areas such as dynamic product configuration where scalability is a major concern.

2 LOGIC RULES

We shall resort to first order logic for expressing the rules.

Rules are clearly easier to read and write using attribute - value pairs, since variables are directly associated to real world domain values. Modeling tools for engineers such as the Component Description Language for Xerox machines [1] or the EXPRESS product data specification language [2] often follow that convention.

We'll further assume that:

- the problem's dimension *n* is known
  (we have a finite number of attributes)
- all attribute domains are finite and —for convenience— integer valued
  (the latter is not a true restriction since one can always use a one-to-one mapping for encoding values).

[1] Implexe, France, email: r-d@implexe.fr

[2] If need be, 3 extra clauses will forbid multiple selections:

$$\texttt{c}_3 \wedge \texttt{c}_4 \Rightarrow \texttt{false}$$
$$\texttt{c}_3 \wedge \texttt{c}_1 \Rightarrow \texttt{false}$$
$$\texttt{c}_4 \wedge \texttt{c}_1 \Rightarrow \texttt{false}$$

2.1 Language definition

In our language, constants are integers. Variables correspond to attribute names. The functional expressions allowed are linear expressions with integer coefficients. The domain of interpretation is the set of integers with the usual arithmetic operations and the comparison relations =, ≤ and ≥.

A fact or atomic formula is a diophantine equation or inequality, *i.e.*, a constraint of the form :

linexp comparator constant

where *linexp* denotes a linear expression in (at most) n variables. An atomic formula can be used to state a fact or a basic restraint. Often, there will be just one variable with a coefficient of 1, the other variables having zero coefficients, as in $\texttt{code} \geq \texttt{1}$.

A conjunctive formula is defined recursively: an atomic formula is a conjunctive formula, and so is the conjunction of an atomic formula with a conjunctive formula. For instance, the constraint model $\mathbf{A}\boldsymbol{x} \leq \boldsymbol{b}$ of an integer program (IP) is a conjunctive formula.

A rule or conditional formula is an implication constraint:

conjunctive formula ⇒ *atomic formula*

It is a restricted form of rule with possibly several antecedents but only one consequent. An example is a car configuration statement excluding the air conditioning option when the battery and the engine are of small size (*e.g.*, 1 on a scale of 3), which writes:

$$\underset{\texttt{(small)}}{\texttt{battery} = \texttt{1}} \wedge \underset{\texttt{(small)}}{\texttt{engine} = \texttt{1}} \Rightarrow \underset{\texttt{(excluded)}}{\texttt{aircond} = \texttt{0}}$$

Lastly, a knowledge base is a finite conjunction of, say m, formulas which can be either rules or facts[1]. We can view it formally as a quantifier-free Presburger arithmetic formula.

2.2 Global consistency checking

Proving the consistency of a knowledge base is a central issue in automated deduction. This question applies to constraint programs made of logic rules.

During an interactive session, new facts or restraints are typically added to the rule base which grows incrementally. Therefore, a truth maintenance system repeatedly needs to check the satisfiability of a closed formula:

$$\exists\, \boldsymbol{x} \quad R_1(\boldsymbol{x}) \wedge ... \wedge R_m(\boldsymbol{x}) \wedge \boldsymbol{x} \in \mathrm{D}_1 \times ... \times \mathrm{D}_n \qquad (1)$$

where each n-ary constraint R_i in the conjunct states a fact or a rule, and each unary range predicate $x_j \in \mathrm{D}_j$ in the cross-product sets a domain constraint (*i.e.*, an integer constraint plus a pair of linear inequalities, basically).

An inference system will be complete if it is able to answer *no* to the existential query as soon as the constraint system becomes inconsistent.

The formula will be satisfiable if there is some instantiation of the variables by constants from the domains which makes it true (an instantiation makes each atomic constraint either true or false, and consequently, the formula can be evaluated to be true or false).

Because there are finitely many possible assignments, the satisfiability problem is decidable. However, the decision procedure is equivalent in complexity to SAT solving. *E.g.*, determining whether there exists a valid configuration for a set of rules can be a formidable challenge (the general problem being NP-complete) [3]. In practice, solution methods involve a search process —and don't scale well.

2.3 Finite domains solvers

General-purpose solvers routinely used in constraint programming languages are by design incomplete. Their deductive power is a trade-off between inference speed and strength. They check local consistency conditions that are necessary but not sufficient for ensuring global consistency.

Most of the time, the initial domain filtering step (propagation) will have to be followed up by —and interlaced with— a programmable enumeration phase (labeling), in order to answer the question.

3 INFERENCES AS DOMAIN REDUCTIONS

3.1 Database semantics

A logic rule defines a relation. Interestingly, without resorting to constraints, our introductory example rule could not be expressed *intensionally* as a pure Prolog program, because of the Horn limitation. To do so, we would need to extend logic programs with disjunction [4] [5].

However, the semantics of the virtual relation R a logic rule defines can be expressed by an *extensional* database:

$$\{\boldsymbol{x} \mid R(\boldsymbol{x}) \wedge \boldsymbol{x} \in \mathrm{D}_1 \times ... \times \mathrm{D}_n\}$$

The set of points (or tuples) that satisfies the formula corresponds to the disjunctive normal form.

Assume we use the following encodings in our example:

- `0` for `dry`, `1` for `humid`
- `-1`, `0` and `1` for `IP3X`, `IP4X` and `IPX1`, respectively.

Then, the intensional model is the logical implication:

$\texttt{env} = \texttt{1} \Rightarrow \texttt{code} = \texttt{1}$

together with the finite domain constraints at the source of the disjunctions:

$\texttt{env} \in \{\texttt{0}, \texttt{1}\}$ and $\texttt{code} \in \{\texttt{-1}, \texttt{0}, \texttt{1}\}$

The extensional model is the relational table:

$R(\texttt{env}, \texttt{code}) = \{(\texttt{0}, \texttt{-1}), (\texttt{0}, \texttt{0}), (\texttt{0}, \texttt{1}), (\texttt{1}, \texttt{1})\}$

The compatibility constraint thus rules out 2 out of 6 possibilities.

Figure 1 depicts the relation R on a 2-dimensional lattice. We've highlighted the convex hull of its solution points which shows a least element, (`0`, `-1`) in the plane. We'll go back later over this special geometrical property.

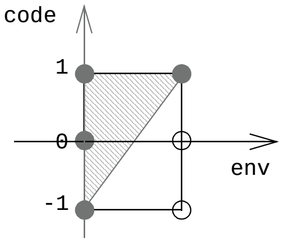


Figure 1. 2D representation

3.2 Compiled vs. interpreted approaches

Once a relation is explicitly stored in an extensional table, the search for solutions is linear in the size of that table. However, the number of rows exponentially grows with the number of variables. With decision diagrams, compiled versions may enjoy much more compact forms (depending, of course, on the variable ordering).

[1] We'll include the facts into the rule base by likening a fact to a rule with no antecedent.

But again, there is no guarantee that the process of generating the table will terminate is reasonable time.

The main advantage of the compilation approach is that the bulk of the work can be carried out off-line. The drawback is that the rule base then becomes static. When one has to deal with dynamicity, the alternative is a logic rule interpreter. This is the choice implicitly made in the sequel, through the use of a contraint solver for implementing the inference engine.

3.3 Constrained queries

The standard conditions that can be expressed in the `where` clause of a database selection query closely resemble our conjunctive formulas —at least, they share the same syntax.

3.3.1 Modus ponens

The SQL query:

```
select code from R where env = 1
```

gives a unique solution:

Answer(`code`) = `{1}`

`code = 1` is also the result obtained when applying the rule in forward chaining mode, in a cause and effect relationship. This corresponds to the classical inference:

from `env = 1` and `env = 1` ⇒ `code = 1` we draw `code = 1`

Through the angle of constraint solving, the effect of applying a rule amounts to domain narrowing. One local consistency check performed on the domains is the following: when the domain D_e of `env` is reduced to `{1}`, the domain D_c of `code` is narrowed down to $D_c \cap$ `{1}`. Thus, if $D_c \cap$ `{1}` is empty, inconsistency is immediately detected.

On the other hand, a query like:

```
select code from R where env = 0
```

yields three solutions for the IP code since there is no restriction then on the degree of protection. Consequently, there will be no domain reduction.

3.3.2 Modus tollens

The symmetric query:

```
select env from R where code = 0
```

gives again a unique solution:

Answer(`env`) = `{0}`

This time, `env = 0` is a result that cannot be obtained directly, in forward or backward chaining mode, from:

`env = 1` ⇒ `code = 1`

since it doesn't match the antecedent nor the consequent of the rule. It is indirectly the result of the contrapositive of the rule, *i.e.*, the implication:

`code` ≠ `1` ⇒ `env` ≠ `1`

The contrapositive can be expressed in our language as:

`code` ≤ `0` ⇒ `env = 0`

But there is no need to add it to the rule base, since the implication and its contrapositive are logically equivalent. The engine takes care of it through a local consistency check: when the domain D_c of `code` is reduced to `{-1, 0}`, the domain D_e of `env` is narrowed down to $D_e \cap$ `{0}`. Thus, if $D_e \cap$ `{0}` is empty, inconsistency is immediately detected.

It should be clear from the above that local consistency enforcement at the rule level can endow us with versatile deductive inference capabilities, through domain reductions.

3.4 What propagation guarantees

Local consistency is a prerequisite for global consistency. The preceding examples required only bounds-consistency checking (stronger degrees of consistency can be defined).

Before moving to the system level (*i.e.*, to the consistency analysis of a set of rules), we need to recall the main properties of the general filtering mechanism which is at work with a finite domain solver [6].

While preserving equivalence, the propagation mechanism basically transforms formula (1) into formula (2):

$$\exists\, \boldsymbol{x} \quad R_1(\boldsymbol{x}) \wedge \ldots \wedge R_m(\boldsymbol{x}) \wedge \boldsymbol{x} \in \underline{D}_1 \times \ldots \times \underline{D}_n \qquad (2)$$

The structural constraints R_i of (1) are unchanged. Only the variable domains may change and be contracted from D_j to $\underline{D}_j$. The reduced domain $\underline{D}_j \subseteq D_j$ is the result of a fixpoint computation in which all inter-related rules take part. Note that domain constraints are the sole ones handled globally.

No matter how local consistency is defined and implemented, the following property hold:

1. if there are any solutions to the system, then the solutions lie within the bounds specified by the cross-product of the reduced domains.

In other words, there are no solutions outside. Should one domain become empty, then we have the proof that the system is inconsistent.

When no domain is empty, the following properties equally hold:

2. for each constraint R_i, the formula (3*i*) is satisfiable:

$$\exists\, \boldsymbol{x} \quad R_i(\boldsymbol{x}) \wedge \boldsymbol{x} \in \underline{D}_1 \times \ldots \times \underline{D}_n \qquad (3i)$$

3. in particular, for each variable x_j, the formulas ($3ij^{\min}$) and ($3ij^{\max}$) are satisfiable:

$$\exists\, \boldsymbol{x} \quad R_i(\boldsymbol{x}) \wedge x_j = min(\underline{D}_j) \wedge \boldsymbol{x} \in \underline{D}_1 \times \ldots \times \underline{D}_n \qquad (3ij^{\min})$$

$$\exists\, \boldsymbol{x} \quad R_i(\boldsymbol{x}) \wedge x_j = max(\underline{D}_j) \wedge \boldsymbol{x} \in \underline{D}_1 \times \ldots \times \underline{D}_n \qquad (3ij^{\max})$$

In other words, each rule does its best to narrow down the domains (*e.g.*, by checking the consistency of partial assignments involving maximum and minimum domain values). When all domains reduce to a point, *i.e.*, when $min(\underline{D}_j) = max(\underline{D}_j)$, those properties guarantee that the unique corresponding assignment is indeed a solution. However in general, nothing certifies that a satisfying assignment for one constraint is also a satisfying assignment for another constraint.

We now come to a very specific case for which we can prove, thanks to a distinguished witness point, that local consistency automatically enforces global consistency.

4 A REMARKABLE CLASS OF CONDITIONAL CONSTRAINTS

Using vector notation, we'll write $\boldsymbol{x} \leq \boldsymbol{y}$ whenever $\boldsymbol{x}$ is greater than $\boldsymbol{y}$ component-wise. The binary relation ≤ defines a partial order in the space. If $\boldsymbol{x}$ and $\boldsymbol{y}$ are two integer points, their least upper bound is the point $\boldsymbol{z} = \max\,\{\boldsymbol{x}, \boldsymbol{y}\}$ with integer coordinates $z_j = \max\,\{x_j, y_j\}$.

4.1 Semilattices

A partially ordered set S is said to be a join- (resp., meet-) semilattice if for all elements $\boldsymbol{x}$ and $\boldsymbol{y}$ of S, the least upper bound (resp., the greatest lower bound) of the pair $\{\boldsymbol{x}, \boldsymbol{y}\}$ exists.

If S is a finite join-semilattice, it follows that every non-empty subset has a least upper bound. So, if S is non empty, it has a least upper bound, max(S), which is the maximum (or greatest element) of S.

Now, with S referring to the extensional database, this is the remarkable property that every constraint of our class will exhibit: the presence of a "max" (resp., a "min") element. Following [7], we can express the same property conveniently by referring to the intensional relation R.

4.2 Max-closed constraints

4.2.1 *Motivation for study*

Cooper and Jeavons claim that max-closed constraints [7] give rise to a class of tractable problems which "generalizes the notion of a Horn formula in propositional logic to larger domain sizes". That is precisely what we are aiming at.

4.2.2 *Definition*

A relation R is said to be *max-closed* if:

whenever $R(\boldsymbol{x})$ and $R(\boldsymbol{y})$ hold, then $R(\max\ \{\boldsymbol{x}, \boldsymbol{y}\})$ holds.

4.2.3 *Example*

Constraints setting upper and lower bounds on the domains of variables, like $2 \le x_j$ or $x_j \le 5$, are max-closed.

More generally, all unary constraints are max-closed.

4.2.4 *Property of max-closed systems*

Consider a system with m max-closed constraints R_i in n unknowns ranging over finite domains. Then, if none of the reduced domains is empty, the greatest element of the cross-product $\underline{D}_1 \times ...\times \underline{D}_n$ satisfies the formula:

$$\exists\, \boldsymbol{x} \quad R_1(\boldsymbol{x}) \wedge ...\wedge R_m(\boldsymbol{x}) \wedge \boldsymbol{x} \in \underline{D}_1 \times ...\times \underline{D}_n \qquad (2)$$

4.2.5 *Proof*

If $\underline{D}_1 \times ...\times \underline{D}_n$ is not empty, the maximum element is $(\max(\underline{D}_1), ..., \max(\underline{D}_n))$. Let us first show that it satisfies the formula:

$$\exists\, \boldsymbol{x} \quad R_i(\boldsymbol{x}) \wedge \boldsymbol{x} \in \underline{D}_1 \times ...\times \underline{D}_n \qquad (3i)$$

Consider the partial assignment $x_j = \max(\underline{D}_j)$. By definition of the reduced domains, there is an instantiation $\boldsymbol{x}^j$ of $\boldsymbol{x}$ which satisfies the formula:

$$\exists\, \boldsymbol{x} \quad R_i(\boldsymbol{x}) \wedge x_j = max(\underline{D}_j) \wedge \boldsymbol{x} \in \underline{D}_1 \times ...\times \underline{D}_n \qquad (3ij^{\max})$$

We can thus exhibit n points $\boldsymbol{x}^1, ..., \boldsymbol{x}^n$ satisfying R_i.

Since R_i is max-closed, $\max\ \{\boldsymbol{x}^1, ..., \boldsymbol{x}^n\}$ belongs to R_i and it corresponds the point $(\max(\underline{D}_1), ..., \max(\underline{D}_n))$.

Since $(\max(\underline{D}_1), ..., \max(\underline{D}_n))$ satisfies $\boldsymbol{x} \in \underline{D}_1 \times ...\times \underline{D}_n$, as well as each constraint $R_i(\boldsymbol{x})$, it also satisfies their conjunction. Hence the property.

4.2.6 *Alternative view*

Geometrically speaking, the extensional database of a max-closed constraint is a finite semilattice with a greatest element. The non-empty intersection of such semilattices is a semilattice with the same property.

4.2.7 *Main consequence*

A max-closed system that is locally consistent is globally consistent. Finite domain propagation is therefore sufficient for global consistency checking.

4.2.8 *Corollary*

Any system of max-closed constraints is polynomial time solvable. This follows from the fact that the propagation algorithm is polynomial in the size of the constraint system and of the integer grid used for finitely discretizing the space (we here assume that grid is large enough to carry out all the computations accurately[1]).

4.3 Sub-language definition

We now restrict our attention to rule based systems only containing contraints of a special syntactic kind:

- inequalities of type m
- conditionals of type h.

By showing that those constraints have the max-closed property, we’ll derive a scalability certificate for the knowledge base. This means that sizeable problem instances can be checked for consistency without resorting to enumeration.

4.3.1 *max-closed inequalities*

Assuming that all the a_j’s are non-negative coefficients, we define a constraint of type m as:

- either of the form:

$$a_1x_1 + ... + a_nx_n \ge b$$

- or of the form:

$$a_1x_1 + ... + a_{k\text{-}1}x_{k\text{-}1} + a_{k+1}x_{k+1} + ... + a_nx_n \ge a_kx_k + b$$

In other words, a constraint of type m is a linear inequality $\boldsymbol{a’x} \ge b$ with at most one negative coefficient ($a’_k = -|a_k|$).

Property *Every inequality of type m is max-closed.*

To simplify the proof, we’ll write:

$$a_1x_1 + ... + a_kx_k + ... + a_nx_n \ge a^+x_k + b$$

where x_k can appear on both sides with non-negative coefficients (the trick is: if $a^+ = 0$, we are in the first case; if $a_k = 0$, we are in the second). Now assume with have:

$$a_1x_1 + ... + a_nx_n \ge a^+x_k + b$$

and: $\quad a_1y_1 + ... + a_ny_n \ge a^+y_k + b$

Let $z_j = \max\ \{x_j, y_j\}$.

