GMD Report 29

GMD –
Forschungszentrum
Informationstechnik
GmbH

Ulrich Geske,
Wolfgang Goerigk (Hrsg.)

Deklarative KI-Methoden zur Implementierung und Nutzung von Systemen in Netzen

Workshop der FG 1.1.1 der GI

zur KI-Konferenz '98

Bremen, 15.- 17. September 1998

August 1998

© GMD 1998

GMD – Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

In der Reihe GMD Report werden Forschungs- und Entwicklungsergebnisse aus der GMD zum wissenschaftlichen, nichtkommerziellen Gebrauch veröffentlicht. Jegliche Inhaltsänderung des Dokuments sowie die entgeltliche Weitergabe sind verboten.

The purpose of the GMD Report is the dissemination of research work for scientific non-commercial use. The commercial distribution of this document is prohibited, as is any modification of its content.

Anschriften der Herausgeber/Addresses of the editors:
Prof. Dr. Ulrich Geske
Institut für Rechnerarchitektur und Softwaretechnik
GMD – Forschungszentrum Informationstechnik GmbH
Rudower Chaussee 5
D-12489 Berlin-Adlershof
E-mail: Ulrich.Geske@gmd.de

Dr. Wolfgang Goerigk
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preußenstraße 1-9
D-24105 Kiel
E-mail: wg@informatik.uni-kiel.de

ISSN 1435-2702

Zusammenfassung

Dieser Band enthält die Proceedings der zum Workshop *Deklarative KI-Methoden zur Implementierung und Nutzung von Systemen in Netzen* angenommenen Arbeiten. Zunehmend wird von komplexen Softwaresystemen erwartet, daß sie mit lokal verteilten Ressourcen arbeiten können, oder daß sie über lokale oder Weitverkehrsnetze, z.B. im Internet, nutzbar sind. Einen besonderen Schwerpunkt setzt der Workshop auf die Bezüge zwischen KI-Methoden und -Systemen und dem Network-Computing. Dabei spielen Paradigmen wie Deklarativität, Constraint-Solving, Objektorientierung oder Agentenorientierung eine wesentliche Rolle.

Keywords: Network-Computing, KI-Methoden, Softwaretechnik, Deklarativität

Inhaltsverzeichnis

Vorwort

Bisherige Workshops zur Deklarativen KI-Programmierung standen unter den Thematiken: *Neuere Entwicklungen der deklarativen KI-Programmierung* (zur KI-Konferenz 1993 in Berlin), *Entwicklung, Test und Wartung deklarativer KI-Programme* (zur KI-Konferenz 1994 in Saarbrücken), *Objekt-orientierte KI-Programmierung* (zur KI-Konferenz 1995 in Bielefeld) und *Deklarative Constraint-Programmierung* (zur KI-Konferenz 1996 in Dresden).

Der diesjährige Workshop wird als eine gemeinsame Veranstaltung der Fachgruppe Deklarative KI-Programmierung des Fachbereichs Künstliche Intelligenz und der Fachgruppe Alternative Konzepte für Sprachen und Rechner des Fachbereichs Softwaretechnologie und Informationssysteme der Gesellschaft für Informatik e.V. durchgeführt. Er richtet sich an Teilnehmer aus Industrie und Forschung, die an Themen an der Schnittstelle zwischen Künstlicher Intelligenz auf der einen und Softwaretechnik und Programmierung auf der anderen Seite Interesse haben. Das Ziel des Workshops ist es, die Beziehungen zwischen der Entwicklung von KI-Methoden und den Erfordernissen an Softwaretechnik und Programmierung zu intensivieren, aber auch den Nutzen von KI-Methoden in der Softwaretechnik aufzuzeigen. Zunehmend wird von komplexen Softwaresystemen – auch KI-Systemen – erwartet, daß sie mit lokal verteilten Ressourcen arbeiten können, oder daß sie über lokale oder Weitverkehrsnetze, z.B. im Internet, nutzbar sind. Um KI-Methoden in derartige Systeme einbringen zu können, bedarf es neuer Ansätze, die eine Integration mit den zur Zeit üblichen Methoden z.B. des Network-Computing ermöglichen. Umgekehrt können Aspekte von KI-Methoden aus softwaretechnischer Sicht der Verbesserung und Erleichterung von Programmierung, Testung und Wartung von Programmsystemen dienen; Software-Entwurfsmuster beispielsweise entstammen häufig der Einsicht, daß gewisse Programmierparadigmen sich zur Lösung gewisser wiederkehrender Aufgabenstellungen besonders eignen. Paradigmen wie Deklarativität, Constraint-Solving, aber auch Objektorientierung oder Agentenorientierung sind wesentlicher Teil der Schnittstelle zwischen KI und Softwaretechnik. Dabei handelt es sich sowohl um die Nutzung von KI-Methoden in der Softwaretechnik als auch um softwaretechnische Aspekte von KI-Systemen. Einen besonderen Schwerpunkt setzt der Workshop auf die Bezüge zwischen KI-Methoden und -Systemen und dem Network-Computing. Gegenwärtig besteht ein besonderes Interesse an Werkzeugen oder Systemen, für die eine Nutzung über Netze im Vordergrund steht. Zur Abgrenzung sei bemerkt, daß ausdrücklich nicht Techniken der Verteilten KI im Vordergrund stehen. Die Nutzung von KI-Methoden und -Systemen über Netze meint nicht notwendigerweise, daß die Problemlösung selbst verteilt bearbeitet wird.

Dmitri Boulanger und Ulrich Geske behandeln im Artikel „Using Declarativeness of Logic Programming in Java Environment“ die Verwendung von Java, Oracle und des Logischen Programmiersystems Minerva für die Entwicklung GIS-orientierter Constraint-Datenbanken. Für die Realisierung der erforderlichen intelligenten Suchprozeduren werden Prolog-Klassen von Minerva verwendet. Minerva erzeugt Java-Code,

der die Netzwerkfähigkeit sichert. Ein weiterer Vorteil der Java-Umgebung ist das in verschiedener Hinsicht effiziente SQL-Interface. Das Konzept und eine erste Implementierung werden beschrieben.

Alexander Nareyek beschreibt in „Constraint-basierte Planung für Agenten in Computerspielen“ ein agenten-orientiertes Planungssystem, das durch die Anwendung constraint-basierter Techniken, lokalen Suchverfahren und eines explicit-timeline-Ansatzes Nachteile traditioneller Planungsalgorithmen in extrem laufzeitkritischen Anwendungen, wie z.B. Computerspielen, vermeidet.

Lothar Hotz und Michael Trowe stellen in „Parallel Programming in Common Lisp using Actors and Parallel Abstractions“ eine Erweiterung von Common Lisp vor, die die Definition paralleler Programme ermöglicht. Damit können Anwendungen unter Verwendung eines Workstation-Clusters parallel abgearbeitet werden.

R. Diekmann, W. Goerigk und U. Hoffmann beschreiben ein Konzept zur Nutzung von Standard-Werkzeugen zur Entwicklung graphischer Bedienoberflächen, konkret Tcl/Tk, für den Entwurf ereignisorientierter Benutzeroberflächen von Lisp-Anwendungen. Typische Anwendungen orientieren sich an dem Software-Entwurfsmuster „Model-View-Control“, wobei der Kern der Anwendung in Lisp modelliert ist. Standard-TCP/IP-Verbindungen erlauben zudem die Verteilung der Oberfläche und damit die Nutzung der Anwendung über Netze. einfache Portierung auf Anwendungsrechner.

Abdel Kader Diagne behandelt in „Die MORE Architektur. Ein objekt-orientiertes Architekturmodell für komplexe und netzwerkfähige Softwaresysteme” ein Architekturmodell, das die Entwicklung und Implementierung von komplexen und verteilten Systemen, sowie deren Bereitstellung im Netz erheblich erleichtert. Zum implementierten Werkzeug gehören u.a. eine generische Schnittstelle, die aus einem relativ protokollunabhängigen Kommunikationsmodul besteht und eine deklarativen Spezifikationssprache, die es ermöglicht, Module als Server im Netz zur Verfügung zu stellen.

Bertram Ludäscher, Rainer Himmeröder und Wolfgang May zeigen in ihrem Beitrag „Techniques and Rule Patterns for Declaratively Querying Web Data with FLORID“ wie mit Hilfe einer deduktiven, objekt-orientierten Sprache, Web-Daten in einer intuitiv verständlichen und deklarativen Art abgefragt werden können. Auf der Basis eines generischen Skeleton-Extraktors ist es beispielsweise möglich, automatisch die Hyperlinkstruktur von Web-Dokumentenmengen zu extrahieren.

An dieser Stelle sei schließlich noch den Organisatoren der KI-98 für die intensive Unterstützung bei der Vorbereitung des Workshops gedankt.

September 1998

Ulrich Geske
Wolfgang Goerigk
(*Hrsg.*)

Workshop-Zeitplan

Deklarative KI-Methoden zur Implementierung und Nutzung von Systemen in Netzen

Dienstag, 15. September 1998

Using Declarativeness of Logic in Java Environment

Dmitri Boulanger Ulrich Geske
GMD-FIRST, Rudower Chaussee 5, 12489 Berlin, Germany
email: {dmitri,geske}@first.gmd.de

Abstract Using the logic programming system Minerva, Java and DMBS Oracle for development of GIS-oriented constraint database systems is discussed. The central topic is a possibility to develop systems, which are approaching industrial standards. Namely, a technology for building applications, which need high quality graphical interfaces, which use data stored by in commercial SQL databases and which enjoy advantages of declarativeness of logic programming and constraint solving is the main goal. A number of important basic elements of the corresponding toolkit, which is currently under development, are introduced.

Key Words: logic programming, Java, object-oriented programming, declarative semantics, constraint database systems, persistence, GIS applications

1 Introduction

There exists a large class of important applications, for which a declarative logic-based knowledge representation is a crucial point. On the other hand, a very good adequate way of representing data does not always imply a *real* possibility to use this data for solving complex problems. Moreover, declarativeness creates very often serious efficiency problems. This observation is one of the main reasons to investigate tools, which enable to combine both the declarative way of reasoning about data and efficiency.

The class of applications in focus, is the so-called Geographical Information Systems (GIS). Currently this kind of applications constitute a large field of interest. GIS applications are exactly those, which need both the declarativeness combined with efficient algorithms, which are applicable on large data sets.

In GIS the initial data are geographically referenced data (spatial data), which are typically coming in the so-called *vector* mode. A vector is a sequence of geographical points. Each point of a vector is a pair, which usually indicates longitude and latitude. The vector format is commonly used to provide a structured representation of geometric objects in terms polygons or polylines, defined by the list (vector) of their vertices or segments. For instance, a popular public domain system GMT (Generic Mapping Tool) provides a large database in this format [4]. The high-resolution (0.2 km) world-wide GMT database has been created from 20 millions points (more than 100Mb), which represent shorelines, lakes, rivers and political borders.

GIS systems are not that new. A lot of them are used over a decade or more. Most of them are based on an architecture, which is coupled with a relational SQL DBMS and which provides special tools to manipulate spatial data. Among this tools, graphical user interfaces, which enable visualization of geographic data, play an important role. Also, it is clear that a good commercial DBMS must be a basis of any GIS, which is designed for non-trivial applications (cf. the above GMT database). Among various limitations of existing systems, the most serious one is that there is no sufficiently general high level query and

data manipulation facility, which is based on a satisfactory model for representing various geographic data. Because of complexity of geographic data, it is quite difficult to suggest a satisfactory data model. Moreover, there is a class of very important applications dealing with data of dimension more than 2, which make use of information that for instance is represented in a map and that may range over a period of time (temporal attribute). An example of such a *spatio-temporal* (or multi-dimensional) application is a monitoring of real estate objects in the given urban area. For multi-dimensional data there is only a few recent proposals, which exploit the so-called *constraint database* approach of [2, 15]. Recently an experimental system based upon constraint database model has been implemented and tested for a spatial database [14].

A constraint database model is essentially a logic-based *declarative* way of knowledge representation, which is a natural and simple generalization of the classical relational model of Codd [1]. The relational model is based on tables, which are finite collections of tuples. The constraint database model uses *generalized* tuples, which are defined as conjunctions of *atomic* (primitive) constraints. For instance, given the attribute variables x and y, the expression

$$t(x, y) \diamond \{x \leq y, x \geq 0, y \leq 1\}$$

defines a binary generalized tuple, which is a triangle with the vertices $(0, 0), (0, 1)$ and $(1, 1)$. Next, a generalized table (or a relation) is a finite collection of generalized tuples.

It is important to notice that the above is a simple but a very expressive and flexible construction. Indeed, the generalized tuple above is defined over the *linear* constraints, whose atomic constraint predicates $\leq /2$ and $+/2$ are defined over the rational numbers as a domain. Following the classical terminology of Lloyd [17], given a set of atomic constraint predicates, a domain and the truth values of constraint predicates constitute a *constraint model*. As it is mentioned in [14], the above binary linear constraint database model closely corresponds to the vector representation of GIS data. Most important, the linear constraint database provides a simple formal generalization of all possible variants of vector models. Indeed, the translation from one to another is based on the well-known operation, called *triangulation* (see, e.g. [20]). If a vector represents a non-convex polygon P, in order to translate it to a generalized relation over binary linear constraint database, the polygon P can be split in triangles. Since a triangle has a straightforward representation as a generalized tuple, the representation of P as a generalized relation is obtained by collecting the generalized tuples in the table (if P is convex, its translation into a single generalized tuple is immediate). However, in practice one needs more complex but again well-known algorithms (e.g. plotting some maps in GMT require the optimal Delaunay triangulation and the resulting network of triangles [4]).

Varying constraint model and/or extending its dimension one obtains another constraint database model, which possibly (or hopefully) captures needs of another more sophisticated application. An example is the spatio-temporal applications. An extension of the binary linear constraint databases with the third temporal attribute generates again the linear constraint database. It has generalized tuples for instance of the form

$$t(x, y, \tau) \diamond \{x \leq y, x \geq 0, y \leq 1, \tau \leq 3\}$$

which says that the above triangle $\langle (0, 0), (0, 1), (1, 1) \rangle$ is valid (or *was* valid) in the time interval before 3. The important features of this constraint database model are discussed in detail in [12]. It is observed that such an extension enables to express interoperability problems in spatio-temporal databases.

As another illustration, extending the linear constraints to polynomial equations and inequations and using points in the vector model for generation of polynoms, one obtains a more elaborated tool to represent borders of spatial objects. In this way it is possible to bridge the gap between different but closely related at the semantic level databases since generalized tuples have a potential to capture very different but semantically related data within a single framework. With usual software technology, access of data from different databases is done by means of interfaces, which has to be created within an application program. On the contrary, generalized tuples being used to wrap database in focus, provide a generic interface with transparent properties. This is one of the most attractive consequences of the above declarative logic-based approach.

The difference between the above mentioned traditional interfaces in application programs and wrappers, which are based on generalized tuples, is very important. Indeed, if an application is build using special interfaces to access different databases, it is impossible to perform any kind of optimization or correctness tests, which cross interface borders. On the contrary, if generalized tuples are used instead, the known relation between constraint models of the database wrappers can be used to perform correctness test and query optimization, which goes through the boundaries of wrappers. Given a constraint solver, which is often capable to detect inconsistency of a constraint set without instantiating much of its atomic constraints, it is possible to stop query evaluation only using constraints of a database warper rather than the actual data.

Consider one real problem. Its solution was very important for creation the GMT database [21]. The database has been obtained from two large public-domain data sets, the World Data Bank II (WDB) and the World Vector Shoreline (WVS). The WVS data are more accurate but has no lakes, rivers and political borders, which are presented in WDB. Therefore, the GMT database has been developed from WVS using data from WDB as a supplement. In order to use this data in various applications, spatial data must be arranged as *closed* polygons but the data in both VWS and WDB were only unsorted line segments: there was no information included, which indicated which segments belong to the same polygon. Moreover, since not all segments joined exactly, it was necessary to find out all possible combinations and choose the simplest combinations. Also, when the data from WVS and WDB were combined, a lot of polygons were duplicated since most of features in WVS were also in WDB. Since resolution of data were different, it was nontrivial to decide which polygon to ignore.

The above is only a small part of the list of problems which had to be solved. It has been mentioned to show that the linear binary constant database model fits this problem in a sense that all the above requirements for data filtering and transformation can be formalized in an elegant and declarative way as a number of linear constraint queries. These queries are supposed to be applied on the database, which is a union of WVS and WDB. The problem is that WVS and WDB together have more than 100Mb of binary data (or 20 million points). This immediately rises a question how to evaluate constraint queries over such databases and a more general question: "How an architecture of a constraint database system should be organized to be useful for complex GIS applications such as the above problem?"

As a conclusion of the above discussion, it should be mentioned that declarative model-based approach for constraint databases have a number of remarkable features. Among them is a uniform formalism for a large field of GIS applications, which includes multi-dimensional databases. Also this includes interoperability of GIS databases, solving complex data consistency problems and design of expressive problem-oriented query languages.

So far only advantages of constraint databases have been emphasized. On the other hand,

there exists the well-known drawback of systems, which implement high level declarative data models: sometimes even very simple queries require very long time to be evaluated.

2 Motivations of the Approach

Above in Section 1 we have introduced a number of well-know motivations for using a declarative logic-based formalism for development of constraint databases. It allows to see most of GIS databases as collections of generalized relations. So far we didn't bother approaches for implementation of of constraint databases.

Currently there exist only a few implementations of constraint databases. One of the most well-known is described in [13, 14]. Let us first discuss basic elements of its architecture and then explain our approach.

The constraint database in [13] is based upon binary linear constraint database model, which has been discussed in Section 1. The central element of the system architecture is the front-end constraint database query language. The system is build on top of the O_2 object-oriented database system and has been tested on a real geographical data. The main targets of the implementation were to investigate the feasibility of constraint database model for GIS application and develop a reasonably efficient query evaluation engine [14].

The system has been developed keeping in mind complex GIS applications. Therefore, a number of "pragmatic" interfaces have been implemented. For instance, the *data loading* interface enables to load data, which are coming in vector mode. Also, a special *data display* interface enables to visualize spatial data. On the other hand, a dedicated linear constraint algebra has been developed. It extends the standard relational algebra, which is the basis for relational operators of most of existing SQL-DBMS such as Oracle [11]. The most well-known relational operators are join, projection, etc. Similar *linear constraint operators*, which significantly extend the above relational operators and which actually form a constraint query evaluation engine of the system, have been developed and implemented.