Since $z_i \ge x_i$, $a_1z_1 + ... + a_nz_n \ge a_1x_1 + ... + a_nx_n \ge a^+x_k + b$

Since $z_i \ge y_i$, $a_1z_1 + ... + a_nz_n \ge a_1y_1 + ... + a_ny_n \ge a^+y_k + b$

Hence: $\quad a_1z_1 + ... + a_nz_n \ge \max\ \{a^+x_k + b, a^+y_k + b\}$

that is: $\quad a_1z_1 + ... + a_nz_n \ge a^+z_k + b$ ■

Let us briefly mention a direct and important consequence of that property: linear integer systems involving constraints of type m form a polynomial class [8]. When using finite domain solvers, propagation implements a global consistency check as well.

4.3.2 *max-closed conditionals*

We define a rule of type h as a conditional formula:

$$\mathbf{A}\boldsymbol{x} \le \boldsymbol{b} \quad \Rightarrow \quad \boldsymbol{ax} \ge a^+x_k + b$$

where

- $\mathbf{A}$ is a non-negative matrix (*i.e.*, all $\mathbf{A}$ entries are ≥ 0)
- the consequent is an inequality of type m.

Property *Every conditional of type h is max-closed.*

Assume again we have two satisfying instanciations $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\boldsymbol{a}_1\boldsymbol{x} \le b_1 \wedge ... \wedge \boldsymbol{a}_m\boldsymbol{x} \le b_m \Rightarrow \boldsymbol{ax} \ge a^+x_k + b$$

$$\boldsymbol{a}_1\boldsymbol{y} \le b_1 \wedge ... \wedge \boldsymbol{a}_m\boldsymbol{y} \le b_m \Rightarrow \boldsymbol{ay} \ge a^+y_k + b$$

Let $z_j = \max\ \{x_j, y_j\}$. By definition $\boldsymbol{a}_i \ge 0$, so we always have $\boldsymbol{a}_i\boldsymbol{x} \le \boldsymbol{a}_i\boldsymbol{z}$ and $\boldsymbol{a}_i\boldsymbol{y} \le \boldsymbol{a}_i\boldsymbol{z}$.

Hence, if $\boldsymbol{a}_1\boldsymbol{z} \le b_1 \wedge ... \wedge \boldsymbol{a}_m\boldsymbol{z} \le b_m$ is true, the conditions:

$$\boldsymbol{a}_1\boldsymbol{x} \le b_1 \wedge ... \wedge \boldsymbol{a}_m\boldsymbol{x} \le b_m$$

$$\boldsymbol{a}_1\boldsymbol{y} \le b_1 \wedge ... \wedge \boldsymbol{a}_m\boldsymbol{y} \le b_m$$

are both true.

We therefore have:

$$\boldsymbol{ax} \ge a^+x_k + b$$

and $\quad \boldsymbol{ay} \ge a^+y_k + b$.

[1] The price to pay can be a huge "constant" time.

The latter being a max-closed system[1], we infer:

$$\boldsymbol{az} \geq a^{+}z_k + b \ ■$$

4.3.3 *Level of generality*

Clearly, our sub-language us not universal. So, what does it encompass?

First, it allows to write rules with:

- true antecedents: $\mathbf{0}\boldsymbol{x} \leq \mathbf{1}$ (*i.e.*, no conditions for facts)
- false consequent: $\mathbf{0}\boldsymbol{x} \geq 1$ (for expressing integrity constraints).

So, right- and left-hand sides can be considered optional.

Second, by a duality argument, we have the min-closed property for the reverse order:

$$\mathbf{A}\boldsymbol{x} \geq \boldsymbol{b} \ \Rightarrow \ \boldsymbol{ax} \leq a^{+}x_k + b$$

Min-closed systems that are locally consistent are *ipso facto* globally consistent[2].

Third, we can express Horn propositional clauses

$$x_1 \wedge ... \wedge x_{n\text{-}1} \ \Rightarrow \ x_n$$

in two different ways:

- either as (dual) inequalities of type *m*

$$x_1 + ... + x_{n\text{-}1} \leq x_n + (n–2)$$

- or as (dual) conditionals of type *h*

$$x_1 \geq 1 \wedge ... \wedge x_{n\text{-}1} \geq 1 \ \Rightarrow \ x_n \geq 1$$

with $\boldsymbol{x} \in \{0,1\}^n$.

The fact that propagation implements unit resolution explains the good behaviour of finite domain constraint solvers in that case.

Finally, the theory applies to our introductory example. We can rewrite `env = 1` ⇒ `code = 1` as a (dual) conditional of type *h*: `env ≥ 1` ⇒ `0 ≤ code -1`, *i.e.*, as:

`env ≥ 1` ⇒ `code ≥ 1`

Therefore, our single rule enjoys the min-closed property.

4.3.4 *Constraint solving with SAT solvers*

Provided that no addition is involved, a similar conclusion could be reached with the unary representation of integer intervals suggested by Bailleux *et al.* [9]. When translating a constraint into a propositional logic formula for SAT solving, this alternative mapping cleverly encodes a domain of size *n* using a vector of *n*–1 booleans b_i. If *i*–1 refers to the *i*th element of the domain of *x*, we have the correspondance:

$b_i = 1$ if and only if *x* is greater than *i* $\qquad (2 \leq i \leq n)$

The b_i's are then connected by a set of Horn clauses:

$$b_n \Rightarrow \ b_{n–1}$$
$$...$$
$$b_2 \Rightarrow \ b_1$$

rendering the logical entailment $x \geq i \Rightarrow x \geq i–1$.

For domains of size 2, we regain the standard encoding.

For a domain of size 3 such as {–1, 0, 1}, we need 2 booleans:

$x = –1$ is encoded as [0,0], $x = 0$ as [1,0] and $x = 1$ as [1,1].

This way, our single (function-free) introductory rule directly translates into a Horn system, thus confirming that solving can go without developing a full decision tree.

The class property is more general and removes the need for a reformulation. As a side effect, the theory is able to explain why, without going into many experiments, such SAT encodings can be more efficient.

4.4 Application to configuration problems

We'll conclude this paper with an example from the literature on dynamic product configuration. For the sake of brevity, we'll use the simplified version of [10] which describes the possible variants of a hierarchical car model with 2 first-level characteristics (frame and package) and 3 components (engine, *aircond*, *sunroof*), the latter 2 being optional. The product structure of figure 2 can easily be flattened, leading to a problem in 8 unknowns, subject to the configuration rules listed in table 1.

In our modeling, domains are ordered from left to right in their definition statement. For optional components, attributes have a supplementary "not available" value. When a component is not "active" (in the sense it isn't part of the solution), its attributes are assigned the `na` value (any other assignment would be irrelevant).

Frame …	frame ∈ {hatchback, sedan, convertible}
Package …	package ∈ {standard, deluxe, luxury}
Engine [Size …]	engine_size ∈ {small, med, large}
	aircond_opt ∈ {discarder, selected}
Aircond [Type …]	aircond_type ∈ {na, ac1, ac2}
	sunroof_opt ∈ {discarded, selected}
Sunroof [Type … Glass …]	sunroof_type ∈ {na, sr2, sr1}
	sunroof_glass ∈ {na, plain, tinted}

Figure 2. The 8 car characteristics

Table 1. Car configuration rules

A1	Sunroof option included if Package = luxury
A2	Sunroof option included if Package = deluxe
A3	Aircond option included if Package = luxury
A4	Sunroof option excluded if Frame = convertible
C5	Package = standard excludes Frame = convertible
C6	Package = standard excludes Aircond.Type = ac2
C7	Package = luxury excludes Aircond.Type = ac1
C8	Sunroof.Type = sr1 and Aircond.Type = ac2 excludes Sunroof.Glass = tinted
A9	Aircond option included if Sunroof.Type = sr1

Part of the configuration rules are "activity" rules (*A*) for selecting options. The others are "compatibility" rules (*C*).

Table 2. Translation into logic rules

1-2	package ≥ deluxe	⇒	sunroof_opt ≥ selected
3	package ≥ luxury	⇒	aircond_opt ≥ selected
4	frame ≥ convertible	⇒	sunroof_opt ≤ discarded
5	package = standard	⇒	frame ≤ sedan
6	package = standard	⇒	aircond_type ≤ ac1
7	package ≥ luxury	⇒	aircond_type ≥ ac2
8	sunroof_type ≥ sr1 ∧ aircond_type ≥ ac2	⇒	sunroof_glass ≤ plain
9	sunroof_type ≥ sr1	⇒	aircond_opt ≥ selected

The translation of the configuration rules gives 8 logic rules. We end up with adding 6 extra ones in order to keep the "not available" value for discarded options[3]:

10	aircond_opt ≥ selected	⇒	aircond_type ≥ ac1
11	aircond_type ≥ ac1	⇒	aircond_opt ≥ selected

[1] Should the consequent be any type of max-closed constraint, the validity of the conclusion would not be affected.

[2] Of course, the property is lost if min-closed and max-closed constraints are mixed together.

[3] *E.g.*, `aircond_type = na` iff `aircond_opt = discarded`.

```
12  sunroof_opt ≥ selected  ⇒  sunroof_type ≥ sr2
13  sunroof_type ≥ sr2      ⇒  sunroof_opt ≥ selected
14  sunroof_opt ≥ selected  ⇒  sunroof_glass ≥ plain
15  sunroof_glass ≥ plain   ⇒  sunroof_opt ≥ selected
```

Nearly all of the rules are of type *h*. The only exceptions are the conditions 5 and 6 laid down on the standard `package` assignment. Hence, a complete consistency diagnosis can be achieved with at most one variable split, *i.e.*, with two checks. In comparison, the product of the domain sizes would give a gross overestimate of 2916 tries. For this configuration problem, the worst case complexity is consequently quite low.

5 SUMMARY

We have defined a domain language with a logical semantics whose statements are sets of facts (generalized here to conjunctions of integer linear equations and inequalities, *i.e.*, to IP models), and sets of rules (*i.e.*, conjunctions of conditionals).

A rule is just a high-level constraint to satisfy, and should be understandable at the knowledge level. By viewing a constraint as a set of low-level rewrite rules, the authors of [11] follow a diametrically opposed goal (which amounts to describe how to procedurally enforce local consistency at the implementation level). Rather than programming experts, the target users for our declarative modeling language are domain experts. The reductions entailed by the rules on the variables' domains are akin to inferences and are not limited to the usual forward chaining mode.

We have identified a sub-language with a remarkable property and linked it to the theory of max-closed constraints. Knowledge bases expressed in that language can be certified conflict-free in polynomial time, by a propagation engine. Global consistency of the constraint system automatically derives from local consistency. There is no search involved, therefore no concern for scalability. Although severely restricted, this sub-language generalizes Horn propositional logic to finite domains that are non-binary.

On the practical side, we've given an insight into the applicability of this theory to technical engineering problems where maintaining the consistency of a large and possibly dynamic model is central. A similar analysis could also provide some guidance on how to derive efficient encodings for handing the formula to check over to a SAT solver.

REFERENCES

[1] Fromherz M., and Saraswat V., *Model-Based Computing: Using Concurrent Constraint Programming for Modeling and Model Compilation*. In Principles and Practice of Constraint Programming - CP'95 Industrial Session, (Montanari & Rossi eds.), LNCS 976, 629-635, Springer, 1995.

[2] ISO 10303-11 standard. Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: *The EXPRESS language reference manual*, ISO, 1994.

[3] Soininen, T. and Niemelä, I., *Developing a declarative rule language for applications in product configuration*. In Proceedings of the First International Workshop on Practical Aspects of Declarative Languages - PADL'99 (Gupta ed.), LNCS 1551, 305-319, Springer, 1998.

[4] Lobo J., Minker J., and Rajasekar A., *Theory of Disjunctive Logic Programs*. In Computational Logic: Essays in honor of Alan Robinson (Lassez, ed.), MIT Press, 1991.

[5] Balduccini M., Gelfond M. and Nogueira M., Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, **47**:1-2, 183 - 219, 2006.

[6] Cohen J. (guest editor), Special issue on Concepts and Topics in Constraint Programming Languages. *Constraints*, **4**:4, Kluwer, 1999.

[7] Jeavons P. and Cooper M., Tractable Constraints on Ordered Domains, *Artificial Intelligence*, **79**:2, 327-339, 1995.

[8] Chandrasekaran R., *Integer programming problems for which a simple rounding type of algorithm works*. In Progress in Combinatorial Optimization (Pulleblank ed.), 10-106, Academic Press, 1984.

[9] Bailleux O., Boufkhad Y. and Roussel O., *New Encodings of Pseudo-Boolean Constraints into CNF*. In Theory and Applications of Satisfiability Testing - SAT'09, 181-194, Springer, 2009.

[10] Narboni G., *A Tentative Approach for Integrating Sales with Technical Configuration*. In Papers from the 2007 AAAI Workshop on Configuration (O'Sullivan & Orsvarn eds.), TR WS-07-03, 41-44, AAAI Press, 2007.

[11] Apt K. and Monfroy E., Constraint Programming viewed as Rule-based Programming. *Theory and Practice of Logic Programming*, **1**:6, 713-750, 2001.

Protocols for Governance-free Loss-less Well-organized Knowledge Sharing

Philippe MARTIN [1]

Abstract. This article first lists typical problems of knowledge sharing approaches based on static files or semi-independently created ontologies or knowledge bases (KBs). Second, it presents a protocol permitting people to collaboratively build and evaluate a well-organized KB without having to discuss or agree. Third, it introduces extensions of this support to allow a precise collaborative evaluation of information providers and pieces of information. Fourth, it shows how a global virtual KB can be based on individual KBs partially mirroring each other.

1 INTRODUCTION

Ontology repositories – and, more generally, the Semantic Web – are often envisaged as composed of many small static (semi-)formal files (e.g., RDF or RDFa documents) more or less independently developed, hence loosely interconnected and with many implicit redundancies or inconsistencies between them [16] [4]. These relations are difficult to recover manually and automatically. This “static file based approach” – as opposed to a “collaboratively-built well-organized large knowledge base (cbwoKB) server approach” – makes knowledge re-use tasks complex to support and do correctly or efficiently, especially in a collaborative way. Most Semantic Web related research works are intended to support such tasks (ontology creation, retrieval, comparison and merging). However, most often, they lead people to create new files – thus contributing to the problems of knowledge re-use – instead of inserting their knowledge into a cbwoKB. Such a KB may be on a single machine or may be a global virtual KB distributed into various correlated KBs on several Web servers and/or the machines of a peer-to-peer network.

Except for WebKB-2 [13] (webkb.org) - the tool implementing the new techniques described in this article, no other ontology/KB server has an ontology-based protocol permitting and enforcing or encouraging people to interconnect their knowledge into a *cbwoKB, while keeping it well-organized (this means that detected partial redundancies or inconsistencies are prevented or made explicit via relations of specialization, identity and/or correction) and without forcing them to agree on terminology or beliefs*. Indeed, i) this is often but wrongly assumed to be impossible or to involve centralization or domain restrictions, ii) this requires the users to see and write (semi-)formal knowledge representations, and iii) this does not directly re-use already existing ontologies. Furthermore, supporting a cbwoKB also requires proposing and managing a large general ontology (WebKB-2 does so). Other KB servers/editors (e.g., Ontolingua, OntoWeb, Ontosaurus, Freebase, CYC and semantic wiki servers) have no such protocols and i) let every authorized user modify what other ones have entered (this discourages information entering or leads to edit wars), or ii) require all/some users to approve or not changes made in the KB, possibly via a workflow system (this is bothersome for the evaluators, may force them to make arbitrary selections, and this is a bottleneck in information sharing that often discourages information providers). By avoiding these two governance problems and leading to a well organized KBs, such kinds of cbwoKB protocol form a basis for a scalable knowledge sharing, even when multiple communities are involved. Actually, unlike with other approaches, a same cbwoKB can be used by many communities with partially overlapping focus since the KB is organized and can be filtered/queried/browsed by each person according to her needs or according to a community viewpoint. Even if built by many communities a (virtual) cbwoKB is unlikely to be huge since i) redundancies are reduced, ii) “well organized knowledge” (as opposed to data) is difficult to build. However, a cbwoKB can permit to index or relate the content of data-bases. In any case, the bigger and the more organized the cbwoKB, the more information are easier to access and compare. Since building a cbwoKB can partly re-use resources of more classic (i.e., less organized) Semantic Web solutions or database solutions, it can be incrementally built to overcome the limitations of these solutions when they become clear and annoying to the users.

Section 2 presents the knowledge representation model used by the rules of the collaborative “KB editing” protocol of WebKB-2. Section 3 presents them and introduces many yet unpublished ideas. Due to space restrictions and for readability reasons, the model and rules are presented via sentences rather than in a fully formal way. Furthermore, as with most methodological rules, the “completeness” criterion does not apply well to these rules.

Collaborative evaluation of knowledge representations is an extension of collaborative KB editing since, for precision and re-use purpose, evaluations should themselves be knowledge representations. In this article, the collaboration scheme of WebKB-2 is not developed but quickly introduced in the last point of the collaborative “KB editing” protocol.

This protocol is not restricted to a physical KB: Section 4 shows how a global virtual cbwoKB can be composed of correlated cbwoKBs of users or communities of users.

Section 5 concludes and reminds that the presented knowledge sharing approaches are complementary. WebKB-2 has been applied to the collaborative representation of many domains by students (for learning purposes), researchers (for knowledge sharing and evaluation purposes) and, currently, experts in the classification of coral species. Due to space restrictions, no evaluation by these users is reported in this article.

[1] University of La Reunion and adjunct researcher of Griffith University, email: ec@phmartin.info

2 LANGUAGE MODEL FOR THE PROTOCOL

The cbwoKB editing protocol used in WebKB-2 are not tied to any particular knowledge representation (KR) language or inference mechanism (hence, this is not the point of this article and no comparison is made on such mechanisms). They only require that *conflicts between knowledge representations – i.e., partial redundancies or inconsistencies between terms or statements* - are detected by some inference mechanism or by people (hence, the protocol also works with informal pieces of knowledge as long as they can be interrelated by semantic relations). The more conflicts between statements are detected, the more relations between the statements are set between them to solve the conflicts, and hence the more the KB is kept organized and thus exploitable.