Following the above approach, given a constraint database model M, a possibility to develop its algebra and the corresponding operators having a reasonably efficient implementation, is a crucial point when considering the model M as "pragmatic". For instance, in Section 1 we have mentioned polynomial constraint database model. Taking into account a number of rather complex problem, which are reported in [14], a design and an implementation of an algebra and efficient operators for a non-linear model is a very difficult task. However, polynoms are used in many existing CAD/CAM systems. This shows that the algebraic-based approach for an implementation of constraint databases have limitations with respect to the main advantages of constraint databases in Section 1. For instance, it is difficult to expect that there exists an algebraic-based approach, which can be used to in an implementation of various constraint-based wrappers by means of "just changing the underlying constraint database model".

It seems that the above is a reflection of very similar problems in deductive databases. On the most well-known deductive database systems is LDL++ [23]. It also relies on the so-called *bottom-up* evaluation schema, which needs a kind of algebra and corresponding operators. A more recent system, which provides most of declarative features of LDL++ (the LDL++ sets are missed) is the XSB-prolog [22]. The XSB-prolog has much more efficient *top-down* (or "imperative") WAM-based implementation, which exploits the so-called tabulation mechanism and a dedicated "declarative" interface with databases. A number of experiments

have shown a remarkable efficiency of the system. In our approach for constraint databases we follow this this way.

There exists another crucial ingredient of constraint systems, called *constraint propagation*. Most of logic constraint-based programming systems, which enjoy a top-down prolog-based evaluation mechanism, exploit it ([18, 19]). Constraint propagation is the process of reducing a constraint satisfaction problem to another one that is equivalent but simpler to solve. The algorithms that achieve such a reduction usually aim at reaching some form of *local consistency*, which denotes some property approximating *global consistency*. For some applications such an enforcement of local consistency is already sufficient for finding a solution in an efficient way or for determining that none exists. In some other cases this process substantially reduces the size of the search space which makes it possible to solve the original problem more efficiently by means of some *search* algorithm. For instance, as it was mentioned in Section 1, this feature might be very important for an optimization of applications, which use constraint-based wrappers as interfaces with different databases. It is clear that it is rather difficult to exploit constraint propogation in top-down frameworks.

To summarize all above, we consider generalized tuples as very important high-level declarative concept, which enables to capture semantics of a large field of GIS applications. Moreover, we are interested in a flexible implementation of generalized tuples, which can enjoy different constraint models still having a reasonable efficiency. On the other hand, it seems that only an "imperative" implementation with imperative front-end interfaces can meet this goal. Therefore, as an implementation basis we have chosen two "non-declarative" systems: Minerva [5] and Java [9, 10, 8]. Minerva is a commercial implementation of standard Prolog, which enables a very smooth integration in Java environment. In fact, the compiled Minerva program is a "pure" Java binary code. This, as it is, has a lot of advantages because Java is platform independent and provides a rich environment for application development. Using Java will make components of our toolkit below more widely available and easier to use. Also, applications, which significantly exploit Internet, Java provides a simple and reliable background. A more detailed discussion is in the next section.

3 A Constraint Database Toolkit

It is noteworthy to mention that recently an integration of Prolog and Java attracts considerable attention. Several implementations are reported [6, 7]. Therefore, let us first to emphasize some important features of Java, which are relevant to the above discussion.

First of all, from the point of view of software engineering a flexible efficient implementation of constraint databases means providing a foundation of reusable software to speed the development of a class of GIS applications. In fact, generalized tuples can be seen as an *abstract* specification of these application. As it was mentioned above, implementation of a GIS application very often needs rather complex but *well-known* algorithms, e.g. triangulation of polygons. Also, an examination of the above implementation of the constraint database in [13, 14] reveals that it exploits many well-known algorithms, which have been *re-implemented* to fit specific environment of the system. This is exactly what we would like to avoid. The two-dimensional geometry, which is a basis for implementation of binary linear constraint data model, is a very well-known field, for which nearly the best algorithms are known. Therefore, an approach should focus on delivering these algorithms to the right place rather than to develop or re-implement them. In other words, an implementation of

the generalized tuples should help to avoid developing applications from "scratch" to solve a specific problem without the benefit of a foundation of tested software. Relying on a completely "self-made" implementation is time consuming and expensive because we would like to have a flexible one. On the contrary, by simply reusing the many proven programs, a particular constraint database model can be build more efficiently and its applications will be more reliable.

Java has very flexible mechanisms to facilitate that. Namely, efficiency and flexibility can be achieved through the extensive use of Java *interfaces*, which allows to abstract algorithms and data structures. All algorithms should only use these interfaces when accessing internal objects. It allows nearly error-free substitution of any object that provides the appropriate interface. The appropriateness of the object interface is checked by the Java compiler. Our experience indicates that such checking can be made very reliable since Java is a strictly typed language. One example of such an abstracted data structure is class of different coordinate systems, which are used in GIS applications. Using such an interface, which abstract features of geographic coordinates, one can only use a single peace of software to process different formats of vector data.

Another important aspect is a flexible usage of available "core" software for different constraint data models. Again, using Java interfaces, one can implement access of objects, which are specific for a particular constraint model, only through intermediate objects that extend the generic data type, which in our settings is the binary spatial constraint data model. In this way an algorithm uses a specialized object.

A significant motivation of the above approach was borrowed from the architecture and philosophy of OR-Objects package [3]. It is a library of Java classes for developing Operations Research applications, which includes data structures and algorithms for developing custom solutions as well as many classical algorithms. The package ranges from 2-dimensional geometry to linear and nonlinear programming, various algorithms on graphs, etc. The architecture of the package exactly follows the above discussion. As an illustration, the geometry sub-package enabled us to implement very quickly a flexible generic interface to manipulate spatial data, which are coming in vector mode.

Another recent aspect of Java is its interface to SQL-databases, which enables to see most of commercial databases in a uniform way. An important element is the the so-called *thin drivers*, which allows to access SQL-databases via TCP/IP protocol. Since thin drivers are "pure Java" software, an access of databases becomes extremely simple and reliable. Moreover, our experience with the Oracle driver indicate that it is very efficient. As it follows from above, a reliable persistent platform is a crucial point for GIS applications.

Finally, the capability to store and retrieve Java objects is essential to building GIS applications. The key to storing and retrieving objects is representing the state of objects in a *serialized form* sufficient to reconstruct the object(s). Combined with a possibility to save large binary files in Oracle database, one obtains a tool for implementing essential ingredients of a persistent object-oriented environment. This allows to exploit standard DBMS as a persistent object oriented environment. From pragmatic point of view, this has important advantages compared with a special DBMS, which is used in [13, 14]. Indeed, using standard DBMS allows to extended already existing systems with new features. Again, this is aimed at reusing existing GIS databases.

It is clear that a development of an architecture of package for complete generalized tuples data model is a very difficult problem. Therefore, our current goal is only binary spatial constraint databases, which, however may use rather sophisticated set of constraints.

Our current goal is to provide the range of constraint models, which have an efficient and flexible support from the the above OR-objects package [3]. This, for instance, extends the linear binary spatial model with various graph constraints. Such an extension allows to deliver critical path planning algorithms, vehicle routing algorithms and traveling salesman planning into the context of GIS systems. To achieve our goal, we have started with a prototype whose importance in an understanding important aspects of an architecture of a package for spatial generalized tuples.

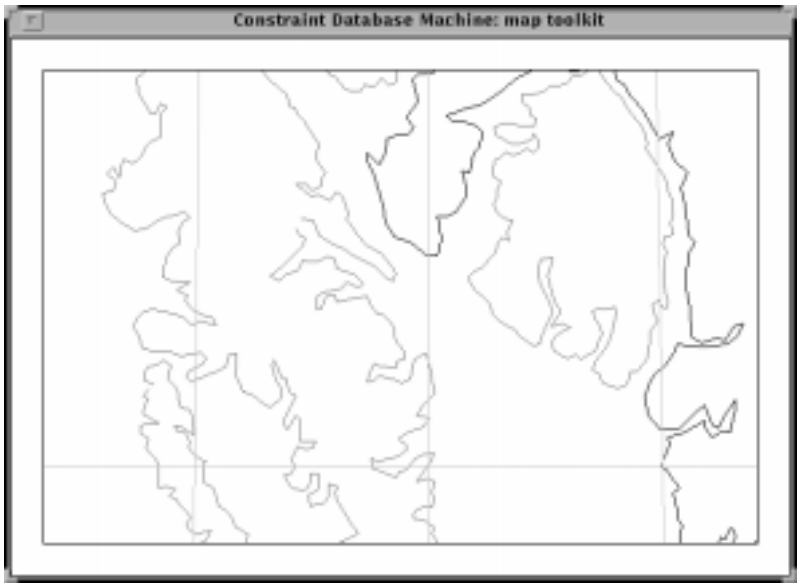

Fig.1: *GIS Window*

First of all, we have implemented a sub-package for building classical GIS interfaces (see Figure 1 as an illustration). This interface enables allows to deal with abstractions of digital maps in vector mode. The main functionality inclused access of maps in different formats, display them using stack-based zoom/dezoom mechanism, allocate icons, save regions as objects in database, convert the chosen polygons into binary spatial constraints. This functionality rather close to that in [13, 14].

Next, another interactive constraint solving interface has been implemented (see Figure 2). This is also a generic abstract interface to provide an interactive constraint solving. Namely, its importance in visualizing constraint propagation progress and possibility to correct it in an interactive way. Currently it was tested with graph layout constraint solver, which only allows a simple geometry of spatial objects.

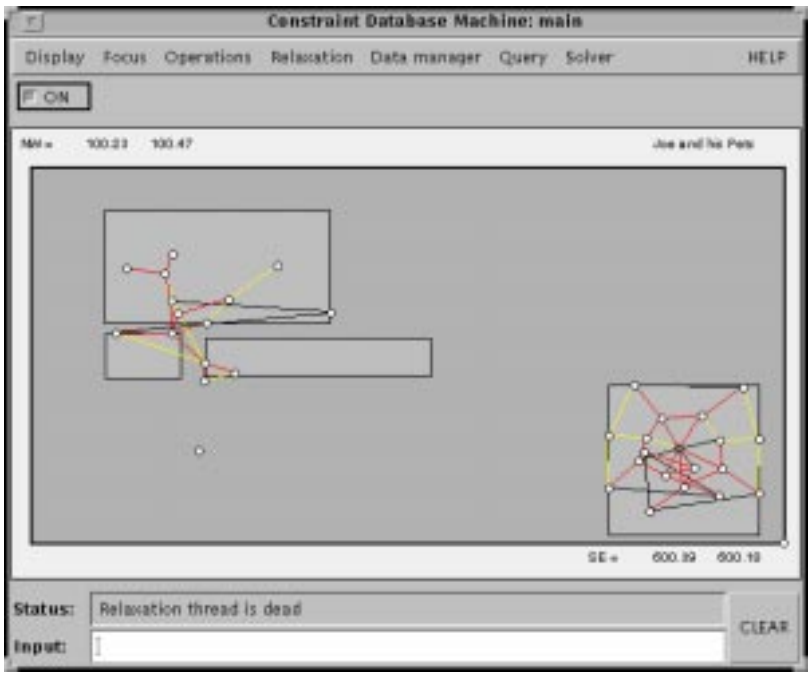

Fig.2: *Constraint Solver Window*

Both the above interfaces are implemented as a generic packages, which can be specialized and extended to fit particular requirements of constraint database in hand. Such a generic architecture was the main goal.

An important current research goal is a development of a generic interface to build a constraint solving/propagation mechanism using algorithms from the OR-objects package. Also, intended applications are expected to require various combinatorial search algorithms. Therefore, we are studying a possibility to use Prolog classes of Minerva [5] to bring such algorithms in Java environment. This is a central element of the corresponding package. Another possibility could be to use commercially available logic-based constraint systems as CHIP [18] or ECLiPSe [19] via a foreign interface or a database interface. However, this significantly complicates the package.

A constraint database model have very strong model-theoretic background, which gives rise to various static analysis algorithms. Such algorimthms may drastically improve run-time efficiency of applications. Therefore, another our goal for future is applying the results in [16] for static analysis and optimization of complex GIS applications. We plan to develop so-called "abstract" constraint data models, which can be used as a safe (correct) abstraction of real constraint models. Importance of such abstract models in using them as approximations in to speed up constraint solving procedures.

As a conclusion, it is clear that exploiting the full potential of generalized tuples constraint model is a non-trivial task, which needs research and experiments with applications.

Therefore, , in parallel we are going to implement a couple of application. One will use the GMT database and another will based upon high-resolution complex map of the city of Berlin.

4 Conclusion

An approach for an implementation of constraint databases for GIS applications has been discussed. Our concept of constraint database uses the well-known formalism, called generalized constraint tuples. Constraint tuples is a high level declarative logic-based data model, which is considered as a single unifying framework to cover a wide range of GIS applications. As it was explained, one of the main consequences of the declarativeness of this high-level logic-based data model is that it can be used to build GIS applications in a systematic and reliable way. It is used as main underlying philosophy of our approach.

One of our goals is to obtain a flexible implementation of generalized tuples, which can be used for developing complex GIS applications. This is a a very difficult problem if such an implementation attempts to provide a front-end interface, which, for instance is a high level declarative query language. In the above we use another approach, which is aimed at implementation of a package or toolkit, which delivers basic blocks to build various constraint database models. A straightforward usage of such a toolkit can be achieved simply specializing available interfaces. More advanced users can extend the package elements, add new problem-specific interfaces and special algorithms to deal with sophisticated constraint models.

In our implementation we exactly follow the above concept. Namely, our GIS-oriented interfaces are build upon already existing geometrical and problem-solving packages, which provide essential "low-level" ingredients such as coordinate systems and geographical projections, geometry, liner algebra and linear programming, optimization algorithms on graphs, etc. In fact, the role of the core software of the toolkit is putting such algorithms to-gather to obtain an implementation of a particular spatial constraint database model. To achieve this goal we significantly rely on the core Java environment, which provides abstract classes and interfaces.

Another role of the core software of the toolkit is managing persistent object-oriented store, which is can be build on the top of commercial SQL DBMS. This enables a smooth extension of already existing GIS databases or simply use them as components of new systems.

Currently our goal is an investigating of an architecture of such a toolkit. To do that, a couple of sub-packages have been implemented: map toolkit and a simple graph layout constraint solver. Both of them exploit an interface with SQL databases. This rather modest implementation has brought the following functionality: we have a flexible interactive graphical interface to deal with geographical data in vector mode and an interactive graph layout interface, which uses simple spatial constraints.

References

[1] Codd, E. A relational model of data for large shared data banks, *Communications of ACM, 1970, 13(6), 377-387.*

[2] P. Kanellakis, P.C. Kuper, and P. Revesz Constraint Query Languages,*Journal of Computer and System Science 1995, 51(1), 26-52.*

[3] OR-Objects: a library of Java classes for Operations Research Applications, *http://OpsResearch.com/OR-Objects/index.html.*

[4] GMT: Generic Mapping Tool, *http://www.soest.hawaii.edu/soest/gmt.html.*

[5] IFProlog Minerva Reference Manual, *http://www.ifcomputer.com/Products/MINERVA/.*

[6] Demoen, B., Tarau, P. jProlog, a Prolog to Java compiler, *http://www.cs.kuleuven.ac.be/ bmd/PrologInJava.*

[7] Kino, N. JIPL: Java Interface for Prolog, *http://Prolog.isac.co.jp/jipl.*

[8] Java Home Page, *http://java.sun.com/.*

[9] Gosling, J., McGilton, H. The Java Languge Environment. A White Paper, *A Sun Microsystems, Inc.*

[10] Kramer, D. The Java$^{\text{tm}}$ Platform. A White Paper, *A Sun Microsystems, Inc.*

[11] Oracle Database Management System, *http://www.oracle.com/.*

[12] Chomicki,J., Revesz., P Constraint-based Interoperability of Spatiotemporal Databases, *Interop'97, International Conference on Interoperating Geographic Information System, Santa Barbara, CA, 1997 (submitted to GEOInformatica).*

[13] Grumbach,S., Rigaux,P., Scholl,M., Segoufin, L. DEDALE, A Spatial Constraint Database, *Int. WS on Database programming languages, DBPL'95, 1995.*

[14] Grumbach,S., Lacroix, Z. Computing Queries on Linear Constraint Databases, *Fifth Int. WS on Database programming languages, DBPL'97, 1997.*

[15] Gaede,V., Wallace, M. Database systems, *LNCS* **1191**, *1996, 1-52.*

[16] Gallagher,J., Boulanger,D., Sağlam,H. Practical Model-Based Static Analysis for Definite Logic Programs, *Proc. 1995 ISLP, 1995, 351-365.*

[17] Lloyd,J. Foundations of Logic Programming, *Springer, 1987.*

[18] Chip User's Guide, *Cositec SA. Orsay, France 1997.*

[19] ECLiPSe 3.4 User's Guide ECRC Common Logic Programming sysem, *ECRC, Munich, Germany 1994.*

[20] Tarjan, R.E., van Wyk, C. An $O(n \log\log n)$-Time Algorithm for Triangulating a Simple Polygon, *SIAM Journal of Computing,1988, 17(1), 143-178.*

[21] Wessel, P., and W. H. F. Smith, A global self-consistent, hierarchical, high-resolution shoreline database, *J. Geophys. Res., 1996, 101, 8741-8743.*

[22] Sagonas,K., Swift, T., Warren, D. The XSB Programmer's Manual, *Version 1.6.*

[23] Arni,N., Ong,S., Tsur,S., Zaniolo,C. *LDL++*: A Second Generation Deductive Database System, *Technical Report, MCC Corporation, 1993.*

Constraint-basierte Planung für Agenten in Computerspielen*

Alexander Nareyek
GMD — FIRST
Forschungszentrum Informationstechnik GmbH
Forschungsinstitut für Rechnerarchitektur und Softwaretechnik
Rudower Chaussee 5
D - 12489 Berlin
alex@first.gmd.de

Zusammenfassung

Für die Führung von computergesteuerten Gegnern/Charakteren/Einheiten (Agenten) in Computerspielen steht in der Regel nur sehr wenig Rechenzeit zur Verfügung, da die CPU mit Grafik und Spiele-Engine bereits stark ausgelastet ist. Traditionelle Planungs-Algorithmen berücksichtigen derartige Laufzeitrestriktionen nur unzureichend. Umplanungen aufgrund des hochgradig dynamischen Umfeldes sind ebenso selten realisierbar. Des weiteren sind zur Koordination der Agenten komplexe temporale Abhängigkeiten in Betracht zu ziehen, mit denen die meisten Planungssysteme nicht umgehen können. Diese Arbeit stellt ein Planungssystem vor, welches durch die Anwendung constraint-basierter Techniken, lokaler Suchverfahren und eines explicit-timeline-Ansatzes die Beseitigung der angeführten Mängel anstrebt.