The *model for the protocol* – i.e., their view on a KB (whichever KR language it actually uses) – is a set of *objects* which are either *terms* or *statements*. Every object has at least one associated source (creator, believer, interpreter, source file or language) represented by a formal term. A *formal term* is a unique identifier for anything that can be though of, i.e., either a source, a statement or a category. It has a unique meaning which may be made partially/totally explicit by its creator via *definitions* with necessary *and/or* sufficient conditions. An *identifier* may be an URI or, if it is not a creator identifier, may include the identifier of its creator (this is the classic solution to avoid lexical conflicts between terms from various sources). An informal term is one name of one or several objects. Two objects may share one or several names but cannot share identifiers. A *statement* is a sentence that is either formal, semi-formal or informal. It is informal if it cannot be translated into a logic formula, for example because it does not have a formal grammar with an interpretation in some logics. Otherwise, it is formal if it only uses formal terms, and semi-formal if it uses some informal terms. A statement is either a *category definition* or a *belief*. A belief must have a source that is its creator and that believes in it and/or that has represented (and hence interpreted) a statement from some other source. Finally, a *category* is either a type of objects or an *individual* (object). A type (a "class" in OWL) is either a relation type or a concept type. An individual is an instance of a first-order type.

The *KR model* of WebKB-2, its associated notations and its inference mechanism must also be mentioned in this section for illustration purposes. Although graph-based, this model is equivalent to the model of KIF (Knowledge Interchange Format; http://logic.stanford.edu/kif/dpans.html), i.e., it permits to use first order logic plus collections (sets, lists, ...) and contexts (meta-statements that restrict the interpretation of statements). WebKB-2 allows the use of several notations: RDF/XML (an XML format for knowledge using the RDF model), the KIF standard notation and other ones which are here collectively called KRLX. These KRLX languages were specially designed to ease knowledge sharing: they are expressive, intuitive and normalizing, i.e., they guide users to represent things in ways that are automatically comparable. One of them is a formal controlled English named FE. It will be used for the examples along with KIF. These languages can be used for creating assertion/query commands and these commands can be sent to the WebKB-2 server via the HTTP/CGI protocol, from an application or from a WebKB-2 Web form. Other communication interfaces are being implemented: one based on SOAP and one based on OKBC (Open Knowledge Base Connectivity; http://www.ai.sri.com/˜okbc) to query (or be queried by) frame-based tools or servers, e.g., Loom, SRI and the GKB-Editor.

Here are some examples of terms in KRLX that highlights its use (the various kinds of quotes are important) en#"bird" and "bird" refer to the English informal word "bird". wn#bird is a formal term referring to one of the WordNet categories for "bird". Here are examples of statements in FE. u1#u2#"birds fly" is an informal statement from u2 and represented by u1. u1#`any u1#bird is pm#agent of a pm#flight' is a formal statement and definition by u1 of u1#bird as something that necessarily fly (the first quote is a backquote, not a straight quote, to allow the embedding of statements). u1#`every u1#bird is agent of a flight' is a semi-formal statement and belief of u1 that "every u1#bird flies". In KIF [9], these last two statements would respectively be

```
(creator u1 ^(defrelation u1#bird (?b) :=>
  (exists ((?f pm#flight)) (pm#agent ?b ?f))))
```

and

```
(believer u1 ^(forall ((?b u1#bird))
              (exists ((?f flight)) (agent ?b ?f)))).
```

When the creator of an object is not explicitly specified, WebKB-2 exploits its "default creator" related rules and variables to find this creator during the parsing. Similarly, unless already explicitly specified by the creator, WebKB-2 uses the "parsing date" for the creation date of a new object. The creator of a belief is also encouraged to add contextualizing relations on it (at least temporal and spatial relations must be specified).

RDF/XML – the W3C recommended linearisation of RDF – and OWL – the W3C recommended language ontology – are currently not particularly well suited for the cbwoKB editing protocol or, more generally, for the representation or interconnection of expressive statements from different users in a same KB.

- They offer no standard way to associate a believer, creator or interpreter to every object in an RDF/XML file. Since 2003, RDF/XML has no bagID keyword, thus no way to represent contexts and hence believers or beliefs. XML name-space prefixes (e.g., u1:bird), Dublin Core relations and statement reification do not permit to do this. This is likely a temporary only constraint since many RDF-related languages or systems extend RDF in this direction: Notation3 (N3), Sesame, Virtuoso, ...
- RDF and OWL – like almost all description logics – do not permit their users to distinguish definitions from universal quantifications. More precisely, they do not offer a universal quantifier. N3 does (Turtle, the RDF-restricted subset of N3, does not). The distinction is important since, as noted in the documentation of KIF [9], a universally quantified statement (belief) may be false while a definition cannot. A definition may be said to be "neither true nor false" or "always true by definition". A user u1 is perfectly entitled to define u1#cat as a subtype of wn#chair; there is no inconsistency as long as the ways u1#cat is further defined or used respect the constraints associated with wn#chair. A definition may be changed by its creator but then the meaning of the defined term is changed rather than corrected. This distinction is important for a cbwoKB editing protocol since it leads to different conflict resolution strategies: "term cloning" and "lossless correction" (Point 5 and Point 6 of the next section).
- Many natural language sentences are difficult to represent in RDF/XML+OWL or N3+OWL, since they do not yet have various kinds of numerical quantifiers, contexts, collections, modalities, ... (FE has concise syntactic sugar for the different kinds). However, at least N3 might soon be extended.
- Like most formal languages, RDF/XML and N3 do not accept – or have a special syntax for – the use of informal objects instead of formal objects. KRLX does and this permits WebKB-2 to create one specialization/generalization hierarchy catego-

rizing all objects. More precisely, this is an "extended specialization/generalization" hierarchy since in WebKB-2 the classic "generalization" relation between formal objects (logical implication) has been extended to apply to informal objects too.

For its cbwoKB editing protocol, WebKB-2 detects partial redundancies or inconsistencies between objects by exploiting exclusion and extended specialization relations between objects. A statement Y is an *extended specialization* of a statement X (i.e., Y includes the information of X and hence *either contradicts it or makes it redundant*) if X structurally matches a part of Y and if each of the terms in this part of Y is identical or an extended specialization of its counterpart term in X. For example, WebKB-2 can detect that u2#`Tweety can be agent of a flight with duration at least 2.5 hour' (which means "u2 believes that Tweety can fly for at least 2.5 hours") is an extended specialization (and an "extended instantiation") of both u1#`every bird can be agent of a flight' and u1#`2 bird can be agent of a flight'. In KIF, the first of these two statements can be written:

```
(believer u1 '(modality possible
     '(forall ((?b bird))
        (exists ((?f flight)) (agent ?b ?f))))
```

Furthermore, these last two statements can be found to be extended specializations (and redundant with) *respectively* u2#`75% of bird can be agent of a flight' and u2#`at least 1 bird can be agent of a flight'. Similarly, this last graph can be found to be inconsistent with u3#`no bird can be agent of a flight'.

WebKB-2 uses graph matching for detecting *extended* specializations. Other inference mechanisms could be used. This matching takes into account numerical quantifiers and measures, not just existential and universal quantifiers. Apart for this, it is similar to the classic graph matching for a specialization (or conversely, a generalization which is a logical deduction) between positive conjunctive existential formulas (with or without an associated positive context, i.e., a meta-statement that does not restrict its truth domain). This classic graph matching is sound and complete with respect to first-order logic and can be computed with polynomial complexity if the query graph (X in the above description) has no cycle [5]. Apart from this restricted case, graph matching for detecting an extended specialization is not always sound and complete. However, this operation works with language of any complexity (it is not restricted to OWL or FOL) and the results of searches for extended specializations of a query graph are always "relevant".

3 COLLABORATIVE KB EDITING PROTOCOL

The rules of the protocol are intended for each object to be connected to at least another object via relations of specialization/generalization, identity and/or argumentation. These rules also permit a loss-less information integration since they do not force to make knowledge selections. They apply to the addition, modification or removal of an object in the KB, e.g., through a graphical interface or via the parsing of a new command in a new input file. This does not serialize objects in the KB and waiting till the whole input file is parsed would not permit to detect more *partial redundancies or inconsistencies between the objects*. The word "user" is here used as a synonym for "source".

1. *Any user can add and use any object* but *an object may only be modified or removed by its creator*.
2. *Adding, modifying or removing a term* is done by adding, modifying or removing at least one statement (generally, one relation) that uses this term. *A new term can only be added by specializing another term* (e.g., via a definition), except for process types which, for convenience purposes, can also be added via subprocess/superprocess relations. In WebKB-2, every new statement is also automatically categorized into the extended specialization hierarchy. A new informal statement must also be connected via an argumentation relation to an already stored statement. In summary, all objects are manually or automatically inserted in the extended specialization hierarchy and/or the subprocess hierarchy, and thus can be easily searched and compared. However, it is clear that if one user (say, u2) enters a term (say, u2#object) that is implicitly semantically close to another user's term (say, u1#thing) but does not manually relates them or manages to give u2#object a definition that is not automatically comparable to the definition of u1#thing (i.e., there is no partial redundancies between the two definition) then the two terms cannot be automatically related by the system and the implicit redundancy cannot be rejected by the system. Here, the problem is that u2 has not respected the following "best practice" rule (which is part of WebKB-2 normalization rules): "always relate a term to all existing terms in the KB via the most important or common relations: i) transitive relations, especially (extended) specialization/generalization relations and mereological relations (to specify parts, containers, ...), ii) exclusion/correction relations (especially via subtype partitions), iii) instance/type relations, iii) basic relations from/to processes, iv) contextualizing relations (spatial, temporal, modal,) and v) argumentation relations".
3. *If adding, modifying or removing a* ***statement*** *introduces an* ***implicit redundancy*** *(detected by the system) in the shared KB,* ***or*** *if this introduces a detected* ***inconsistency between statements believed by the user*** *having done this action*, this ***action is rejected***. Thus, in case of an addition, the user must refine his statement before trying to add it again or he must first modify at least one of his already entered statements. An "implicit" redundancy is a redundancy between two statements without a relation between them making the redundancy explicit. Such a relation is typically an equivalence relation in case of total redundancy and an extended specialization relation (e.g., an "example" relation) in case of partial redundancy. As illustrated in the previous section, the detection of implicit extended specializations between two objects reveals an inconsistency or a total/partial redundancy. It is often not necessary to distinguish between these two cases to reject the newly entered object. Extended "instantiations" (one example was given in the previous section) are exceptions: they do not reveal an inconsistency or a total/partial redundancy that needs to be made explicit, since adding an instantiation is giving an example for a more general statement. It is important to reject an action introducing a redundancy instead of silently ignoring it because this often permits the author of the action to detect a mistake, a bad interpretation or a lack of precision (on his part or not). At the very least, this reminds the users that they should check what has already been represented on a subject before adding something on this subject.
4. *If the addition of* ***a new term*** u1#T by a user u1 *introduces an* ***inconsistency with statements of other users***, this ***action is rejected by the system***. Indeed, such a conflict reveals that u1 has directly or indirectly used at least one term from another user in his definition of u1#T and has misunderstood the meaning of this term. The *addition by a user u2 of a definition to u1#T is actually*

a belief of u2 about the meaning of u1#T. This belief should be *rejected if it is found (logically) inconsistent with the definition(s) of u1#T by u1*. An example is given in Point 6.

5. *If the addition, modification or removal of a statement* ***defining an already existing term u1#T*** by a user u1 *introduces an* ***inconsistency involving statements directly or indirectly re-using u1#T and created or believed by other users*** (i.e., users different from u1), ***u1#T is automatically cloned*** to solve this conflict and ensure that the original interpretation of u1#T by these other users is still represented. Indeed, such a conflict reveals that these other users had a more general interpretation of u1#T than u1 had or now has. Assuming that u2 is this other user or one of these other users, the term cloning of u1#T consists in creating u2#T with the same definitions as u1#T except for one, and then replacing u1#T by u2#T in the statements of u2. The difficulty is to chose a relevant definition to remove for the overall change of the KB to be minimal. In the case of term removal by u1, term cloning simply means changing the creator's identifier in this term to the identifier of one of the other users (if this generated term already exists, some suffix can be added). In a cbwoKB server, since statements point to the terms they use, changing an identifier does not require changing the statements. In a global virtual cbwoKB distributed on several servers, identifier changes in one server need to be replicated to other servers using this identifier. Manual term cloning is also used in knowledge integrations that are not loss-less [6].

   In a cbwoKB, it is not true that beliefs and term definitions "have to be updated sooner or later". Indeed, in a cbwoKB, every belief must be contextualized in space and time, as in

   ```
   u3#` `75% of bird can be agent of a flight' in
       place France and in period 2005 to 2006'
   ```

   (such contexts are not shown in the other examples of this article). If needed, u3 can create the formal term u3#75%_of_birds_fly__in_France_from_2005_to_2006 to refer to this last belief. Due to the possibility of contextualizing beliefs, it is rarely necessary to create formal terms such as u2#Sydney_in_2010. Most common formal terms, e.g., u3#bird and wordnet1.7#bird never need to be modified by their creators. They are speciali-zations of more general formal terms, e.g., wn#bird (the fuzzy concept of bird shared by all versions of the WordNet ontologies). What certainly evolutes in time is the popularity of a belief or the popularity of the association between an informal term and a concept. If needed, this changing popularity can be represented by different statements contextualized in time and space.

6. *If adding, modifying or removing a* ***belief*** *introduces an* ***implicit potential conflict (partial/total inconsistency or redundancy) involving beliefs created by other creators***, it is ***rejected***. However, a user may still represent his belief (say, b1) – and thus "loss-less correct" another user's belief that he does not believe in (say, b2) – by connecting b1 to b2 via a ***corrective relation***. E.g., here are two FE statements by u2, each of which corrects a statement made earlier by u1:

   ```
   u2#` u1#`every bird is agent of a flight' has for
         corrective_restriction u2#`most healthy
          flying_bird can be agent of a flight'  ' and
   u2#` u1#`every bird can be agent of a flight' has
         for corrective_generalization
         u2#`75% of bird can be agent of a flight'  '.
   ```

   In the second case, u2's belief generalizes u1's belief and corrects it since otherwise u2 would not have needed to add it. In the first case, u2's belief specializes u1's belief (except for a quantifier which is generalized) and corrects it. In both cases, WebKB-2 detects the conflict by simple graph-matching.

   If instead of the *belief* 'every bird can be agent of a flight' (all birds can fly), u1 entered the *definition* 'any bird can be agent of a flight', i.e., if he gave a *definition* to the type named "bird", there are two cases (as implied by the rules of the two previous points):

   - u1 originally created this type (u1#bird); then, u2's attempt to correct the definition is rejected, or
   - u1 added a definition to another source's type, say wn#bird since this type from WordNet has no associated constraint preventing the adding of such a definition; then i) the types u1#bird and u2#bird are automatically created as clones (and subtypes of) wn#bird, ii) the definition of u1 is automatically changed into `any u1#bird is agent of a flight', and iii) the belief of u2 is automatically changed into u2#`75% of u2#bird can be agent of a flight'.

   In WebKB-2, users are encouraged to provide argumentation relations on corrective relations, i.e., a meta-statement using argument/objection relations on the statement using the corrective relation. However, to normalize the shared KB, they are encouraged not to use an objection relation but a "corrective relation with argument relations on them". Thus, not only the objections are stated but a correction is given and may be agreed to by several persons, including the author of the corrected statement (who may then remove it). Even more importantly, unlike objection relations, most corrective relations are transitive relations and hence their use permits better organization of argumentation structures, thus avoiding redundancies and easing information retrieval. The use of corrective relations makes explicit the disagreement of one user with (his interpretation of) the belief of another user. Technically, there is no inconsistency: an assertion A may be inconsistent with an assertion B but a belief that "A is a correction of B" is technically consistent with a belief in B. Thus, the shared KB can remain consistent.

   For problem-solving purposes, application-dependent choices between contradictory beliefs often have to be made. To make them, an application designer can exploit i) the statements describing or evaluating the creators of the beliefs, ii) the corrective/argumentation and specialization relations between the beliefs, and more generally, iii) their evaluations via meta-statements (see Point 7). For example, an application designer may choose to select only the most specialized or restricted beliefs of knowledge providers having worked for more than 10 years in a certain domain. Thus, this approach is unrelated to defeasible logics: it does not solve the problems of performing problem-solving-like inferencing with imprecise knowledge and thus contradictory statements but it leads to more precise knowledge and permits to delay the choices between contradictory beliefs until they can be made, that is, in the context of applications (meanwhile, as in WebKB-2, graph-matching based mechanisms can be used to perform semantic search and checks without being affected by contradictions between beliefs from *different* sources).

   The approach also avoids the problems associated with classic "version management" (furthermore, as above explained, in a cbwoKB, formal objects do not have to evolve in time).

   This approach assumes that all beliefs can be argued against and hence be "corrected". This is true only in a certain sense. Indeed, among beliefs, one can distinguish "observations", "interpretations" ("deductions" or "assumptions"; in this approach, axioms are considered to be definitions) and "preferences"; although all these kinds of beliefs can be false (their authors can lie, make a mistake or assume a wrong fact), most people would be reluctant to argue against self-referencing beliefs such as `u2#"u2 likes flowers"` and `u2#"u2 is writing this sentence"`. Instead of trying to formalize this into exceptions, the editing protocols of WebKB-2 rely on the reluctance of people to argue against such beliefs that should not be argued against.