1 Einführung

Die vorliegende Arbeit ist Teil des EXCALIBUR-Projektes. Das Ziel des Projektes ist die Entwicklung einer generischen Architektur für autonom operierende Agenten innerhalb komplexer Computerspiele. Diese Agenten müssen die richtigen Aktionen zur Erreichung gegebener Ziele ausführen und ihr Verhalten neuen Situationen anpassen.

Ein wichtiger Aspekt eines Agenten ist die Art und Weise, wie sein Handeln bestimmt wird. Soll der Agent nicht auf rein reaktives Handeln beschränkt sein, wird ein Planungssystem benötigt, welches einen aus Einzelaktionen bestehenden Handlungsplan für den Agenten erstellt. Das Gebiet der Planung ist bereits seit längerem ein lebendiger Forschungsbereich, und es existiert eine große Bandbreite von Planungssystemen, wie z.B. STRIPS (Fikes & Nilsson 1971), UCPOP (Penberthy & Weld 1992) oder PRODIGY (Veloso et al. 1995).

Um den harten Echtzeit-Anforderungen gerecht zu werden, muß für den Agenten jederzeit ein Plan zur Verfügung stehen. Dieser Plan kann iterativ über die Zeit verbessert werden. Nachdem in Abschnitt 2 die Grundlagen einer constraint-basierten Repräsentation für die Planung dargelegt werden, geht Abschnitt 3 auf die iterative Erstellung/Verbesserung der Pläne ein. Die konkreten Plankomponenten werden in Abschnitt 4 erläutert.

*Das zugrundeliegende Forschungsprojekt EXCALIBUR wird mit Mitteln der Deutschen Forschungsgemeinschaft (DFG), der Conitec Datensysteme GmbH und Cross Platform Research Germany (CPR) unterstützt.

2 Constraint-basierte Repräsentation

Die Constraint-Programmierung beschäftigt sich insbesondere mit der Behandlung kombinatorischer Suchprobleme und bietet sich somit als Rahmen für die Formulierung und Lösung von Planungsproblemen an.

Die Spezifikation eines Constraint-Satisfaction-Problems besteht aus einer Menge von Variablen $X = \{X_1, \ldots, X_n\}$, wobei jede Variable eine Domäne $D_1, ..., D_n$ besitzt, und einer Menge von Constraints $C = \{C_1, ..., C_m\}$ über diesen Variablen. Constraints sind Relationen zwischen Variablen, die die möglichen Wertebelegungen einschränken.

Im EXCALIBUR-Modell werden sogenannte *globale Constraints* (Puget & Leconte 1995) verwendet, anstatt das Problem über Standardverfahren wie lineare Ungleichungen und dergleichen zu kodieren. Ein globales Constraint ist ein Ersatz für eine Menge von low-level Constraints, wobei zusätzliches domänenspezifisches Wissen die Anwendung spezialisierter Datenstrukturen und Algorithmen erlaubt, um die Suche nach einer konsistenten Wertebelegung zu steuern bzw. zu beschleunigen.

Durch die globalen Constraints kann der derzeitige Plan evaluiert werden. Jedes Constraint berechnet dazu einen lokalen Wert, der die derzeitige Inkonsistenz des Constraints angibt. Dieser Wert kann dabei nicht nur die Verletzung des Constraints ausdrücken, sondern auch den Abstand zu zu präferierenden Lösungen (Zielen). Die einzelnen Werte werden mittels einer *objective Function* zu einem Gesamtwert zusammengefaßt, der als Maßstab für die Qualität des Plans dient.

Die zweite Aufgabe der globalen Constraints ist es, Vorschläge zur Verbesserung des derzeitigen Plans zur Verfügung zu stellen. Solche Vorschläge können in der Änderungen von Kontrollvariablen, wie z.B. dem Beginn von Aktionen, bestehen. Genausogut können aber auch ganze Aktionen ein- oder ausgeplant werden, Objekte in mehrere Objekte aufgeteilt werden, etc. In der derzeitigen Implementierung wird bei jedem iterativen Verbesserungsschritt der Vorschlag des Constraints mit der höchsten Inkonsistenz gewählt.

3 Satisfaction durch Local Search

Die Konstruktion eines Planes erfolgt durch die iterative Verbesserung eines initialen Plans (z.B. des leeren Plans). Hierdurch steht jederzeit ein Plan zur Verfügung, auf den die Spiele-Engine zur Steuerung des Agenten zurückgreifen kann. Die iterativen Verbesserungen sorgen für eine kontinuierliche Verbesserung der Planqualität und passen den Plan an Situationsänderungen an (siehe Abbildung 1).

Ein Problem von lokalen Suchverfahren ist deren Unvollständigkeit, wodurch es möglich ist, in lokalen Optima oder auf Plateaus steckenzubleiben. Eine Vielzahl von Methoden wurde entwickelt, um den lokalen Optima und Plateaus zu entkommen, wie z.B. Neustarts, Zufallsänderungen und Tabu-Listen. Unsere bisherigen Experimente deuten allerdings darauf hin, daß die Verwendung globaler Constraints den Suchraum bereits so gut strukturiert, daß derartige Erweiterungen die Suche eher behindern als beschleunigen. Eine Ausnahme bildet eine Auswahl mit einer Wahrscheinlichkeitsverteilung entsprechend den Verbesserungsaussichten von Alternativen, allerdings nur in speziellen Entscheidungsfällen. Hinzu kommt die hohe Dynamik eines Computerspieles, durch die derartige Erweiterungen eine untergeordnete Rolle spielen, da der Suchraum sich schnell ändert und generell weniger in die Tiefe gehende Verbesserungen möglich sind.

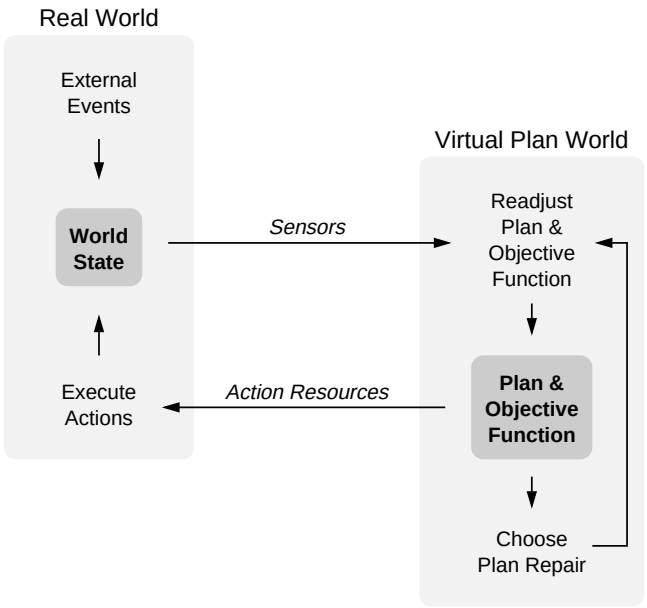

Abbildung 1: Situationsänderungen und Planverbesserung

4 Plankomponenten

Das EXCALIBUR-Modell ist vom Szenario des Job-Shop-Scheduling inspiriert, welches sich als erstklassige Anwendung für die Constraint-Programmierung erwiesen hat. Infolgedessen sind die Komponenten des Modells in einer Terminologie von Ressourcen, Aufgaben und Constraints beschrieben.

4.1 Aktionen, Aktionsaufgaben und Aktionsressourcen

Die Ausführung einer **Aktion** (wie EAT PEANUT) kann in verschiedene **Aktionsaufgaben** untergeteilt werden. Jede dieser Aktionsaufgaben benötigt eine **Aktionsressource** für ihre Ausführung. Beispielsweise erfordert die Aktion EAT PEANUT Aktionsaufgaben mit den Aktionsressourcen MOUTH und LEFT HAND oder RIGHT HAND. Abbildung 2 zeigt die Zuweisung der Aktionsaufgaben zu Aktionsressourcen. Innerhalb einer Aktionsressource sorgt ein **Aktionsressource-Constraint** dafür, daß sich einzelne Aktionsaufgaben nicht überlappen.

Die Struktur einer Aktionsaufgabe enthält eine **Beschreibung**, welche Informationen zur Ausführung der Aufgabe mittels der Aktionsressource beinhaltet. Zwei weitere Komponenten spezifizieren den **Beginn** und das **Ende** der Abarbeitung der Aktionsaufgabe. Beginn und Ende sind Entscheidungsvariablen, besitzen jedoch durch den Ansatz der iterativen Verbesserung stets eine eindeutige Wertzuweisung.

Jede Aktion besitzt ein **Aufgaben-Constraint**, durch das Beginn und Ende der Aufgaben der Aktion in spezifische Relationen gesetzt werden können. Beispielsweise sollen die Aktionsaufgaben der Aktion EAT PEANUT zu den gleichen Zeiten beginnen und aufhören, und das Ende soll vier Sekunden nach dem Beginn erfolgen. Die Rolle des **Vorbedingungs-Constraint**s einer Aktion wird im nächsten Abschnitt erläutert.

Um die Rolle der Constraints zu verdeutlichen, wird im folgenden die derzeitge Implementierung des Aufgaben-Constraints skizziert:

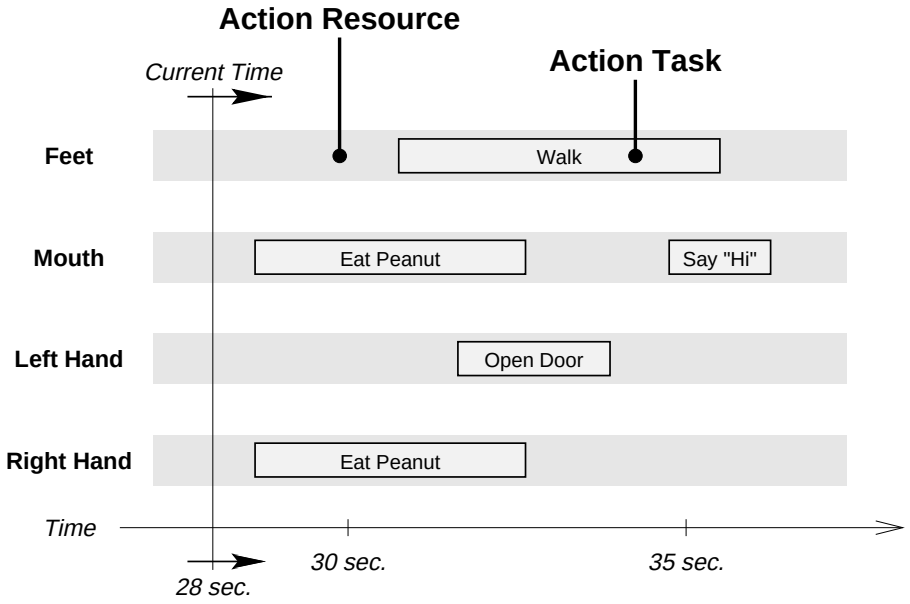

Abbildung 2: Zuweisung von Aktionsaufgaben zu Aktionsressourcen

- Das Aufgaben-Constraint verwaltet eine Menge von Aufgaben-Relationen. Jede Aufgaben-Relation wird durch eine lineare Ungleichung $X_1 \otimes X_2 + c$ repräsentiert, mit X_1 und X_2 als Beginn- oder Endzeitpunkten von Aufgaben, $\otimes \in \{<, \leq, =, \geq, >\}$ und einer Verschiebekonstanten c.
- Für jede nicht erfüllte lineare Ungleichung wird die Länge der minimal nötigen Verschiebung einer der Ungleichungsvariablen, welche die Ungleichung erfüllt werden läßt, zur Inkonsistenz des Aufgaben-Constraints addiert.
- Falls das Aufgaben-Constraint zur Verbesserung des derzeitigen Plans ausgewählt ist, wird eine inkonsistente Ungleichung selektiert und eine minimale Verschiebung einer Aufgabe durchgeführt, welche die Ungleichung erfüllt werden läßt.

4.2 Zustandsaufgaben und Zustandsressourcen

Vor- und Nachbedingungen von Aktionen werden durch Constraints zwischen Arbeitsaufgaben und Zustandsressourcen und Zustandsaufgaben realisiert.

Eine **Zustandsressource** gleicht einer Aktionsressource. Allerdings verwaltet sie keine aktiv geplanten Aktionen, sondern die Entwicklung einer spezifischen Eigenschaft der Umgebung oder des Agenten selbst[1]. Zum Beispiel kann eine Zustandsressource OWN PEANUT mit einer Bool'schen Belegung für jeden Zeitpunkt Informationen über den Besitz einer Erdnuß bereitstellen (siehe Abbildung 3).

Der Status einer Zustandsressource kann die Anwendbarkeit von Aktionen einschränken. Um eine Aktionsaufgabe EAT PEANUT auszuführen, muß die Erdnuß vorher im Besitz des Agenten sein. Derartige Relationen werden durch das Vorbedingungs-Constraint einer Aktion gewahrt, welches den Zustand von Zustandsressourcen abfragen kann.

[1] Zustandsressourcen können als *fluents* interpretiert werden.

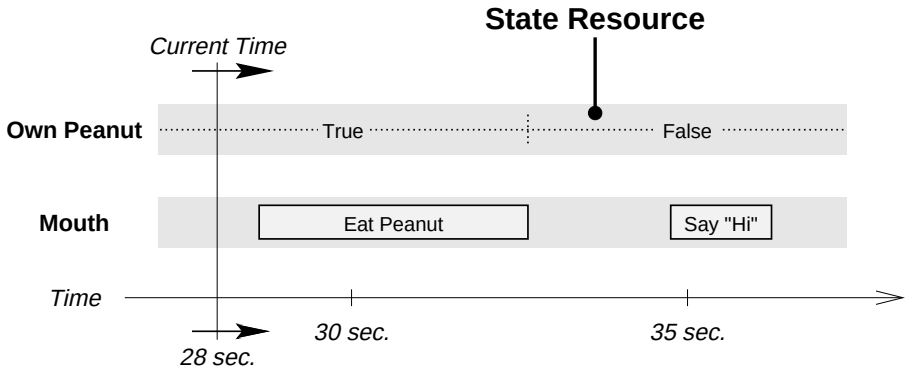

Abbildung 3: Eine Zustandsressource

Die Effekte von Aktionen sind schwieriger darzustellen, da mehrere Aktionen und Ereignisse synergetische Auswirkungen haben können. Beispielsweise kann eine Zustandsressource HUNGER mit einer Domäne von natürlichen Zahlen gleichzeitig von einer verbessernden Aktion EAT PEANUT und einer verschlechternden Aktion WALK beeinflußt werden.

Es ist die Aufgabe von **Zustandsaufgaben**, die Effekte einer Aktion zu realisieren. Beispielsweise ist eine Zustandsaufgabe der Aktion EAT PEANUT für einen verringernden **Beitrag** von -3 zur Zustandsressource HUNGER während der Ausführung der Aktion verantwortlich (siehe Abbildung 4). Jede Zustandsressource hat einen spezifischen Abbildungsmechanismus, welcher die Beiträge der Zustandsaufgaben auf Werte der Domäne der Zustandsressource abbildet. Im Falle der Ressource HUNGER erfolgt der synergetische Effekt durch die einfache Addition der einzelnen Gradienten. Ein zusätzliches **Zustandsressource-Constraint** sorgt für die Einhaltung der internen Konsistenz, wie z.B. Domänenrestriktionen.

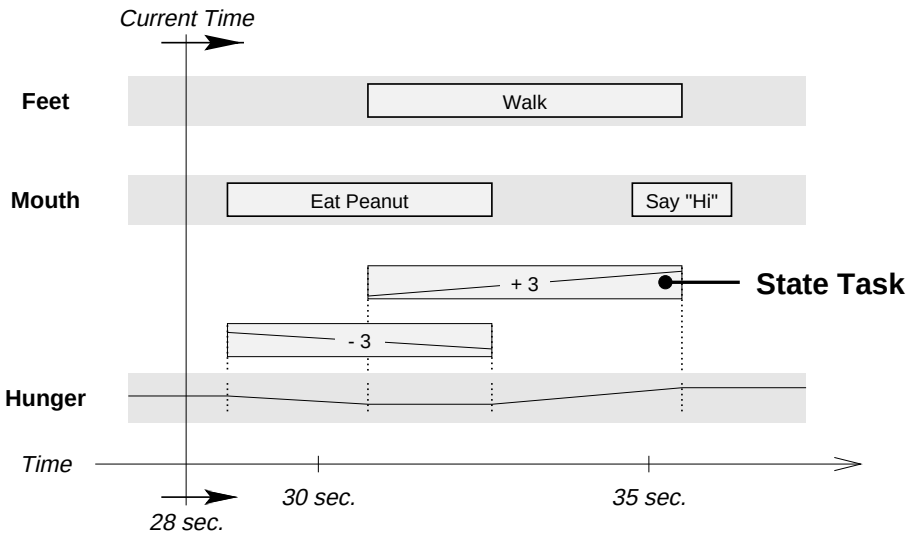

Abbildung 4: Der Abbildungsmechanismus von Zustandsressourcen

Es kann weitere Effekte geben, welche durch synergetische Effekte innerhalb einer Zustandsressource auftreten. Aus dem Einfüllen von Wasser in eine Badewanne kann z.B. eine Überschwemmung resultieren und den Boden des Badezimmers rutschig machen. Die einzelnen Aktionen können für diese weiterführenden Effekte keine zusätzlichen Zustandsauf-

gaben zur Verfügung stellen, da eine Aktion nur die eingeschränkte Sicht ihrer Beiträge hat. Um die Effekte spezifischer Zustände einer Zustandsressource abzubilden, werden deshalb **Abhängigkeiten** verwendet. Diese Abhängigkeiten sind spezielle Aktionen, welche vom Agenten nicht aktiv kontrollierbar sind. Erwartete externe Ereignisse können durch diese Abhängigkeiten ebenfalls modelliert werden.