7. To support more knowledge filtering or decision making possibilities and lead the users to be careful and precise in their contributions, a cbwoKB server should propose "default measures" deriving a global evaluation of each statement/creator from i) users' individual evaluations of these objects, and ii) global evaluations of these users. These measures should not be hard-coded but explicitly represented (and hence be executable) to let each user adapt them – i.e., combine their basic functions – according to his goals or preferences. Indeed, only the user knows the criteria (e.g., originality, popularity, acceptance, ..., number of arguments without objections on them) and weighting schemes that suit him. Then, since the results of these evaluations are also statements, they can be exploited by queries on the objects and/or their creators. Furthermore, before browsing or querying the cbwoKB, a user should be given the opportunity to set "filters for certain objects not to be displayed (or be displayed only in small fonts)". These filters may set conditions on statements about these objects or on the creators of these objects. They are automatically executed queries over the results of queries. In WebKB-2, filtering is based on a search for extended specialization, as for conceptual querying. Filters are useful when the user is overwhelmed by information in an insufficiently organized part of the KB. The KB server Co4 [7] had protocols based on peer-reviewing for finding consensual knowledge; the result was a hierarchy of KBs, the uppermost ones containing the most consensual knowledge while the lowermost ones were the private KBs of contributing users. Establishing "how consensual a belief is" is more flexible in a cbwoKB: i) each user can design his own global measure for what it means to be consensual, and ii) KBs of consensual knowledge need not be generated.

The approach described in the above points is incremental and *works on semi-formal KBs*. Indeed, the users can set corrective or specialization relations between objects even when WebKB-2 cannot detect an inconsistency or redundancy. As noted, a new informal statement must be connected via an argumentation relation (e.g., a corrective relation) or an extended specialization relation to an already stored statement. For this relation to be correct, this new statement should generally not be composed of several sub-statements. However, allowing the storing of (small) paragraphs within a statement eases the incremental transformation of informal knowledge into (semi-)formal knowledge and allows doing so only when needed. This is necessary for the general acceptance of the approach. The techniques described in this article work do not seem particularly difficult for information technology amateurs, since the minimum they require is for the users to set the above mentioned relations from/to each term or statement. Hence, these techniques could be used in semantic wikis to avoid their governance problems cited in the introduction and other problems caused by their lack of structure. More generally, the presented approach removes or reduces the file-based approach problems listed in the previous section, without creating new problems. Its use would allow merging of (the information discussed or provided by the members of) many communities with similar interests, e.g., the numerous different communities working on the Semantic Web.

The hypothesis of this approach are that i) conflicts can always be solved by adding more precision (e.g., by making their sources explicit: different "observations", "interpretations" or "preferences"), ii) solving conflicts in a loss-less way most often increases or maintain the precision and organization of the KB, and iii) different, internally consistent, ontologies do not have to be structurally modified to be integrated (strongly inter-related) into a unique consistent semantic network. None of the various kinds of integrations or mappings of ontologies that I made invalidated these hypothesis.

4 DISTRIBUTION IN A VIRTUAL KB

One cbwoKB server cannot support knowledge sharing for all communities. For scalability purposes, the cbwoKB servers of communities or persons should be able to interact to act as one global virtual cbwoKB (gv_cbwoKB), without a central brokering system, without restrictions on the content of each KB, and without necessarily asking each server to register to a particular super-community or peer-to-peer (P2P) network. For several cbwoKB servers to be seen as a gv_cbwoKB, it should not matter which KB a user or agent chooses to query or update first. Hence, object additions/updates made in one KB should be replicated into all the other KBs that have a scope which covers these objects; idem for queries when this is relevant.

Given these specifications, current approaches for collaboration between KB servers/owners (e.g., [4] [14] which are based on integrating changes made in other KBs, and [15] which also use a workflow system) or distributed querying between a few KB servers (e.g., as described by [12]) are insufficient. Indeed, they are based on partial descriptions of the content of each KB or on predefined roles for each KB owner or user, and the redundancies or inconsistencies between the KBs are not made explicit. This often makes difficult to find the relevant KBs to search or add in and to integrate query results.

As in the previous sections, a solution is to let the knowledge indexation and distribution be made at the object level instead of the document/KB/community/owner level. The requirement is that for every term T stored in a cbwoKB server, the KB must ***either***

- have a ***Web-accessible formal description specifying that it is committed to be a "nexus" for T***, i.e., that i) it stores any statement S on T (if S is inserted in another KB of this gv_cbwoKB, it is also inserted in this KB), or ii) it associates to T the URLs of cbwoKB servers permitting to find or store any statement on T, ***or***
- not be a "nexus" for T and hence associate to T either i) the URLs of all cbwoKB servers that have advertised themselves to be a nexus for T, or ii) the URL of at least one server that stores these URLs of nexus servers for T.

Thus, via forwards between servers, all objects using T can be added or found in all the nexus for T. This requirement refines the 4th rule of the Linked Data approach [3]: "link things to their related ones in some other data sets". Indeed, to obtain a gv_cbwoKB, the data sets must be cbwoKB servers and there must be at least one nexus for each term. A consequence is that when the scopes of two nexus overlap, they share common knowledge and there is no *implicit* redundancies or inconsistencies between them. Thus, the gv_cbwoKB has a unique ontology distributed on the various cbwoKB servers.

The difficult task is, whenever the owners of a new cbwoKB server want to join a gv_cbwoKB, to integrate their ontology into the global one (they must find some nexus of the gv_cbwoKB, only one if it has a nexus for its top level type). This integration task is at the core of most knowledge sharing/re-use approaches. In this one, it is done only by the owners of the new cbwoKB; once this is done, regularly and (semi-)automatically integrating new knowledge from/to other nexus is much easier since a common ontology is shared. Thus, it can be envisaged that one initial cbwoKB server be progressively joined by other ones to form a more and more general gv_cbwoKB.

The key point of the approach is the formal commitment to be a nexus for a term (and hence to be a cbwoKB since direct searches/additions by people must be allowed). There is currently no standard vocabulary to specify this, e.g., from the W3C, the Dublin Core and voiD [10] (a vocabulary for discovering linked datasets). It is in the interest of a competitive company to advertise that it hosts a nexus for a certain term, e.g., `apartment_for_rent_in_Sydney` for a real estate agent covering the whole of Sydney. If the actual coverage of a nexus is less than the advertised one, a competitor may publish this. In a business environment, it is in the interest of a competitive company to check what its competitors or related companies offer and, if it is legal, integrate their public information in its cbwoKB. It is also in its interest to refer to the most comprehensive KBs/nexus of its related companies. To sum up, the approach could be technically and socially adopted. Since its result is a gv_cbwoKB, it can be seen as a way to combine advantages commonly attributed to "distributed approaches" and "centralized approaches".

5 CONCLUSION

This article first aimed to show that a (gv_)cbwoKB is *technically and socially possible*. To that end, Section 4 presented a protocol permitting, enforcing or encouraging people to *incrementally* interconnect their knowledge into a well-organized (formal or *semi-formal*) KB without having to discuss and agree on terminology or beliefs. As noted, it seems that all other knowledge-based cooperation protocols that currently exists work on the comparison or integration of whole KBs, not on the comparison and loss-less integration of all their objects into a same KB. Other required elements for a (gv_)cbwoKB – and for which WebKB-2 implements research results - were also introduced (Section 5 and Section 6) or simply mentioned: expressive and normalizing notations, methodological guidance, a large general ontology, and an initial cbwoKB core for the application domain of the intended cbwoKB.

Already explored kinds of applications were cited. One currently explored is the collaborative representation and classification by Semantic Web experts of "Semantic Web related techniques". This means that in the medium term Semantic Web researchers will be able and invited to represent and compare their techniques in WebKB-2, instead of just indexing their research via domain related terms, as was the case in the KA(2) project [2] or with the Semantic Web Topics Ontology [1]. More generally, the approach proposed in this article seems interesting for collaboratively-built corporate memories or catalogues, e-learning, e-government, e-science, e-research, etc. [11] describes a "Knowledge Web" to which teachers and researchers could add "isolated ideas" and "single explanations" at the right place, and suggests that this Knowledge Web could and should "include the mechanisms for credit assignment, usage tracking and annotation that the Web lacks" (pp. 4-5). [11] does not give indications on what such mechanisms could be. The cbwoKB elements described by this article can be seen as a basis for such mechanisms.

A second aim of this article (mainly via Section 2) was to show that – in the long term or when creating a *new* KB for *general* knowledge sharing purposes – using a cbwoKB does/can provide more possibilities, with *on the whole* no more costs, than the mainstream approach [16] [3] where knowledge creation and re-use involves searching, merging and creating (semi-)independent (relatively small) ontologies or semi-formal documents. The problem – and related debate – is more social than technical: which formalisms and user-centric methodologies will people accept to learn and use to gain precision and flexibility at the expense of more initial involvement? The answers depend on the considered kinds of users and time frames (now or in the long term future). A cbwoKB is much more likely to be adopted by a small communities of researchers but could incrementally grow to a larger and larger community. In any case, research on the two approaches are complementary: i) techniques of knowledge extraction or merging ease the creation of a cbwoKB, ii) the results of applying these techniques with a cbwoKB as input would be better, and iii) these results would be easier to retrieve, compare, combine and re-use if they were stored in a cbwoKB.

REFERENCES

[1] ISWC 2006. Owl specification of the semantic web topics ontology. http://lsdis.cs.uga.edu/library/resources/ontologies/swtopics.owl, 2006.

[2] V.R. Benjamins, D. Fensel, A. Gomez-Perez, D. Decker, M. Erdmann, E. Motta, and M. Musen. Knowledge annotation initiative of the knowledge acquisition community: (ka)2. KAW 1998, 11th Knowledge Acquisition for Knowledge Based System Workshop, April 18-23, 1998.

[3] C. Bizer, T. Heath, and T. Berners-Lee, 'Linked data - the story so far', *International Journal on Semantic Web and Information Systems*, **5 (3)**, 1–22, (2010).

[4] P. Casanovas, N. Casellas, C. Tempich, D. Vrandecic, and R. Benjamins, 'Opjk and diligent: ontology modeling in a distributed environment', *Artif. Intell. Law*, **15 (2)**, 171–186, (2007).

[5] M. Chein and M.-L. Mugnier, 'Positive nested conceptual graphs', *ICCS 1997*, **LNAI 1257**, 95–109, (1997).

[6] R. Djedidi and A. Aufaure, 'Define hybrid class resolving disjointness due to subsumption', *Web page*, (2010).

[7] J. Euzenat, 'Corporate memory through cooperative creation of knowledge bases and hyper-documents', *KAW 1996*, (36)1–18, (1996).

[8] J. Euzenat, O. Mbanefo, and A. Sharma, 'Sharing resources through ontology alignment in a semantic peer-to-peer system', *Cases on semantic interoperability for information systems integration: practice and applications*, 107–126, (2009).

[9] M.R. Genesereth, 'Knowledge interchange format', *Draft proposed American National Standard (dpANS), NCITS.T2/98-004*, (1998).

[10] M. Hausenblas, 'Discovery and usage of linked datasets on the web of data', *Nodalities Magazine*, **4**, 16–17, (2008).

[11] W.D. Hillis, 'Aristotle (the knowledge web)', *Edge Foundation, Inc.*, **138**, (May 6, 2004).

[12] J. Lee, J. Park, Park M., C. Chung, and J. Min, 'An intelligent query processing for distributed ontologies', *Systems and Software*, **83 (1)**, 85–95, (Jan. 2010).

[13] Ph. Martin and M. Eboueya, 'For the ultimate accessibility and reusability', 589–606, (July 2008).

[14] N.F. Noy and T. Tudorache, 'Collaborative ontology development on the (semantic) web', *AAAI Spring Symposium on Semantic Web and Knowledge Engineering (SWKE)*, (March 2008).

[15] R. Palma, P. Haase, Y. Wang, and M. d'Aquin. Propagation models and strategies. Deliverable 1.3.1 of NeOn (Lifecycle Support for Networked Ontologies; NEON EU-IST-2005-027595), Jan. 2008.

[16] N. Shadbolt, T. Berners-Lee, and W. Hall, 'The semantic web revisited', *IEEE Intelligent Systems*, **21 (3)**, 96–101, (May/June 2006).

[17] J. Sowa. Theories, models, reasoning, language, and truth. Web document http://www.jfsowa.com/logic/theories.htm, Dec. 2005.

Knowledge-based Implementation of Metalayers - The Reasoning-Driven Architecture

Lothar Hotz and **Stephanie von Riegen**[1]

Abstract. The *Meta Principle*, as it is considered in this paper, relays on the observation that some knowledge engineering problems can be solved by introducing several layers of descriptions. In this paper, a knowledge-based implementation of such layers is presented, where on each layer a knowledge-based system consisting, as usual, of a knowledge model and separated inference methods is used for reasoning about the layer below it. Each layer represents and infers *about* knowledge located on a layer below it.

1 Introduction

Typically, knowledge engineering has the goal to create a model of knowledge of a certain domain like car periphery supervision [33], drive systems [29], or scene interpretation [16]. For knowledge-based tasks, like constructing or diagnosing a specific car periphery system, a strict separation is made into *domain model* which covers the knowledge of a certain domain and a *system model* which covers the knowledge of a concrete system or product of the domain. The domain model and the system models are represented with a *knowledge-modeling language* which again is interpreted, because of a defined semantic, through a *knowledge-based system*. Examples are a terminology box (TBox) as a domain model representing e.g. knowledge about an animal domain; and an assertional box (ABox) as a system model representing e.g. a specific animal. The TBox and ABox are represented with a certain Description Logic [5] which is again interpreted through a Description Logic reasoner like PELLET or RACER. In this paper, we use configuration systems (*configurators*) as knowledge-based systems. In such systems, a domain model (also called *configuration model*) is used for constructing a system model (also called *configuration*). Configurators typically combine a broad range of inference mechanisms like constraint solving, rule-based inference, taxonomical reasoning, and search mechanisms.

The domain model and the system model constitute two layers: In the domain model all knowledge *about* possible systems is expressed, in the system model all knowledge *about* one real system is expressed. Often, these two layers are sufficient for expressing the knowledge needed for a certain knowledge-based task. However, in some domains more layers may be needed for adequately representing the domain knowledge.

We take the biological classification as an example for multiple layers. In [32], a detailed discussion of alternative modeling for biological classification is supplied. Figure 1 presents an extract of the traditional biological classification of organisms established by Carolus Linnaeus. Each biological rank joins organisms according to shared physical and genetic characteristics. We conceive of a rank as knowledge about the next layer. The main ranks are kingdom, phylum, class, order, genus, species, and breed, which again are divided into categories. Each category unifies organisms with certain characteristics. For instance, the `Mammal` class includes organisms with glands only, thus a downward specialisation from `Mammal` to its subcategories is depicted in Figure 1. For clarity reasons, only extracts of ranks and categories are given, for example the rank of kingdom contains more than the category animal, among others plants, bacteria, and fungi. The ranks are representing an additional layer ($BioCl^M$) above the domain model of the biological classification. The categories of the ranks form the domain model layer ($BioCl^D$) and each of them is an instance of the correspondent rank. The system model layer ($BioCl^S$) is covering specific individuals (also called *domain objects*), e.g. Tux the penguin. By the given classification, the need for multiple layers becomes directly evident: It is understandable that a `King Penguin` is an instance of `Breed`. But it would be improper to denote `Tux` as a `Breed`, which would hold if `King Penguin` would be a specialization of `Breed`.

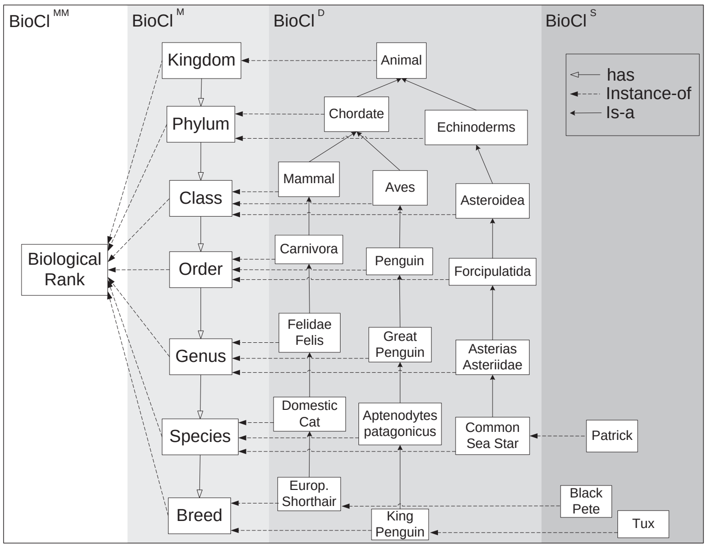


Figure 1. Biological Classification represented with several layers. Figure inspired by [4].

Because each layer specifies knowledge about knowledge in the layer below it, we also speak of a *metalayer approach* or of *metaknowledge* because as [28] pointed out: "Metaknowledge is knowledge about knowledge, rather than knowledge from a specific domain such as mathematics, medicine or geology." From a knowledge engineering viewpoint, metaknowledge is being created at the same time

[1] HITeC e.V. c/o Fachbereich Informatik, Universität Hamburg, Germany, email: {hotz,svriegen}@informatik.uni-hamburg.de

as knowledge [27]. For supporting metaknowledge, representation facilities are needed that allow the adequate representation of these types of knowledge, in here called *metaknowledge models*. A strict separation of these knowledge types from each other and the encapsulation of metalayer facilities are further requirements for a metalayer approach [6]. Furthermore, for facilitating the use and maintenance of the layers, if possible, each layer should be realized with the same modeling facilities. For being able to reason about the models on a metalayer and not only to specify them, a declarative language with a logic-based semantic should be used for implementing each layer. For allowing domain specific models on the layers, each layer should be extensible.

In this paper, we propose a Reasoning-Driven Architecture, RDA (rooted at the Model-Driven Architecture, MDA, see [23] and Section 2). The RDA consists of an arbitrary number of layers. Each layer consists of a model and a knowledge-based system. Both represent the facilities used for reason about the next lower layer.

For this task, we consider the Component Description Language (CDL) as a knowledge-modeling language which was developed for representing configuration knowledge [15]. CDL combines ontology reasoning similar to the Web Ontology Language (OWL) [2] with constraint reasoning [30], and rules [13] (see Section 3). Because knowledge-based configuration can be seen as model construction [8, 16, 15], these technology provides a natural starting point for implementing the modeling layers of RDA. Typically, a configuration model (located at the domain model layer) generically represents concrete configurations which themselves are located on the system model layer. The crucial point of RDA is to provide each layer with a model that represents knowledge about the next lower layer (see Section 4) and uses a knowledge-based configuration system to infer about this layer (see Section 5). By introducing a configuration system on each layer of the RDA, we enable reasoning tasks like consistency check, model construction and enhancement on each layer, i.e. also on metalayers.