4.3 Objekte, Referenzen und Sensoren

Es kann in der Welt mehr Dinge als eine Erdnuß zu essen geben. Natürlich wäre es möglich, für jedes dieser Objekte eine Aktion zum Essen zur Verfügung zu stellen. Eine flexiblere Lösung ist die Definition einer generelleren EAT-Aktion, deren Aufgaben **Referenzen** zu den betroffenen Ressourcen besitzen.

Eventuell besitzt der Agent zwei Ernüsse, eine kleine und eine große. So lange die Objekte der Welt nicht eindeutig sind, wären die Zustandsressourcen der Erdnüsse ununterscheidbar. Die Anwendung der EAT-Aktion könnte dann die große Erdnuß verschwinden lassen, während der Sättigungwert der kleinen Erdnuß zum Verringern des Hungers verwendet wird.

Um solche Mehrdeutigkeiten zu verhindern, wird eine Gruppe von zusammengehörigen Zustandsressourcen zu einem **Objekt** zusammengefaßt. Durch das objekt-orientierte Konzept können weiterhin Eigenschaften wie Vererbung etc. genutzt werden. Abbildung 5 zeigt die Anwendungen der EAT-Aktion auf eine Erdnuß.

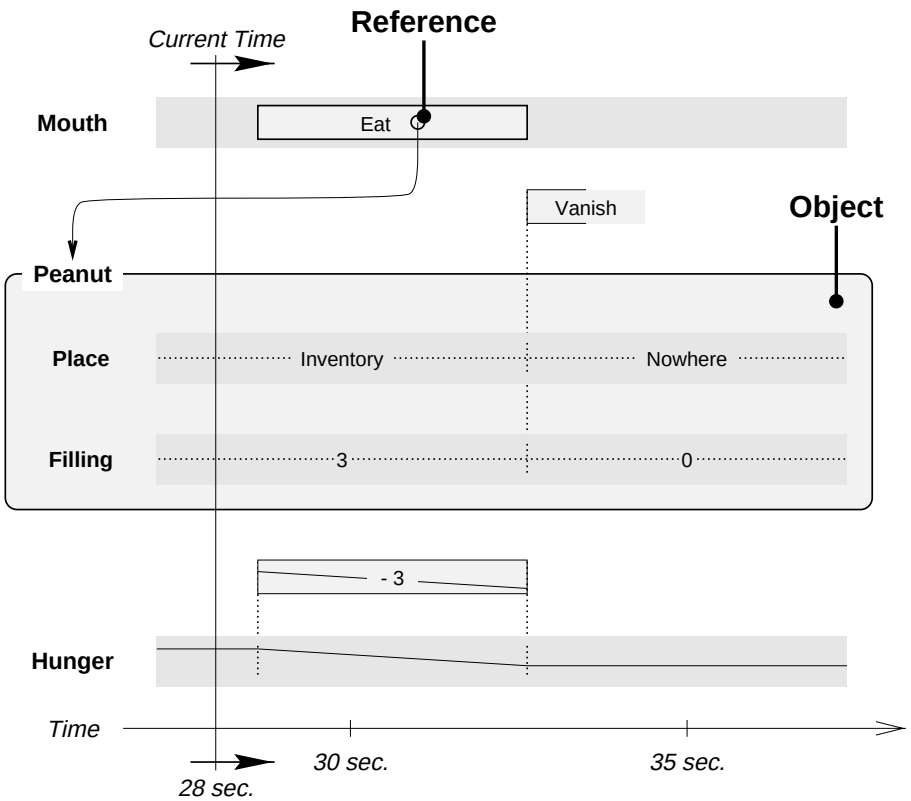

Abbildung 5: Referenzen und Objekte

Natürlich können nicht nur Aktionsaufgaben Referenzen nutzen. Zustandssaufgaben und Zustandsressourcen können diese ebenfalls verwenden, um beispielsweise Dinge wie BLOCK

A LIEGT AUF BLOCK B auszudrücken (siehe Abbildung 6).

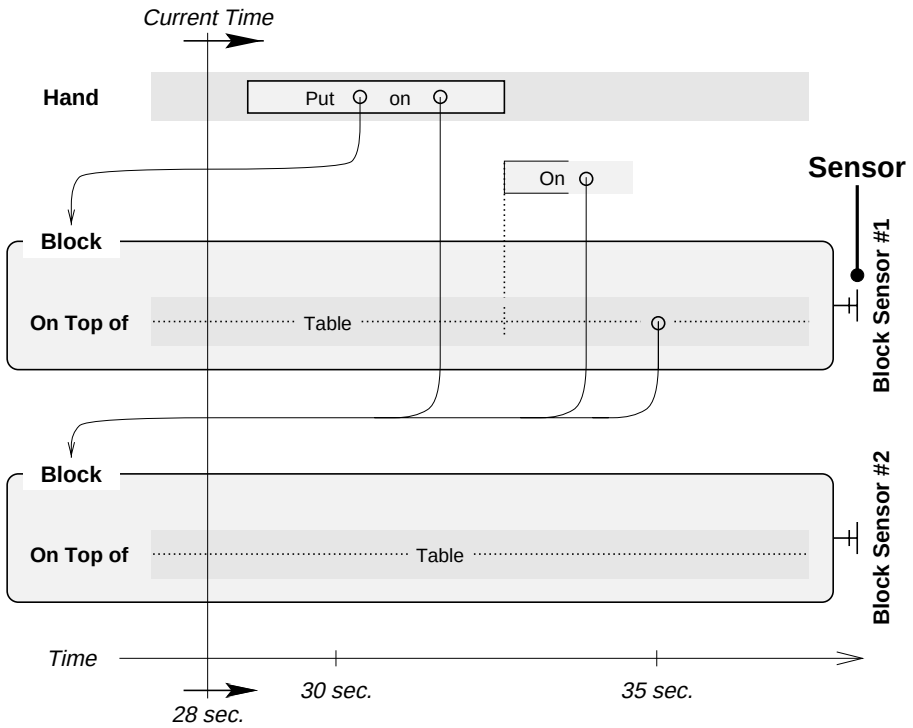

Abbildung 6: Ein Beispiel der Blocks World

Objekten kann ein **Sensor** zugewiesen werden. Sensoren spielen Daten der realen Welt ein, wie z.B. die derzeitige Wahrnehmung des Hungers. Es wird eine high-level-Wahrnehmung vorausgesetzt, welche bereits strukturierte Objekte liefert. Wahrgenommene Objekte werden zu den virtuellen Planobjekten in Bezug gesetzt.

Da sich der Agent in einer dynamischen Umgebung befindet, kann er sich nicht vollständig auf sein bisheriges Wissen verlassen. Deshalb besitzt jedes Objekt einen **Vertrauenswert**, der die Höhe des Vertrauens ausdrückt, daß ein virtuelles Planobjekt ein reales Gegenstück hat. Die Zuweisung eines Sensors zu einem Objekt bewirkt in der Regel einen sehr hohen Vertrauenswert. Ein Sensor muß jederzeit einem Objekt zugeordnet sein.

5 In Bezug stehende Arbeiten

Die meisten Planungssysteme verwendet eine STRIPS-artige Repräsentation, welche auf dem Sitation Calculus (McCarthy & Hayes 1969) basiert. Der Situation Calculus verwendet eine verzweigende Zeitpunkt-Struktur, wobei vollständige Weltbeschreibugen (Mengen von Zuständen) durch Aktionen verbunden werden. Im Gegensatz dazu gehört das EXCALIBUR-Modell der Gruppe der explicit-timeline-Ansätze an. Ihr prominentester Vertreter ist Allens temporale Intervallogik (Allen & Ferguson 1994). Die verzweigende Struktur des Sitation Calculus erlaubt einen sehr direkten Umgang mit unterschiedlichen möglichen Welten, während der Fokus der explicit-timeline-Ansätze in der Handhabung komplexer temporaler Zusammenhänge liegt.

Es gibt nur wenig Planungssysteme mit einer expliziten Darstellung von Zeit, z.B. *parc-*

PLAN (Lever & Richards 1994), ZENO (Penberthy & Weld 1994) und *Descartes* (Joslin 1996). Alle diese Systeme haben Probleme beim Ausdruck von Zustandsübergängen. In den meisten Fällen werden nur temporale Überlappungen sich widersprechender Aussagen verboten. *parc*PLAN erlaubt zusätzlich das Verbot einfacher Eigenschaftskonstellationen.

Das Problem der Repräsentation ist für normale Constraint-Solving-Tools für Planung und Scheduling, wie beispielsweise OZONE (Smith, Lassila & Becker 1996) oder CHIP (Simonis 1995), noch größer. Sie besitzen kein generelles Zustandkonzept und stellen die Verarbeitung von Kapazitätsressourcen in den Vordergrund. Die HSTS-Repräsentation (Muscettola 1994) kann Zustände und Zustandsübergänge sehr viel besser ausdrücken, ist allerdings auf einem sehr niedrigen Ausdruckslevel. EXCALIBURs spezialisierte high-level-Constraints machen es demgegenüber möglich, constraint-spezifisches Wissen über Suchkontrolle und Datenstrukturen zu integrieren.