An application of such an RDA is naturally to support the knowledge acquisition process needed for knowledge-based systems. In a first phase of a knowledge acquisition process, the typically tacit knowledge about a domain is extracted by applying knowledge elicitation methods and high interaction between a knowledge engineer and the domain expert (*knowledge elicitation phase*). A model sketch is the result, which in turn is formalized during the *domain representation phase*. During this phase a domain model is created. The domain model has to be expressed with the facilities of a knowledge-modeling language. The RDA can e.g. be used to check such knowledge models for being consistent with the knowledge-modeling language.

2 Reasoning-Driven Architecture

For defining the Reasoning-Driven Architecture (RDA), we borrow the notion of layers from the Model-Driven Architecture [24, 22, 12, 25]. In MDA, the main task is to specify modeling facilities that can be used for defining models (*metamodeling*), see for example [25]: "A metamodel is a model that defines the language for expressing a model". Or compiled to terms used here: "A metaknowledge model (called *Meta-CDL-KB*, see below and Section 3) is a knowledge model that defines CDL, which in turn is used for expressing a domain model". MDA provides four layers for modeling (see Figure 2, *MDA view*): $M2$ is the language layer, which is realized by (or *is a (linguistic, s.b.) instance-of*) a metamodel located on the $M3$ layer. The language is used for creating a model of a specific system on the $M1$ layer. The system model represents a system which is located in the reality ($M0$ layer; not shown in the figure for brevity) [9]. Please note, that each layer contains elements which are instances of classes of the layer above. Typically a specific implementation in a tool ensures that a system model on $M1$ conforms to a model on $M2$ and a model on $M2$ conforms to a metamodel on $M3$.

For clarifying our approach, we will use $R1$, $R2$, $R3$ for RDA which roughly correspond to $M1$, $M2$, $M3$ in MDA, respectively. Ri stands for "Reasoning Layer i". We separate $R1$ in several reasoning layers, because for knowledge-based systems one single model on this layer is not sufficient. This is due to the above mentioned separation of domain model and system model. $R1$ consists of a domain model specified with concepts and constraints of CDL (denoted by $R1_C$) and knowledge instances (denoted $R1_I$) representing the system model. Furthermore, corresponding reasoning facilities of CDL allow to reason about entities on layer $R1$ (see Figure 2, *CDL view*).

The RDA itself consists of multiple copies of the CDL view for representing and reason about distinct types of metaknowledge on different layers. These layers are denoted with $R1^{Mi}$ ($i \geq 0$), $R1^D$, $R1^S$ for an arbitrary number of metamodel layers, one domain model layer, and one system model layer, respectively. Because each of these layers are realized with same knowledge-based facilities, i.e. CDL concepts and instances, we do not extend CDL with the notion of a metaclass, which has instances that act as classes and can again have instances (e.g. like OWL Full [2] or like MDA/UML implementations with stereotypes [4]). Thus, the layers of $R1$ are not in one system but are clearly separated into several systems, here called *Knowledge Reflection Servers*. Each server is realized with the typical concept/instance scheme. Hence, each server can be realized with a typical knowledge-based system like Description Logic systems, rule-based systems, or as in our case a configuration system based on CDL. Through a mapping between those servers, concepts on one layer are identified with instances of the next higher layer. This mapping is a one-to-one mapping and based on a metaknowledge model (see Figure 2, *RDA view* and Section 4).

Following [4], we distinguish between a *linguistic* and an *ontological instance-of* relation. However, we explicitly name the internal *implementation instance-of* relation as such, which is a simple UML type-instance relation in [4]. The implementation instance-of relation is provided through the instantiation protocol of the underlying implementation language Common Lisp and its Common Lisp Object System (CLOS) [20, 21] (see Figure 2, classes are instances of the predefined metaclass `standard-class`). The linguistic instance-of relation is originated in the notion of classes and objects known from programming languages. In case of CDL, this relation is realized with the macroexpansion facilities of Common Lisp. Beside others, Figure 2 depicts concept definitions (`define-concept`) of CDL and their linguistic instance-of expansion to `defclass` of CLOS. The ontological instance-of relation represents relationships between knowledge elements, in CDL between concepts and instances (see above).

As we will see in Section 3, the main feature of CDL is given by the use of its inference techniques like constraint propagation. By representing the knowledge of a domain with modeling facilities of CDL these inference techniques can be applied for model construction. This representation is basically a generic description of domain objects of a domain at hand, i.e. CDL is used for specifying a domain model. For the representation of concrete domain objects, this description is instantiated and a system model is constructed. The created instances are related to each other through relations. Furthermore, instances can be checked for concept membership.

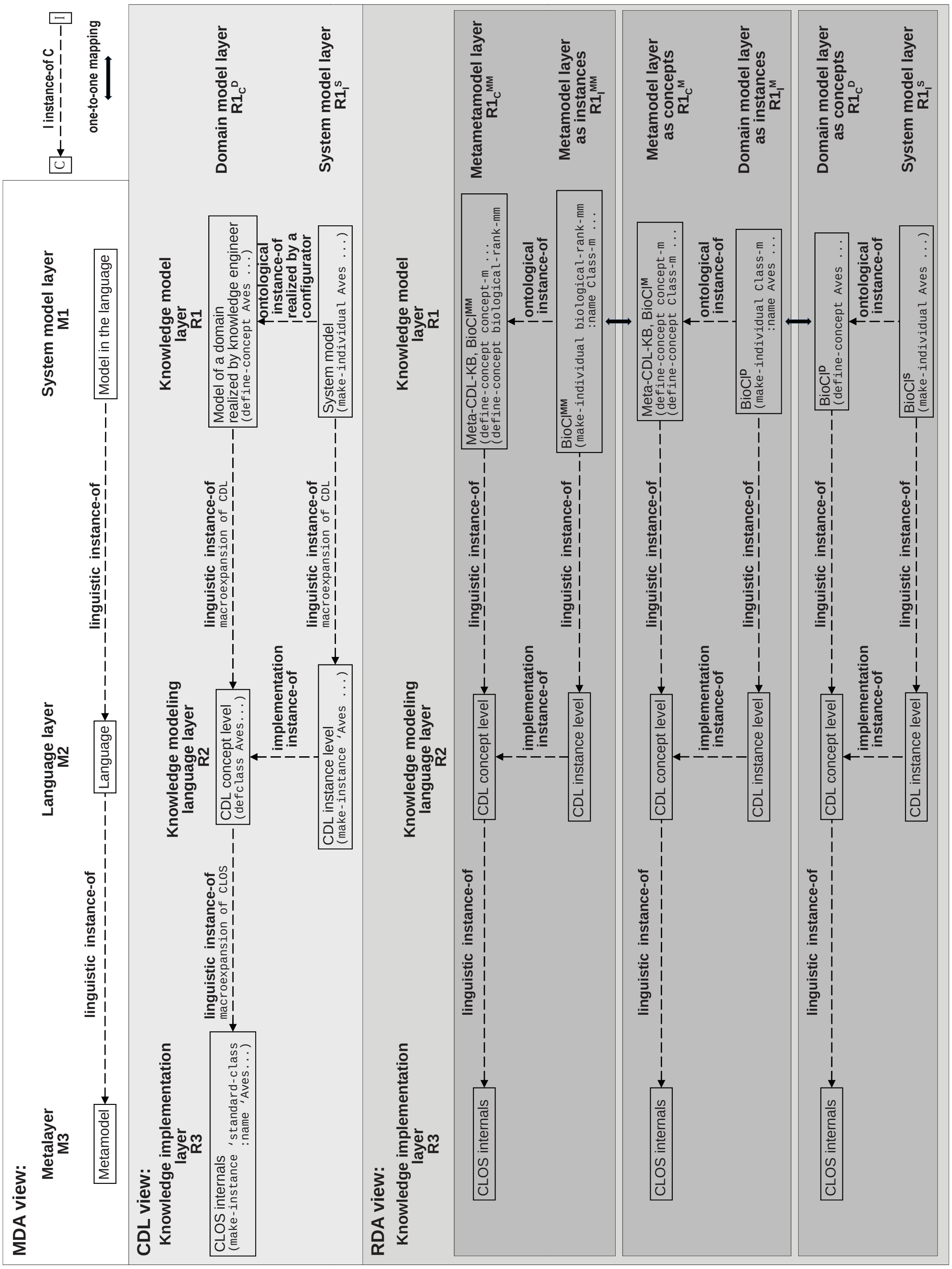

MDA view:
Metalayer M3
Language layer M2
System model layer M1
Metamodel
linguistic instance-of
Language
linguistic instance-of
Model in the language
C
I instance-of C
I
one-to-one mapping
CDL view:
Knowledge implementation layer R3
Knowledge modeling language layer R2
Knowledge model layer R1
Domain model layer R1C^D
System model layer R1I^S
CLOS internals (make-instance 'standard-class :name 'Aves...)
linguistic instance-of macroexpansion of CLOS
CDL concept level (defclass Aves...)
implementation instance-of
CDL instance level (make-instance 'Aves ...)
linguistic instance-of macroexpansion of CDL
linguistic instance-of macroexpansion of CDL
Model of a domain realized by knowledge engineer (define-concept Aves ...)
ontological instance-of realized by a configurator
System model (make-individual Aves ...)
RDA view:
Knowledge implementation layer R3
Knowledge modeling language layer R2
Knowledge model layer R1
Metametamodel layer R1C^MM
Metamodel layer as instances R1I^MM
Metamodel layer as concepts R1C^M
Domain model layer as instances R1I^M
Domain model layer as concepts R1C^D
System model layer R1I^S
CLOS internals
linguistic instance-of
CDL concept level
implementation instance-of
CDL instance level
linguistic instance-of
Meta-CDL-KB, BioCl^MM (define-concept concept-m ... (define-concept biological-rank-mm
ontological instance-of
BioCl^MM (make-individual biological-rank-mm :name Class-m ...
Meta-CDL-KB, BioCl^M (define-concept concept-m (define-concept Class-m ...
BioCl^D (make-individual Class-m :name Aves ...
BioCl^D (define-concept Aves ...
BioCl^S (make-individual Aves ...)

Figure 2. Comparing MDA, CDL, and RDA. The MDA view is refined when applying CDL with its domain and system model on $R1$ ($R1_C$, $R1_I$). The CDL view is three times copied for using RDA for the biological classification domain.

A configuration system supports mainly three tasks (see Figure 2, *CDL view*):

1. It enables the expression of a configuration model that is consistent with the configuration language, which the system implements. For this task, the configuration system performs consistency checks of given configuration models (or parts of it) with the language specification. As a result, the configuration model is conform to the configuration language. However, the creation of the configuration model is done manually during knowledge acquisition.
2. On the basis of the configuration model, the configuration system supports the creation of configurations (system models) that are consistent with the configuration model. For this task, the system interprets the logical expressions of the configuration model and creates configurations according to these definitions. As a result,

the configurations are conform to the configuration model.

3. The configuration model can be defined in textual form or supported by a graphical user interface that enables the creation of concepts and constraints. Thus, the configuration system supplies a user interface for expressing the configuration model and for guiding the configuration process.

Thus, a configuration system supplies means for supporting the step from a domain model ($R1_C^D$) to a system model ($R1_I^S$) (see Figure 2, *CDL view*) and it can check configuration models represented with the configuration language of the system for being compliant with the language. However, the development of the configuration model is not supported with general inference techniques but is system dependent.

In RDA, this instantiation facility is used for supporting the step from the configuration language to the domain model. By applying the configuration system to a domain model that contains every model of a language, i.e. by applying it to a *metaknowledge model* (the Meta-CDL-KB), the configuration of a domain model for any specific domain is supported (Figure 2, *RDA view*, $R1_C^M$). This is achieved because of the general applicability of the language constructs of CDL, which are based on logic (see Section 3). Furthermore, other advantages of configuration systems, like a declarative representation of the configuration model, or the use of inference techniques can be applied to the Meta-CDL-KB. Thus, the construction of metamodel-compliant domain models is supported with this approach. However, the question arises: How can CDL be represented with CDL? (see Section 4.1).

3 Comprehensive Knowledge Representation

This section is organized as follows. First a brief overview of the knowledge representation language CDL (Component Description Language) [15] will be given in Section 3.1. Section 3.2 presents parts of the metamodel of CDL which have to be modeled on $R1^M$. CDL will be used in the following sections to realize the envisioned metalevels through knowledge-based implementations.

3.1 A Sketch of CDL

The CDL mainly consists of two modeling facilities: a concept hierarchy and constraints. Models consisting of concepts and constraints belong to $R1^D$, for the biological classification domain this layer is named $BioCl^D$ (see Figure 1).

The **Concept Hierarchy** contains *concepts*, which represent domain objects, a *specialization hierarchy* (based on the `is-a` relation), and *structural relations*. Concepts gather all properties, a certain set of domain objects has, under a unique name. A specialization relation relates a *super-concept* to a *sub-concept*, where the later inherits the properties of the first. The structural relation is given between a concept c and several other concepts r, which are called *relative concepts*. With structural relations a compositional hierarchy based on the `has-parts` relation can be modeled as well as other structural relationships. For example, the structural relation `has-differences` connects a species with its differentiating characteristics which is not a decomposition, to be precise. Parameters specify the attributes of a domain object with value intervals, values sets (enumerations), or constant values. Parameters and structural relations of a concept are also referred to as *properties* of the concept. *Instances* are instantiations of concepts and represent concrete domain objects. When instantiated, the properties of an instance are initialized by the values or value ranges specified in the concept. Figure 3 gives examples for concept definitions. The structural relation `has-differences` is defined, which relates one biological class with several characteristics and one characteristic with several classes. The concept for a biological class (`Mammal`) is defined with such a relation including number restricted structural relations. The right side of the operator `:>` consists of the super-concept of all relative concepts and the minimal and maximal number of those concepts. The left side restricts the total number of instances in the relation. The characteristics `Hair` and `Fur` are optional and only one of them can be used for describing a mammal, because exactly 6 characteristics are needed for specifying a mammal. Except of the metaconcept specification typical ontological definitions are given.

Constraints summarize conceptual constraints, constraint relations, and constraint instances. *Conceptual constraints* consists of a condition and an action part. The condition part specifies a *structural situation* of instantiated concepts. If this structural situation is fulfilled by some instances (i.e. the instances *match* the structural situation), the *constraint relations* that are formulated in the action part are instantiated to *constraint instances*.

Constraint relations can represent restrictions between properties like `all-different-p` or `ensure-relation`. The constraint relation `ensure-relation` establishes a relation of a given name between two instances. It is used for constructing structural relations and thus provides main facilities for creating resulting constructions. Before establishing a relation between given instances, `ensure-relation` checks whether the relation already exists. The constraint relation `all-different-p` ensures that all objects in a given `set` are of a different type. Please note, that such kind of constraints extend typical constraint technology, which is based on primitive datatypes like numbers or strings [19].

Constraints are multi-directional, i.e. they are propagated regardless of the order in which constraint variables are instantiated or changed. At any given time, the remaining possible values of a constraint variable are given as structural relations, intervals, value sets or constant values.

Constraint relations are used in the action part of conceptual constraints. Figure 6(c) gives also an example of such a conceptual constraint in CDL, however, already on a metalayer. It shows, how instances, which are selected through the structural situation, can be checked for being of a different type. When this check is fulfilled this constraint would be consistent otherwise inconsistent.

A *configuration system* performs knowledge processing on the basis of logical mappings like they are given in [31] for a predecessor of CDL. Thus, the configuration system applies *inference techniques* such as taxonomical reasoning, value-related computations like interval arithmetic [18], establishing structural relations, and constraint propagation. The structural relation is the main mechanism that causes instantiations and thus leads to an extended configuration: If such a relation is given between a concept c and several relative concepts r, depending on what exists first as instances in the configuration (c or one or more of the relative concepts r), instances for the other part of the relation may be created and the configuration increases. This capability together with the fact that descriptions, i.e. models, of systems are constructed lead to the use of configuration systems for constructing models. For a detailed description of CDL and its use in a configuration system, we refer to [15].

a)

```
(define-relation :name has-differences
  :inverse differentiate
  :domain Class-m
  :range Characteristics
  :mapping m-n)

(define-concept :name Animal
  :specialization-of domain-root
  :metaconcept Kingdom-m)

(define-concept :name Chordate
  :specialization-of Animal
  :metaconcept Phylum-m)

(define-concept :name Mammal
  :specialization-of Chordate
  :has-differences
   ((:type Characteristic :min 6 :max 6)
     :>
    (:type Glands :min 1 :max 1)
    (:type Hair :min 0 :max 1)
    (:type Fur :min 0 :max 1)
    (:type MiddleEarBones :min 3 :max 3)
    (:type WarmBlooded :min 1 :max 1))
  :metaconcept Class-m)
```

b)

```
(define-concept :name Characteristic
  :specialization-of domain-root
  :metaconcept Characteristic-m)

(define-concept :name Glands
  :specialization-of Characteristic
  :metaconcept Characteristic-m)

(define-concept :name Hair
  :specialization-of Characteristic
  :metaconcept Characteristic-m)

(define-concept :name Fur
  :specialization-of Characteristic
  :metaconcept Characteristic-m)

(define-concept :name MiddleEarBones
  :specialization-of Characteristic
  :metaconcept Characteristic-m)

(define-concept :name WarmBlooded
  :specialization-of Characteristic
  :metaconcept Characteristic-m)
```

Figure 3. Example of CDL concept definitions from the domain of biological classification.

3.2 Parts of the Metamodel of CDL

Languages are typically defined by describing their abstract syntax, their concrete syntax, and consistency rules. For describing CDL's abstract syntax, we introduce three metalevel facilities: a *knowledge element*, a *taxonomical relation* between knowledge elements, and a *compositional relation* between knowledge elements. These facilities are not to be mixed up with the above mentioned CDL facilities: *concepts*, *specialization relations*, and *structural relations*. The abstract syntax for concepts and conceptual constraints of CDL is given in Figure 5. A concrete syntax for CDL is given in Figure 3 for example.

A CDL concept is represented with a knowledge element of name $concept$ (see Figure 4 (a)), and a CDL structural relation is represented with the knowledge element $relation\text{-}descriptor$. The fact that CDL concepts can have several structural relations is represented with a compositional relation with name $has\text{-}relations$. Parameters are represented similarly.

Structural relations are defined in a further part of the metamodel (see Figure 4(b)). The fact that a concept is related by a structural relation of other concepts (the relative concepts) is represented with three knowledge elements and three compositional relations in a cyclic manner.

Figure 4(c) provides the metamodel for a conceptual constraint with its structural situation and action part. A structural situation consists of a concept expression which in turn consists of a variable and a conditioned concept. The action part consists of a number of constraint relations that should hold if the structural situation is fulfilled.

Several consistency rules define the meaning of the syntactic constructs. For example, one rule for the structural relation defines that the types of the relative concepts of a structural relation have to be sub-concepts of the concept on the left side of the operator :> ($rule$-5). Additionally, consistency rules are given that check CDL instances, e.g. one rule defines when instances match a conceptual constraint ($rule$-6). All rules are given in [15].

4 Metamodels

In this section, the metamodels needed for $R1^M$ and $R1^{MM}$ (Section 4.1 and [14]) and their extensions for modeling the biological classification domain (Section 4.2) are presented.

4.1 CDL in CDL

As discussed in Section 2, $R1^M$ and $R1^{MM}$ will be realized with CDL. Thus, the goal is to define the metamodel of CDL (as sketched in Section 3.2) using CDL itself. In fact, CDL provides all knowledge representation facilities needed for this purpose. The result of this is a metaconfiguration model called Meta-CDL knowledge base (Meta-CDL-KB). In Section 3.2, parts of the metamodel of CDL are defined using three modeling facilities, namely *knowledge elements*, *taxonomical relation*, and *compositional relation*. These modeling facilities are mapped to the CDL constructs *concept*, *specialization relation*, and *structural relation* respectively. Figure 5 shows how the knowledge elements shown in Figure 4 (a) can be represented with the *meta*-concepts $concept\text{-}m$, $relations\text{-}descriptor\text{-}m$, and $parameter\text{-}m$. The consistency rules of CDL have to be represented also. This is achieved by defining appropriate constraints. In Figure 5, a conceptual constraint is represented, which checks the types of a structural relation.[2]

Furthermore, instances can be represented on the metalevel by including the metaconcept $instance\text{-}m$. Having instances available, conceptual constraints and their matching instances can be represented (see Figure 5). The fact that instances fulfill a certain conceptual constraint is represented through establishing appropriate relations using the constraint relation `ensure-relation`. Please note that self references can be described also, e.g. a *concept-m* is related to itself via the *has-superconcept-m* relation (compare the loop in Figure 4 with Figure 5).

Other approaches also use metalevels for defining their language, e.g. UML [34, 24, 26, 25]. In contrast to these approaches, we use

[2] For a complete mapping of the CDL consistency rules to conceptual constraints see [15].

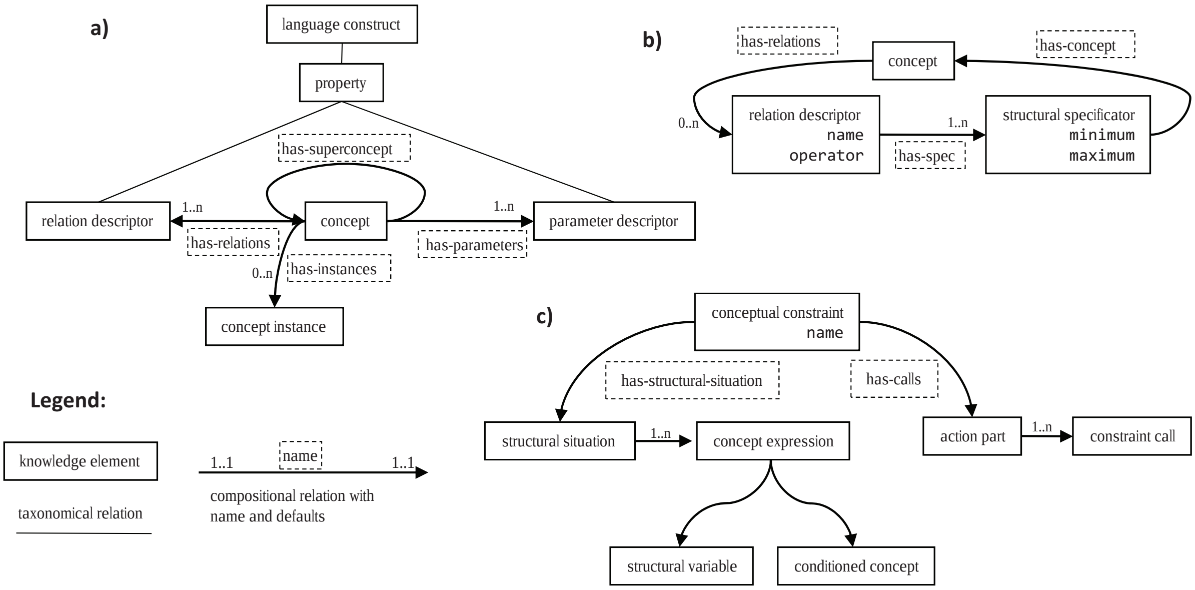


Figure 4. Metamodel for a) a concept, b) a structural relation, and c) a conceptual constraint.

```
(define-concept :name concept-m
  :specialization-of named-domain-object-m
  :concept-of-dom-m (:type domain-m)
  :superconcept-of-m
   (:type concept-m :min 0 :max inf)
  :in-some-m (:type some-m :min 0 :max inf)
  :has-superconcept-m
   (:type concept-m :min 0 :max 1)
  :has-relations-m
   (:type relation-descriptor-m :min 0 :max inf)
  :has-parameters-m
   (:type parameter-m :min 0 :max inf)
  :has-instances-m
   (:type instance-m :min 0 :max inf))

(define-concept :name relation-descriptor-m
  :specialization-of named-domain-object-m
  :relation-of-m (:type concept-m)
  :has-left-side-m (:type some-m :min 1:max 1)
  :has-right-side-m (:type some-m :min 0:max inf)
  :has-relation-definition-m
   (:type relation-definition-m :min 1:max 1))

(define-concept :name some-m
  :specialization-of domain-object-descriptor-m
  :parameters ( (lower-bound [0 inf])
              (upper-bound [0 inf]))
  :in-relation-left-m
   (:type relation-descriptor-m)
  :in-relation-right-m
   (:type relation-descriptor-m)
  :some-of (:type concept-m))

(define-concept :name instance-m
  :specialization-of named-domain-object-m
  :instance-of-dom-m (:type domain-m)
  :instance-of-m (:type concept-m)
  :matching-instance-of-m
   (:type conceptual-constraint-m)
  :has-relations-m
   (:type relation-descriptor-m :min 0 :max inf)
  :has-parameters-m
   (:type parameter-m :min 0 :max inf))
```

```
(define-conceptual-constraint :name consistency-rule-5
  :structural-situation
   ((?c    :name concept-m)
    (?rd   :name relation-descriptor-m
           :relation-of-m ?c)
    (?svt  :name some-m
           :in-relation-left-m ?rd)
    (?stdi :all :name some-m
           :in-relation-right-m ?rd))
  :constraint-calls
   ((all-isp ?stdi ?svt)))

(define-concept :name conceptual-constraint-m
  :specialization-of named-domain-object-m
  :structural-situation
   (:type concept-expression-m :min 1 :max inf)
  :constraint-calls
   (:type constraint-call-m :min 1 :max inf)
  :matching-instances
   (:type instance-m :min 0 :max inf))

(define-conceptual-constraint
  :name instance-consistency-rule-6
  :structural-situation
   ((?cc :name conceptual-constraint-m)
    (?i :name instance-m
        :self (:condition
                (instance-matches-cc-p *it* ?cc))))
  :constraint-calls
   ((ensure-relation ?i matching-instance-of ?cc)
    (ensure-relation ?cc matching-instancs ?i)))
```

Figure 5. Formalizing the knowledge elements shown in Figure 4(a) and some consistency rules with CDL concepts.

a knowledge representation language with a logic-based semantic on the metalevel, i.e. CDL instead of UML derivates like EMOF [24]. Doing so, inference techniques provided by the knowledge representation language can be used, e.g. constraint propagation. This enables the realisation of the Knowledge Reflection Server as introduced in the next section.

4.2 Extension of Meta-CDL-KB for Biological Classification

Because the metalayers are also realized with a knowledge-modeling language (here CDL) they can be extended by simply adding concept and constraint definitions to the metaknowledge base. Thus, the modeling facilities provided by such languages can not only be used for specifying the languages itself (see Section 4.1) but also for domain-specific extensions on the metalayers. In Figure 6 the extensions of $R1^{MM}$ (a) and $R1^{M}$ (b and c) for the biological classification domain are sketched, yielding to $BioCl^{MM}$ and $BioCl^{M}$ respectively. $BioCl^{MM}$ consists simply of one concept which specifies a biological rank (Figure 6 (a)). On $BioCl^{M}$, beside the concept definitions that define the metaconcepts used in $BioCl^{D}$ (see Figure 3 and 1), the conceptual constraint `Specific-Characteristics` is defined on the metalayer. This conceptual constraint checks every combination of biological classes for having specific characteristics by comparing their differences models on $BioCl^{D}$. Thus, with this conceptual constraint on $BioCl^{M}$ it is specified that a biological class should have a unique combination of characteristics. With the constraint, also classes of different phylums are tested (e.g. chordate and echinoderms). On $BioCl^{D}$, these kinds of constraints are hard to define because they are typically not related to one specific concept but to several. Furthermore, such constraints are usually part of some modeling guidelines, e.g. for biological classification such documents state that the definitions of biological classes should be unique. Thus, by the approach presented here a *modeling of modeling guidelines* on the metalayer is achieved.

5 Knowledge Reflection Servers

Each layer described in Section 2 is realized through a Knowledge Reflection Server (KRS). Every server monitors the layer below it and consists of the appropriate model and a configuration system which interprets the model. This has the advantage of using declarative models at each metalayer as well as the possibility to apply inference techniques like e.g. constraint programming at the metalayer. Each server supplies knowledge-based services that can be called by a server below it for obtaining a judgement of its own used models. For example, a KRS monitors the activities during the construction of the domain model $BioCl^{D}$, i.e. during the domain representation phase. If e.g. a concept c of the domain is defined with `define-concept` on $R1^{D}$ the KRS on $R1^{M}$ is informed (see Figure 7) for checking its consistency. Furthermore, a KRS

- supplies services like *check-knowledge-base*, *add-conceptual-constraint*,
- creates appropriate instances of metaconcepts of the Meta-CDL-KB, e.g. *concept-m* or *conceptual-constraint-m*,
- uses constraint propagation for checking the consistency rules,
- applies the typical model configuration process for completing the configuration, e.g. adds mandatory parts,
- checks consistency of created domain specific concepts, e.g. of $BioCl^{D}$,
- can supply new concepts for the layer below, which may be computed by machine learning methods,
- monitors the reasoning process, e.g. for evaluating reasoning performance, and thus, makes reasoning explicit,
- can create and use explanations,
- may solve conflicts that occur during the domain representation phase,
- may apply domain-specific metaknowledge, e.g. “ensuring specific differentiating characteristics of biological classes” with metacontraints as shown in Figure 6.

We implemented parts of the KRS services based on the configuration system KONWERK [10], but have not yet finished the extensive evaluation. The Meta-CDL-KB and its extensions were used for checking versions of knowledge bases for the biological classification domain. Thereby, a mapping of concept definitions of $BioCl^{D}$ to instance descriptions of $BioCl^{M}$ was realized, i.e. concept definitions on one layer are instance definitions on the next upper layer. Furthermore, the concrete syntax for defining concepts in $BioCl^{D}$ was extended, thus, metaproperties can be specified in $BioCl^{D}$. Both implementation issues as well as interfaces for the server functionality could be realized straight forward because of the flexibility of the underlying implementation language Common Lisp. However, the reasoning facilities provided by KONWERK could directly be used for scrutinizing the layers.

The validation of this approach was shown with the help of the following three scenarios, which also illustrate the use of the KRS:

- First, metaknowledge modeling can be adequately enabled, to this no workarounds with specialisations, like in [32] are needed.
- Checking domain dependent constraints: In the event of the introduction of a new biological class in $BioCl^{D}$, the KRS advises according to the specific characteristics of the constraint (e.g. see Figure 6 (c)) on $BioCl^{M}$ whether it is a biological class or not.
- Checking domain independent consistency rules: Nonrelevant to the kind of domain the domain independent rules, like the number restrictions (see Figure 5) will be checked at all times. For example, when a new kind of mammal will be introduced the defined restrictions, like the number of middle ear bones has to be conform.

6 Discussion and Related Work

Main properties of the Reasoning-Driven Architecture realized with the Knowledge Reflection Servers (KRS) as described in the previous sections are:

- the introduction of a model on one layer that represents the knowledge facilities used on the layer below it (i.e. metaknowledge models).
- the use of existing knowledge-based systems with their reasoning facilities on each layer, especially on metalayers. This enables reflection about knowledge.
- the mapping of concepts of one layer to instances of the next higher layer. This approach has the potential of using more tractable instance related inference methods instead of concept reasoning.
- the support of declarative knowledge modeling on several metalayers. This enables the modeling of knowledge and metaknowledge at the same time. Metaknolwledge is typically specified in modeling guidelines. Thus, the described approach enables the modeling of modeling guidelines.

```
(define-concept :name Kingdom-m
  :specialization-of concept-m
  :metaconcept Biological-Rank-mm)

(define-concept :name Phylum-m
  :specialization-of Kingdom-m
  :metaconcept Biological-Rank-mm)

(define-concept :name Class-m
  :specialization-of Phylum-m
  :has-relations-m
   (:type Difference-descriptor-m :min 1 :max 1)
  :metaconcept Biological-Rank-mm)
```

```
(define-concept :name Difference-descriptor-m                b)
  :specialization-of relation-descriptor-m
  :relation-of-m (:type Class-m)
  :has-left-side-m
    (:type Characteristic-some-m :min 1:max 1)
  :has-right-side-m
    (:type Characteristic-some-m :min 0:max inf))

(define-concept :name Characteristic-some-m
  :specialization-of some-m
  :in-relation-left-m
   (:type Difference-descriptor-m)
  :in-relation-right-m
   (:type Difference-descriptor-m)
  :some-of (:type Characteristic-m))

(define-concept :name Characteristic-m
  :specialization-of concept-m
  :has-relations-m
   (:type Difference-descriptor-m :min 1 :max 1)
  :metaconcept Biological-Rank-mm)
```

```
(define-conceptual-constraint                     c)
  :name Specific-Characteristics
  :structural-situation
  ((?c1 :name Class-m)
   (?c2 :name Class-m
        :self #'(not-eq *it* ?c1))
   (?r1 :name Difference-descriptor-m
        :relation-of-m ?c1)
   (?r2 :name Difference-descriptor-m
        :relation-of-m ?c2)
   (?d1 :all :name Characteristic-some-m
        :in-relation-right-m ?r1)
   (?d2 :all :name Characteristic-some-m
        :in-relation-right-m ?r2)
  :action-part
  ((all-different ?d1 ?d2)))
```

```
(define-concept :name Biological-Rank-mm          a)
  :specialization-of concept-m)
```

Figure 6. Extending Meta-CDL-KB with concepts and conceptual constraints for the domain of biological classification, i.e. parts of $BioCl^M$.

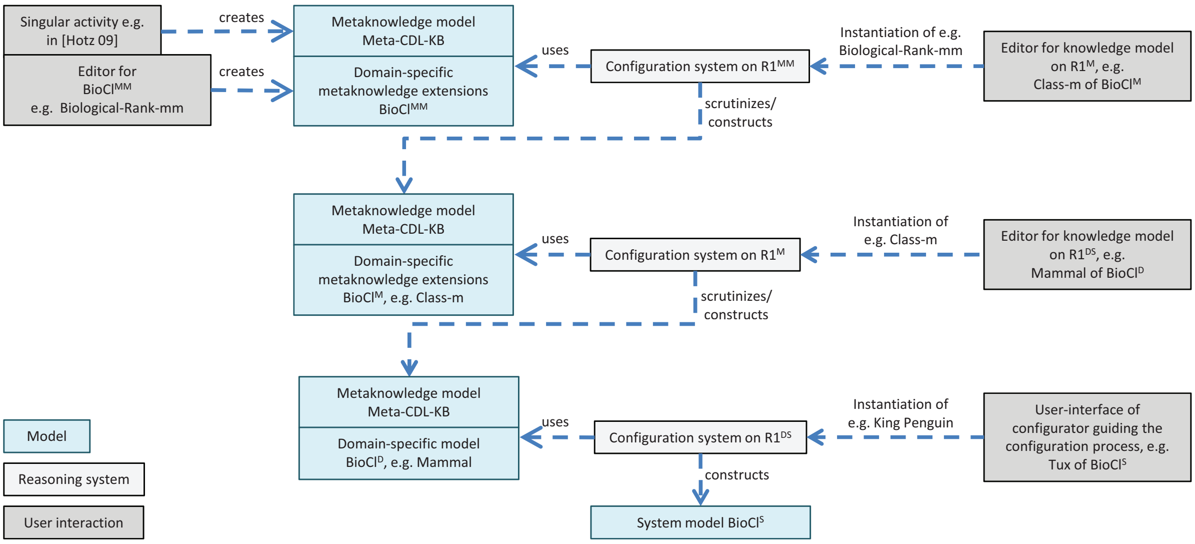

Singular activity e.g. in [Hotz 09]
creates
Metaknowledge model Meta-CDL-KB
Editor for BioCl^MM e.g. Biological-Rank-mm
creates
Domain-specific metaknowledge extensions BioCl^MM
uses
Configuration system on R1^MM
Instantiation of e.g. Biological-Rank-mm
Editor for knowledge model on R1^M, e.g. Class-m of BioCl^M
scrutinizes/ constructs
Metaknowledge model Meta-CDL-KB
Domain-specific metaknowledge extensions BioCl^M, e.g. Class-m
uses
Configuration system on R1^M
Instantiation of e.g. Class-m
Editor for knowledge model on R1^DS, e.g. Mammal of BioCl^D
scrutinizes/ constructs
Metaknowledge model Meta-CDL-KB
Domain-specific model BioCl^D, e.g. Mammal
uses
Configuration system on R1^DS
Instantiation of e.g. King Penguin
User-interface of configurator guiding the configuration process, e.g. Tux of BioCl^S
constructs
System model BioCl^S
Model
Reasoning system
User interaction

Figure 7. Monitoring the construction of metamodels and domain models through Knowledge Reflection Servers realized with configuration systems. A server using $BioCl^M$ scrutinizes $BioCl^D$ and a server using $BioCl^{MM}$ scrutinizes $BioCl^M$.

The use of reasoning methods in RDA is achieved by replacing the Meta Object Facility (MOF) of MDA [23], which is based on UML, with CDL. Other knowledge-representation languages like the Web Ontology Language (OWL) [2] could also be considered for being used on the layers. However, CDL is quite expressive, e.g. also constraints can be expressed on each layer. For realizing the KRS, even more important for us was the possibility to add server technologies to the knowledge-representation language CDL. However, by replacing the Meta-CDL-KB with a metamodel for OWL (e.g. the Ontology Definition Metamodel (ODM) [26]) one could use RDA for scrutinizing the construction of domain models written in OWL. However, with the Meta-CDL-KB a knowledge-based implementation of a metamodel is provided and was used from a knowledge-based system (here a configurator). [3] and [11] present also approaches that include semantics on the metalayer, similar as the Meta-CDL-KB metamodel does. However, these approaches do not emphasise the use of reasoning methods on each layer as well as the capability to define domain-specific extensions on the metalayer. Furthermore, the RDA presented in this paper allows for an arbitrary number of metalayers. By introducing a configurator which allows the definition of procedural knowledge for controling the used reasoning techniques, in our approach the realization of metastrategies on the metalayers can be considered (see also [17]). (See Section 2 for further comparison to MDA and OWL.)

The creation of a metamodel for CDL with the aid of CDL has its tradition in self-referencing approaches like Lisp-in-Lisp [7] or the metaobject protocol, which implements CLOS (the Common Lisp Object System) with CLOS [21]. Such approaches demonstrated the use of the respective language and provide reflection mechanisms. With our approach such reflection mechanisms are extended from object-oriented reflection (e.g. about introspection of methods) to knowledge-based reflection (e.g. about used concepts and constraints for modeling a domain). Thus, our approach provides reflection about knowledge and a way to self-awareness of agents.

A Knowledge Reflection Server is basically an implementation of a configuration tool on the basis of the Meta-CDL-KB, i.e. of a configuration model. A typical configuration tool is implemented with a programming language and an object model implemented with it. During this implementation one has to ensure correct behavior of model construction and the inference techniques. By using CDL, this behavior (e.g. the consistency rules) is declaratively modeled, not procedurally implemented. The bases for this declarative realization are of course the procedural implementation of the inference techniques, so to speak, as a bootstrapping process. However, our approach gives indications how to open up the implementation of configuration systems or other knowledge-based systems for allowing domain-specific extensions and extensions to the inference methods and the used knowledge-modeling language.

The approach of the Metacognitive Loop presented in [1] considers the use of metalevels for improving learning capabilities and human-computer dialogs. Similar to our approach, it points out the need for enhancing agents with capabilities to reason about their cognitive capabilities for gaining self-awareness and a basis for decisions about what, when, and how to learn. However, our approach stems more from the ontology and technical use point of view and supports the idea of using metalevels from that side.

7 Conclusion

This paper presents a technology for using knowledge-based systems on diverse metalayers. Main ingredients for this task are models about knowledge (metamodels). Through the use of knowledge-based systems as they are, a Reasoning-Driven Architecture is provided. It enables reasoning facilities on each metalayer, opposed to the Model-Driven Architecture which focusses on transformations. The Reasoning-Driven Architecture is realized through a hierarchy of Knowledge Reflection Servers based on configuration systems. Future work will include meta strategies for conducting reasoning methods on the metalayers, a complete implementation of the servers, and industrial experiments in the field of knowledge engineering.

REFERENCES

[1] Michael L. Anderson and Donald R. Perlis, ‘Logic, Self-awareness and Self-improvement: the Metacognitive Loop and the Problem of Brittleness’, *J. Log. and Comput.*, **15**(1), 21–40, (2005).

[2] Grigoris Antoniou and Frank Van Harmelen, ‘Web Ontology Language: OWL’, in *Handbook on Ontologies in Information Systems*, pp. 67–92. Springer, (2003).

[3] T. Asikainen and T. Männistö, ‘A Metamodelling Approach to Configuration Knowledge Representation’, in *Proc. of the Configuration Workshop on 22th European Conference on Artificial Intelligence (IJCAI-2009)*, Pasadena, California, (2009).

[4] Colin Atkinson and Thomas Kühne, ‘Model-Driven Development: A Metamodeling Foundation’, *IEEE Softw.*, **20**(5), 36–41, (2003).

[5] F Baader, D Calvanese, D McGuinness, D Nardi, and P Patel-Schneider, *The Description Logic Handbook*, Cambridge University Press, 2003.

[6] Gilad Bracha and David Ungar, ‘Mirrors: design principles for meta-level facilities of object-oriented programming languages’, in *OOPSLA ’04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 331–344, New York, NY, USA, (2004). ACM.

[7] R.A. Brooks, R.P. Gabriel, and L. Steele Jr., ‘Lisp-in-Lisp: High Performance and Portability’, in *Proc. of Fifth Int. Joint Conf. on AI IJCAI-83*, (1983).

[8] M. Buchheit, R. Klein, and W. Nutt, ‘Constructive Problem Solving: A Model Construction Approach towards Configuration’, Technical Report TM-95-01, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, (January 1995).

[9] Dragan Gaŝević, Dragan Djuric, Vladan Devedzic, and Bran Selic, *Model Driven Architecture and Ontology Development*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] A. Günter and L. Hotz, ‘KONWERK - A Domain Independent Configuration Tool’, *Configuration Papers from the AAAI Workshop*, 10–19, (July 19 1999).

[11] P. Haase, R. Palma, and d’Aquin M., ‘Updated Version of the Networked Ontology Model’, Project Deliverable D1.1.5, Neon Project, (2009). www.neon-project.org.

[12] W. Hesse, ‘More Matters on (Meta-)Modelling: Remarks on Thomas Kühne’s ”Matters”’, *Journal on Software and Systems Modeling*, **5**(4), 369–385, (2006).

[13] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean, ‘SWRL: A Semantic Web Rule Language Combining OWL and RuleML’, W3c member submission, World Wide Web Consortium, (2004).

[14] L. Hotz, ‘Construction of Configuration Models’, in *Configuration Workshop, 2009*, eds., M. Stumptner and P. Albert, Workshop Proceedings IJCAI, Pasadena, (2009).

[15] L. Hotz, *Frame-based Knowledge Representation for Configuration, Analysis, and Diagnoses of technical Systems (in German)*, volume 325 of *DISKI*, Infix, 2009.

[16] L. Hotz and B. Neumann, ‘Scene Interpretation as a Configuration Task’, *Künstliche Intelligenz*, **3**, 59–65, (2005).

[17] L Hotz, K Wolter, T Krebs, S Deelstra, M Sinnema, J Nijhuis, and J MacGregor, *Configuration in Industrial Product Families - The ConIPF Methodology*, IOS Press, Berlin, 2006.

[18] E. Hyvönen, ‘Constraint Reasoning based on Interval Arithmetic: the Tolerance Propagation Approach’, *Artificial Intelligence*, **58**, 71–112, (1992).

[19] U. John, *Konfiguration and Rekonfiguration mittels Constraint-basierter Modellierung*, Infix, St. Augustin, 2002. In German.

[20] Sonya E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer’s Guide to CLOS*, Addison-Wesley, 1989.

[21] J. Kiczales, D. G. Bobrow, and J. des Rivieres, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.

[22] T. Kühne, ‘Matters of (Meta-)Modeling’, *Journal on Software and Systems Modeling*, **5**(4), 369–385, (2006).

[23] OMG, *MDA Guide Version 1.0.1, omg/03-06-01*, Object Management Group, 2003.

[24] OMG, *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, Object Management Group, 2006.

[25] OMG, *Unified Modeling Language: Infrastructure, version 2.1.1, formal/07-02-06*, Object Management Group, 2007.

[26] OMG, *Ontology Definition Metamodel, Version 1.0*, Object Management Group, 2009.

[27] Gilbert Paquette, 'An Ontology and a Software Framework for Competency Modeling and Management', *Educational Technology & Society*, **10**(3), 1–21, (2007).

[28] J. Pitrat, 'Metaconnaissance, avenir de l'Intelligence Artificielle.', Technical report, Hermes, Paris, (1991).

[29] K.C. Ranze, T. Scholz, T. Wagner, A. Günter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt, 'A Structure-Based Configuration Tool: Drive Solution Designer DSD', *14. Conf. Innovative Applications of AI*, (2002).

[30] D. Sabin and E.C. Freuder, 'Configuration as Composite Constraint Satisfaction', in *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pp. 153–161. AAAI Press, (1996).

[31] C. Schröder, R. Möller, and C. Lutz, 'A Partial Logical Reconstruction of PLAKON / KONWERK', in *DFKI-Memo D-96-04, Proceedings of the Workshop on Knowledge Representation and Configuration WRKP'96*, ed., F. Baader, (1996).

[32] Stefan Schulz, Holger Stenzhorn, and Martin Boeker, 'The ontology of biological taxa', in *ISMB*, volume 24, pp. 313–321, (2008).

[33] K. Wolter, T. Krebs, and L. Hotz, *Mass Customization - Challenges and Solutions*, chapter Model-based Configuration Support for Software Product Families, 43–62, Springer, 2006.

[34] Dong Yang, Ming Dong, and Rui Miao, 'Development of a Product Configuration System with an Ontology-Based Approach', *Comput. Aided Des.*, **40**(8), 863–878, (2008).

Challenges of Knowledge Evolution in Practice

Andreas Falkner and **Alois Haselböck**
Siemens AG Österreich, Austria, email: {andreas.a.falkner,alois.haselboeck}@siemens.com

Abstract.
This paper enumerates some of the most important challenges which arise in practice when changing a configurator knowledge base: redesign of the knowledge base, schema evaluation of the data bases, upgrade of configuration instances which are already in the field, adaptation of solver, UI, I/O, and test suites. Partially, there are research theories for some of these challenges, but only few of them are already available in tools and frameworks. So we do not provide solutions here, but we want to stimulate a deeper discussion on knowledge evolution. Moreover, we give a self-contained real-world example as a proposal for future research.

1 INTRODUCTION

Product configurators have a long history in artificial intelligence (see [1], [2]) and nowadays various systems based on AI methods are available: scientific frameworks and tools (like Choco, CHR, DLV, etc.) as well as commercial programs (like from Camos, Configit, ILOG, Oracle, SAP, Tacton, etc.). However, many of those systems do not cope well with a scenario which arises in practice whenever long-living products are configured: Knowledge evolution.

Knowledge about the product grows and changes over time, e.g. new types of product components get available, old ones run out of production, experiences in production or operation cause tightening or loosening of assembly restrictions between components' properties, etc. The corresponding configurator's knowledge base (types, relations, attributes, constraints) must be adjusted accordingly. This may also affect existing configuration instances based upon that knowledge base. The necessary reconfiguration - i.e., in general, modification of an existing product individual to meet new requirements - has been subject to only little research work, e.g. by [3] who suggest explicit rules (conditions and operations) and invariants for necessary changes of instances. A more recent paper [4] concentrates on replacing (faulty) components with new or better ones and poses some research questions still to be answered.

This paper identifies some aspects of knowledge evolution worth to be researched in detail. As an example we use a fictitious people counting system for museums. The structure of the problem is similar to problems we encountered in different real-world domains of our technical product configurators. At first we describe the original problem and model it with an object-oriented (UML) and constraint-based (GCSP) approach. Then we specify some exemplary changes. This leads to several challenges how to cope with the implied knowledge evolution: schema evolution, data upgrade, solver adaptations, adaptations to UI, I/O, and tests.

2 ORIGINAL PROBLEM DESCRIPTION

A people counting system (e.g. for use in museums) consists of door sensors, counting zones (rooms), and communication units (see [5] for more details).

A door sensor detects everybody who moves through its door (directed movement detection). There can be doors without a sensor.

Any number of rooms may be grouped to a counting zone. Each zone knows how many persons are in it (counting the information from the sensors at doors leading outside of the zone). Zones may overlap or include other zones, i.e. a room may be part of several zones.

A communication unit can control at most 2 door sensors and at most 2 zones. If a unit controls a sensor which contributes to a zone on another unit, then the two units need a direct connection: one is a partner unit of the other and vice versa. Each unit can have at most 2 partner units. This relation is symmetric but neither transitive nor reflexive.

PartnerUnits problem: Given a consistent configuration of door sensors and zones, find a valid assignment of communication units (with max. 2 partners). We assume that a solution requires only a minimal number of units: the smallest integer greater or equal to the half of the maximum of the number of zones and the number of door sensors. Therefore we treat it as a constraint satisfaction problem, not as a constraint optimization problem.

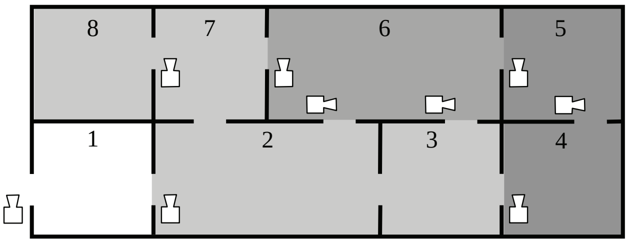


Figure 1. Room layout of example 1

Example 1: Rooms 1 to 8 with eleven doors, nine of them having a door sensor (named D01, D12, D26, D34, etc. by the rooms which they connect). Eight zones named by the rooms which they contain: Z1 (white), Z2345678, Z2367, Z2378 (light gray), Z4, Z45 (dark grey), Z456, Z6 (medium grey).

The relation between zones and door sensors can be represented as a bi-partite graph (see Figure 2). An instance of a solution using the minimum of five units is shown in Table 1.

Unit	Zones		DoorSensors		PartnerUnits	
U1	Z1	Z2345678	D01	D78	U2	-
U2	Z2367	Z45	D12	D56	U1	U3
U3	Z2378	Z6	D34	D67	U2	U4
U4	Z456	Z4	D26	D36	U3	U5
U5	-	-	D45	-	U4	-

Table 1: A minimal solution of example 1

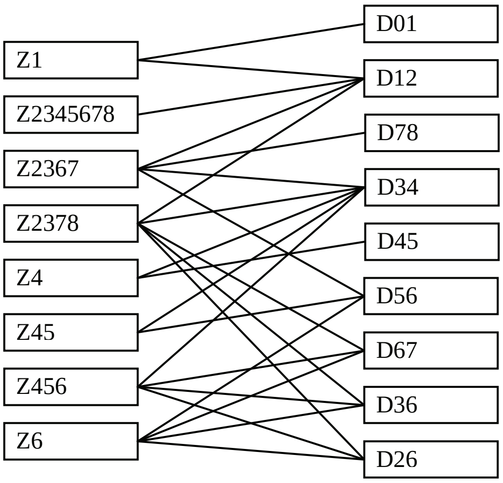


Figure 2. Relation between zones and door sensors of example 1

2.1 Modeling

A knowledge base mainly consists of the representation of the configuration components and the constraints and rules defining valid solutions. For many technical domains, the models get complex and large, so that a high-level modeling language is required. It should provide for an easy, natural and elegant problem description, supporting readability, validation and maintainability of the model.

UML class diagrams are a common way to describe the structure of a system in object-oriented modeling. In combination with OCL (Object Constraint Language) it is also expressive enough to describe product configuration [6].

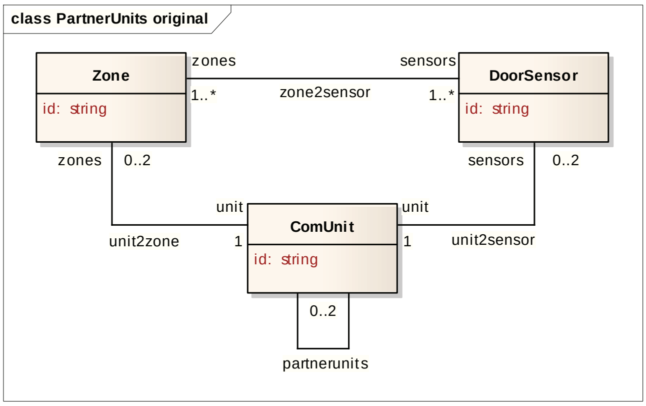


Figure 3. UML diagram of PartnerUnits problem

The UML diagram of Figure 3 shows a class diagram derived from the description. It does not represent rooms as they are not part of the concrete problem at hand. For all associations, it contains the cardinality constraints (minimal/maximal number of connected elements). The fact that the ***partnerunits*** association is derived from the path over the ***zone2sensor*** relation must be expressed with an OCL constraint:

```
context ComUnit inv:
myPartnerUnitsSensor =
  sensors.zones.unit->excluding(self)->asSet()
and myPartnerUnitsZone =
  zones.sensors.unit->excluding(self)->asSet()
and myPartnerUnitsSensor->union(myPartnerUnitsZone)
  ->size() <= 2
```

For each unit, it first computes the sets of different units reachable via the ***zones*** association and the ***sensors*** association, respectively. Then, it checks the cardinality of their union.

2.2 Generative Constraint Satisfaction

Constraint satisfaction is widely used to represent and solve configuration problems. A CSP in the classical sense consists of a fixed set of variables and their domains, as well as constraints which restrict the assignment of the variables. A valid solution is an assignment of all variables with values from their domains where all constraints are satisfied. Any problem instance of the PartnerUnits problem could be specified as a static CSP because a fixed set of necessary variables could be derived. Due to the principally unbounded number of its components and their connections, however, we use Generative CSP [7].

A GCSP knowledge base of the PartnerUnits problem is shown in Figure 4. Class, association, and constraint definitions represent the UML diagram of Figure 3 in a straight-forward and compact way.

```
class Zone
class DoorSensor
class ComUnit

assoc: Zone.sensors(1-*) - DoorSensor.zones(1-*)
assoc: Zone.unit(1) - ComUnit.zones(0-2)
assoc: DoorSensor.unit(1) - ComUnit.sensors(0-2)
assoc: ComUnit.partnerunits(0-2) - self

constraint ComUnit.derivedPartners:
  partnerunits = zones.sensors.unit
               + sensors.zones.unit
               - self
```

Figure 4. GCSP formulation of the PartnerUnits problem

The classes (i.e. component types) are ***Zone***, ***DoorSensor***, and ***ComUnit***. Each class represents a theoretically infinite set of instances (i.e. components). A class can have attributes, associations and constraints. Whenever a new instance of a class is created, it gets instances of its attributes, associations and constraints, too. This is the object-oriented view of the modeling.

From the constraint-oriented point of view, attributes and associations represent the variables. The domain of an attribute variable is its type, e.g. Boolean, an integer interval, etc. Associations are bidirectional and induce two association variables, one for each side. The domain of such an association variable is the set of all instances of the class specified on the other side of the association. E.g. the association definition

```
assoc: Zone.unit(1) - ComUnit.zones(0-2)
```

represents the connection of zones to units. The two association variables induced are ***Zone.unit*** (the link from a zone to its unit) and

ComUnit.zones (the link from a unit to all its associated zones). Allowed cardinalities are given in brackets. Implicit constraints check that all instances associated to an association variable are of the specified type (e.g. ***ComUnit*** for ***Zone.unit***) and that the given cardinalities are not violated (e.g. ***Zone.unit*** must contain exactly one instance).

The constraint specifies in the context of a ***ComUnit*** instance, which elements are to be in the partner association of that unit. These are all units reachable via its zones and its sensors, where the unit itself is not member of the association.

Generative CSP is well-suited for the modeling of configuration problems because of its object-oriented approach (natural and maintainable formulation of the problem structure), its constraint-orientedness (declarative formulation of the problem logics), and its dynamicity. Suitable solvers (e.g. backtracking, heuristic repair) can easily be integrated.

3 CHANGED PROBLEM

Products and product designs change over time - in our case, there are two extensions:

1. A sensor can be connected to an outside system (numbered from 1 upwards). Outside sensors must be placed on special (outside) communication units which must not be changed or used otherwise. For example, D01 of example 1 connects to outside system 1 and shall be placed on a separate unit.

2. There is a new type of zone which collects information from other zones, not from sensors. Of course, the existing constraints shall hold (assignment to a unit with at most 2 partner units). For example, Z456 can be combined from Z45 and Z6, Z2345678 from Z2378 and Z456.

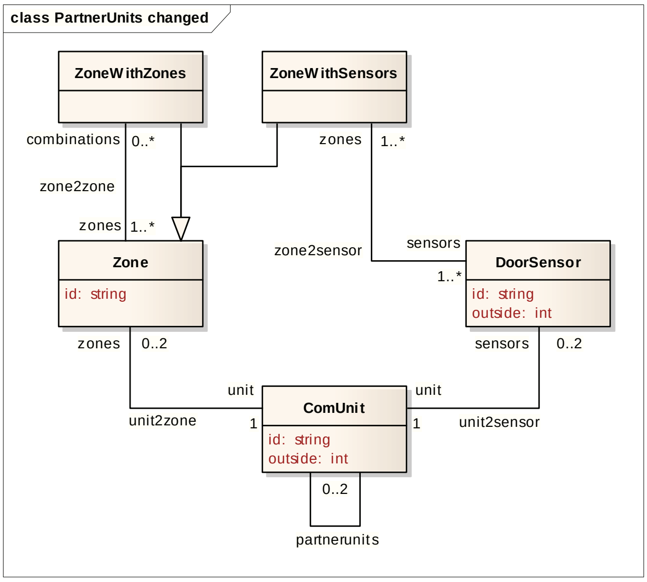


Figure 5. UML diagram of changed PartnerUnits problem

Figure 5 shows a UML diagram which models the changed knowledge base: ***DoorSensor*** and ***ComUnit*** have a new integer attribute which distinguishes connections to outside systems (0 is used for sensors without outside connection). ***Zone*** gets two sub-classes - one for the original zone type (therefore the ***zone2sensor*** association is redirected to it), another for the new type (including a new ***zone2zone*** association).

4 CHALLENGES

In the following, we identify some of the major challenges when a knowledge base is to be changed.

We describe them on the basis of our example and our chosen representation. The changes in the example may give an impression of interesting cases which can occur in real-world. Concerning the representation language, we also tried other approaches like Choco, DLV, Alloy, etc. They showed more or less similar behavior as our GCSP representation, so that the latter stands exemplarily in the following.

4.1 Knowledge Evolution (KB -> KB')

As products and product designs change over time, so the knowledge base of the product configurator must be changed, too.

Routine changes, like price changes, should be treated in a way to avoid expensive knowledge base updates. E.g., instead of hard-coding the prices of the different components in the knowledge base, prices are typically loaded from an Excel table or a database during start-time of the configurator. Care must be taken when values from the external source are directly used in constraints.

Structural changes of the knowledge base like the examples in section 3 are more difficult. The modified GCSP formulation of the knowledge base is shown in Figure 6, the changes are written in bold (we do not have to delete old content).

```
abstract class Zone
class ZoneForSensors isa Zone
class ZoneForZones isa Zone
class DoorSensor
  attr outside: 0-*
class ComUnit
  attr outside: 0-*

assoc: ZoneWithSensors.sensors(1-*)
                      - DoorSensor.zones(1-*)
assoc: ZoneWithZones.zones(1-*)
                      - Zone.combinations(0-*)
assoc: Zone.unit(1) - ComUnit.zones(0-2)
assoc: DoorSensor.unit(1) - ComUnit.sensors(0-2)
assoc: ComUnit.partnerunits(0-2) - self

constraint ComUnit.derivedPartners:
  partnerunits = zones.sensors.unit
               + sensors.zones.unit
               + zones.combinations.unit
               + zones.zones.unit
               - self

constraint DoorSensor.equalOutside:
  outside = unit.outside
```

Figure 6. GCSP formulation of the modified PartnerUnits problem

In general, the following changes are possible and should be handled in a way so that the other challenges can be solved as well:

- Change property names, attribute types, association domains and cardinalities, inheritance hierarchy, etc.
- Add new component types, attributes, or associations.

- Delete component types, attributes, or associations.
- Change, add, or delete constraints.

In our example, the knowledge engineer had to decide which class for zones (if any) to reuse in the model - and he decided for the abstract general class and against the old semantics (for that we would have needed a new super class of the existing ***Zone*** and the new ***ZoneWithZones*** as well as a redirection of ***unit2zone*** instead of ***zone2sensor***). Furthermore, he decided just for a new attribute for ***ComUnit*** instead of a new class.

By that, the existing constraints and cardinalities can stay unchanged. Only the computation of the ***partnerunits*** association must be extended with the path over ***zone2zone***. A new constraint for outside systems is necessary (outside number of sensor and its unit must be the same). Furthermore, implicit knowledge strongly suggests to disallow cycles in the ***zone2zone*** association (omitted for brevity).

Challenges: Are there any patterns or rationale for putting such design decisions for change on a sound base?

4.2 Schema Evolution (DB -> DB')

When existing configuration instances (I) shall be accessed with the new knowledge base (KB'), it is necessary to also change the schema of the database (DB) where the instances I are stored. They must be upgraded to instances which are based on the new database schema DB' (as part of KB').

In our example, the schema evolution must change the type of all ***Zone*** instances to ***ZoneWithSensors***. In all ***ComUnit*** instances, it must add the ***outside*** attribute and set it to 0 (initial value). In all ***DoorSensor*** instances, it must add the ***outside*** attribute and set it to the correct value (which must be supplied externally). The second action may be derived automatically from KB', the others not.

A way to formalize this transformation of a source schema to a target schema is the theory of data exchange [8]. For our example, the source schema is defined by the following predicates:

```
{sensor/1, zone/1, unit/1,
 zone2sensor/2, unit2sensor/2, unit2zone/2,
 unit2unit/2}
```

The solution instance shown in Table 1 is therefore represented by:

```
{sensor(d01), sensor(d12), …,
 zone(z1), zone(z2345678), zone(z2367), …,
 unit(u1), unit(u2), …,
 zone2sensor(z1, d01), zone2sensor(z1, d12), …,
 unit2sensor(u1, d01), unit2sensor(u1, d78), …,
 unit2zone(u1, z1), unit2zone(u1, z2345678), …,
 unit2unit(u1, u2), unit2unit(u2, u1), …}
```

The target schema is:

```
{sensor/1, zone4zones/1, zone4sensors/1, unit/1,
 sensor_outside/2, unit_outside/2,
 zone2sensor/2, unit2sensor/2, unit2zone/2,
 unit2unit/2}
```

Data exchange transformation rules map a source configuration to a target configuration (we assume that information about the outside system of sensors is available in an externally defined table named ***sensor_info***):

```
zone(X) -> zone4sensors(X)
unit(X) -> unit(X), unit_outside(X, 0)
sensor(X) and sensor_info(X, V)
                 -> sensor(X), sensor_outside(X, V)
zone2sensor(X, Y) -> zone2sensor(X, Y)
unit2sensor(X, Y) -> unit2sensor(X, Y)
unit2zone(X, Y) -> unit2zone(X, Y)
unit2unit(X, Y) -> unit2unit(X, Y)
```

The Chase procedure then generates a target instance by applying the transformation rules to the source instance:

```
{sensor(d01), sensor(d12), …,
 sensor_outside(d01, 1), sensor_outside(d12, 0), …,
 zone4sensors(z1), zone4sensors(z2345678), …,
 unit(u1), unit(u2), …,
 unit_outside(u1, 0), unit_outside(u2, 0), …,
 zone2sensor(z1, d01), zone2sensor(z1, d12), …,
 unit2sensor(u1, d01), unit2sensor(u1, d78), …,
 unit2zone(u1, z1), unit2zone(u1, z2345678), …,
 unit2unit(u1, u2), unit2unit(u2, u1), …}
```

The performance of this Chase depends on which logical expressions are used in the transformation rules. In general, there might be various changes of different difficulty, e.g. renaming, type changes, value shifts, conditional changes, etc.

Challenges: What are complete, clear, fast, and reliable methods to specify necessary changes for schema evolution?

4.3 Upgrade Instances (I -> I')

The new configuration instance is now in the schema of DB', but not necessarily consistent to all the constraints of KB'. In order to get a consistent I', the violated constraints must be repaired. KB' might contain a new constraint which contradicts some part of the configuration which is valid with respect to the old KB. Examples include simple cases like new versions of components, but also complicated relations between objects.

In our example, D01 should be placed on a separate unit with outside number 1. Just changing the value of the ***outside*** attribute of its unit is not enough as there are other zones and sensors on that unit. Moving D01 to a new unit U6 is possible because at present U1 has just one partner unit and the second partner unit can be set to U6. The resulting configuration instance is shown in Table 2.

Unit	Zones		DoorSensors		PartnerUnits	
U1	Z1	Z2345678		D78	U2	**U6-**
U2	Z2367	Z45	D12	D56	U1	U3
U3	Z2378	Z6	D34	D67	U2	U4
U4	Z456	Z4	D26	D36	U3	U5
U5	-	-	D45	-	U4	-
U6	-	-	**D01**	-	**U1**	-

Table 2: A solution for the upgraded configuration

This solution is not minimal, however well acceptable because minimality of the solution (i.e. using as few units as possible) is less important than minimality of changes during upgrade (i.e. the upgraded configuration shall be as near to the original configuration as possible). Furthermore there is no trivial way to a better solution: Trying to reuse U5 instead of a new U6 just by exchanging D01 and D45 causes that U2 needs 3 partner units (U1, U3, U4) thus violating another cardinality constraint.

Challenges: How does the system know what to do (e.g. not setting ***DoorSensor.outside*** to 0)? Is it useful to specify a white list (e.g. direction of repair) or black list (e.g. do not change user-set values) for changes? What are repair actions which ensure that I' is as close to I as possible? Can this closeness be formally defined in terms of an optimization function, shifting the upgrade task from a search problem to an optimization problem?

In general, it is not possible to detect contradicting constraints by just checking the set of all constraints in the KB. The question is whether at least one configuration fulfils all constraints. If the constraint language is powerful enough (e.g. has quantors and multiplication), this question is undecidable. For simpler languages the best algorithm might still be of exponential order. An alternative approach based on diagnosis is described in [9].

4.4 Solver Adaptation

If domain-independent solvers are used, then no solver adaptations should be necessary (in the best case).

However, achieving sufficient performance often requires domain-specific algorithms. In a paper submitted to the forthcoming AI EDAM special issue on configuration, we present several domain-independent approaches which all fail to find solutions for large scale configurations (i.e. containing a lot of instances) even when they are simple. Those problem instances can only be solved with domain-specific heuristics exploiting the problem structure or after mapping the problem to special algorithms from appropriate fields, e.g. graph theory [10].

Unfortunately, the changed problem does no longer map to a bipartite graph which changes the problem structure considerably. Domain-specific solving must be adapted, usually with relative high costs. The same is true not only for our problem but also for other domain-specific implementations.

Challenges: Are there any approaches to reduce domain-specific algorithms or to support their adaptation to changed requirements?

4.5 UI Extension

The user interface must support configuration work in the best way.

In our example, the configuration process is severely affected by the new functionality: ***ZoneWithZones*** and ***ZoneWithSensors*** have different associations. Outside sensors and units should be visualized distinctively so that they can be easily distinguished. It must become possible to manually place sensors on units (accordingly to requirements of the outside system) before automatically placing the other sensors and zones.

Some GUI features can be derived directly from the knowledge base (e.g. classes, attributes, associations) but in general need to be filtered and/or parameterized (formatted) for better usability. Some tools allow therefore defining certain UI properties as part of the knowledge base.

Challenges: To what extent can the UI be defined as a natural extension of the knowledge base without introducing redundancy or overhead?

4.6 Adjust I/O

Our example does not contain explicit input or output interfaces. Typically, input could be some product information in external data bases (code numbers of components, variations, prices, etc.) or customer requirements for a product individual (e.g. in XML format). Possible outputs comprise bill of material, production plans, etc.

They must be adjusted to the changed model, e.g. output new attributes' values, input prices of the new types, etc. If many different output formats are involved - as is typical for configuration tasks in engineering-oriented companies - then synchronization may get complicated. A typical source of problems for changes concerning inputs is using the correct version of input data, i.e. the one matching the current knowledge schema.

Challenges: How to easily and reliably synchronize external data and output format with the changed model?

4.7 Upgrade Test Suites

Each knowledge base needs tests to ensure quality. This is especially true for safety-critical domains like railway systems. Common developer knowledge is that testing effort is as high as implementation effort. For knowledge base development it tends to be much higher as the specification language is very concise due to declarative, multi-functional constraints. Tests should include checking and solving with various combinations of set and free variables contributing to each constraint so that a high coverage of all scenarios is reached.

Regression test cases are usually based on problem statements and reference solution instances. These instances need to be upgraded, too (see sections 4.2 and 4.3).

Challenges: How to reduce test efforts, e.g. by automatically creating part of the tests? How can the regression test cases be upgraded when they fail because of changes in the knowledge base? How to distinguish errors in the changed knowledge base from necessary changes in the test cases?

5 CONCLUSION

We presented a real-world example of knowledge evolution and its effects on the design of the knowledge base and the upgrade of existing configuration instances (data bases).

Although over the last years various modeling languages for product configuration have been suggested (e.g. [11]), we do not see one which can cope well with all presented challenges.

Substantial research activities are required to improve efficiency and quality of redesign of knowledge bases for product configurators. A first step is the definition of the challenges in more detail and in a formal way.

REFERENCES

[1] D. Sabin and R. Weigel, ***Product Configuration Frameworks - A Survey***, IEEE Intelligent Systems 13(4), pp. 42-49, 1998.

[2] A. Felfernig, ***Standardized Configuration Knowledge Representations as Technological Foundation for Mass Customization***, IEEE Transactions on Engineering Management, 54(1), pp. 41-56, 2007.

[3] T. Männistö, T. Soininen, J. Tiihonen, and R. Sulonen, ***Framework and Conceptual Mode for Reconfiguration***, AAAI-99 Workshop on Configuration, pp. 59-64, 1999.

[4] P. Manhart, ***Reconfiguration - A Problem in Search for Solutions***, IJCAI-05 Workshop on Configuration, pp. 64-67, 2005.

[5] A. Falkner, A. Haselböck, and G. Schenner, ***Modeling Technical Product Configuration Problems***, ECAI-10 Workshop on Configuration (to appear), 2010.

[6] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, ***Configuration knowledge representation using UML/OCL***, Proceedings of the 5th

International Conference on the Unified Modeling Language, pp. 49-62, Springer-Verlag, 2002.

[7] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, ***Configuring large-scale systems with generative constraint satisfaction***, IEEE Intelligent Systems 13(4), pp. 59-68, 1998.

[8] R. Fagin, P.G. Kolaitis, R.J. Miller, and L. Popa, ***Data exchange : semantics and query answerin***g, Theoretical Computer Science 336, pp. 89-124, 2005.

[9] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, ***Consistency based diagnosis of configuration knowledge-bases***, Technical Report KLU-IFI-99-2, University of Klagenfurt, 1999.

[10] B.W. Kernighan and S. Lin, ***An efficient heuristic procedure for partitioning graphs***, The Bell system technical journal, 1970.

[11] T. Soininen, J. Tiihonen, T. Männistö, R. Sulonen, ***Towards a general ontology of configuration***, AI EDAM 12, pp. 357-372, 1998.