Fast alle Planungssysteme suchen nach einem Plan durch Techniken der Verfeinerung. Die Verfeinerung ist ein schrittweiser Nährungsprozeß. In jedem Schritt wird eine Untermenge von Zuständen (bzw. Plänen) für die weitere Betrachtung ausgewählt, bis schließlich eine Lösung übrig bleibt. Im Falle einer Inkonsistenz wird ein Backtracking über die Verfeinerungsauswahl angestoßen.

Die Verwendung von lokaler Suche ist hingegen sehr selten und ist neben EXCALIBUR z.B. in Satplan (Kautz & Selman 1996) und ASP (Bonet, Loerincs & Geffner 1997) zu finden. Während eine langfristige Suche über Verfeinerungstechniken qualitativ bessere Resultate liefern kann, ergibt die lokale Suche kurzfristig sehr viel bessere Resultate. Für größere Probleme eignet sich die lokale Suche ebenfalls besser (Wallace & Freuder 1996; Gent et al. 1997). Des weiteren erweist sich ein dynamisches Umfeld als sehr viel geeigneter für die lokale Suche, da die lokale Suche von Modifikationen des Suchraumes kaum beeinträchtigt wird. Eine Suche über Verfeinerungstechniken muß hingegen das Wissen über den bereits abgesuchten Suchraum auf den neuesten Stand bringen und eventuell die gesamte Suche neu starten.

Der Ansatz der lokalen Suche beinhaltet bereits eine partielle Constraint-Satisfaction und Anytime-Verfügbarkeit. Dies ermöglicht schnelle Reaktionen, die für lange Zeit die Domäne populärer Alife-Agentensysteme waren, wie z.B. Subsumption Architectures (Brooks 1986). Derartige Systeme verwenden einen reaktiven Ansatz mit vordefinierten Verhaltensmustern und können nicht längerfristig planen. Viele hybride Agentensysteme wie die 3T Robot Architecture (Bonasso et al. 1997) oder der New Millenium Remote Agent (Pell et al. 1996) können reaktive Aktionen nur sehr eingeschränkt durchführen, da sie traditionelle offline-Planungssysteme für die längerfristige Planung einsetzen.

6 Zusammenfassung

Das EXCALIBUR-Agentenmodell ist auf die Anforderungen einer dynamischen Echtzeitumgebung zugeschnitten. Der explicit-timeline-Ansatz macht es möglich, komplexe temporale Relationen darzustellen, das Konzept globaler Constraints erlaubt eine deklarative Problemspezifikation mit Anwendung spezialisierter Lösungsverfahren und Datenstrukturen, und eine kontinuierliche Verbesserung/Umplanung durch lokale Suchmethoden ermöglicht die effiziente Konstruktion und Wartung des Plans des Agenten in Echtzeit.

Unsere zukünftige Arbeit beinhaltet u.a. die Verfeinerung der Verbesserungsheuristiken, die Koordination zwischen Agenten und die Integration von Lernmethoden. Weitere und ausführlichere Informationen über das EXCALIBUR-Projekt sind unter der folgenden Adresse

verfügbar:

`http://www.first.gmd.de/concorde/EXCALIBURhome.html`

Referenzen

Allen, J. F., und Ferguson, G. 1994. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation* 4(5): 531–579.

Bonasso, R. P.; Firby, R. J.; Gat, E.; Kortenkamp, D.; Miller, D. P.; und Slack, M. G. 1997. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9(1).

Bonet, B.; Loerincs, G.; und Geffner, H. 1997. A Robust and Fast Action Selection Mechanism for Planning. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 714–719.

Brooks, R. A. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation* RA-2 (1): 14–23.

Fikes, R. E., und Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2): 189–208.

Gent, I. P.; MacIntyre, E.; Prosser, P.; und Walsh, T. 1997. The Scaling of Search Cost. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 315–320.

Joslin, D. 1996. Passive and Active Decision Postponement in Plan Generation. Promotionsschrift, University of Pittsburgh, Pittsburgh, PA.

Kautz, H., und Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 1194–1201.

Lever, J., und Richards, B. 1994. *parc*Plan: a Planning Architecture with Parallel Actions, Resources and Constraints. In Proceedings of the Ninth International Symposium on Methodologies for Intelligent Systems (ISMIS-94), 213–223.

McCarthy, J., und Hayes, P. J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., und Mitchie, D. (Hrsg.), *Machine Intelligence 4*, Edinburgh University Press.

Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M., und Fox, M. S. (Hrsg.), *Intelligent Scheduling*, Morgan Kaufmann, 169–212.

Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; und Williams, B. C. 1996. A Remote Agent Prototype for Spacecraft Autonomy. In Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation.

Penberthy, J. S., und Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), 102–114.

Penberthy, J. S., und Weld, D. S. 1994. Temporal Planning with Continuous Change. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 1010–1015.

Puget, J.-F., und Leconte, M. 1995. Beyond the Glass Box: Constraints as Objects. In Proceedings of the 1995 International Logic Programming Symposium (ILPS'95), 513–527.

Simonis, H. 1995. The CHIP System and Its Applications. In Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP95).

Smith, S. F.; Lassila, O.; und Becker, M. 1996. Configurable, Mixed-Initiative Systems for Planning and Scheduling. In Tate, A. (Hrsg.), *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*, AAAI Press, Menlo Park (CA).

Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; und Blythe, J. 1995. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1).

Wallace, R. J., und Freuder, E. C. 1996. Anytime Algorithms for Constraint Satisfaction and SAT problems. *SIGART Bulletin* 7(2).

Parallel Programming in Common Lisp using Actors and Parallel Abstractions*

Lothar Hotz and Michael Trowe

Universität Hamburg
Labor für Künstliche Intelligenz
Fachbereich Informatik
Vogt-Kölln-Str.30, D-22527 Hamburg, Germany
hotz@informatik.uni-hamburg.de

Abstract. In this paper we describe an extension of Common Lisp which allows the definition of parallel programs within that functional and object-oriented language. In particular, the extensions are the introducing of active objects, sending synchronous and asynchronous messages between them, automatic and manual distribution of active objects to object spaces, and transparent object managing. With these extensions object-oriented parallel programming on a workstation cluster using different Common Lisp images is possible. These concepts are implemented as an extension of Allegro Common Lisp subsumed by the name NetCLOS. Furthermore, it is shown how NetCLOS can be used to realize parallel abstractions for implementing parallel AI methods at a highly abstract level.

1 Introduction

One of the big problems of Artificial Intelligence (AI) is getting its applications answer in time. Parallel computation is one way to solve this problem. But though there are many parallel implementations of basic AI techniques, there are very few AI applications which use them. This drawback is ascertained due to two reasons:

- Most of these implementations depend on special parallel hardware (e.g., [7, 4, 12]). This hardware is expensive and not widely available. Furthermore, the specification of many applications excludes the use of special hardware (e.g., personal assistant).
- Most of them are written in special parallel programming languages unknown to the application programmer and lacking features important to develop a complete application [22]. Hence their integration into such an application is difficult.

Especially for AI methods the proposed parallel languages and models are too different from commonly used languages. They do not provide the flexibility programmers need, and are not integrated with existing languages [13]. Furthermore, in special parallel languages non-parallel aspects are not adequately expressable.

Our goal is to simplify the parallel implementation of standard AI techniques. We think this means using standard hardware (i.e., a workstation cluster) and extending a language widely used for AI programming in a way that hides any kind of explicit parallel programming from the application programmer. Thus, we extend Common Lisp [19] with two levels of features for parallel programming on workstation clusters. The upper level is intended for easy use by the AI programmer inexperienced with parallel programming, while the lower level is intended for implementing the upper level. In the lower level an integration of an actor-like language [5] in Common Lisp and its object-oriented part CLOS (Common Lisp Object System) is introduced (see Section 3). The upper level realizes parallel abstractions as complex structures and operations on them (Section 4). The use of parallel abstractions is demonstrated with a constraint filtering algorithm (Section 5). In Section 6 related work is discussed. First, we give a more detailed view of our parallel programming model.

* This research has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (BMBF) under the grant 01 IN 509 D 0, INDIA - Intelligente Dffagnose in der Anw31

Fig. 1. Levels of used abstractions.

3 Extending CLOS with actors

In this section we describe the lower level (NetCLOS level), which is an extension of Common Lisp and its object-oriented part CLOS (Common Lisp Object System). Features of NetCLOS are:

- Active objects, which include data, methods, a mail queue, and a process for handling incoming messages by calling methods.
- Message passing for synchronous and asynchronous communication between active objects.

[1] NetCLOS as an extension to Allegro Common Lisp is implemented and can be received from the authors. NetCLOS is implemented using the metaobject protocol of CLOS (see [11] and [8]).

– Synchronization operations for delaying requests.

Programming with NetCLOS is done by creating active objects and sending messages between them, which is fully integrated in the programming style of CLOS. Distribution of active objects to workstations is widely hidden to the user. Thus, active objects are distributed over a virtual machine consisting of several Lisp images residing on several workstations of a cluster. To allow flexible programming of distributed active objects automatic distributions as well as explicit moving of active objects is included in NetCLOS.

There are some approaches which include parallel programming in Lisp (see e.g., [22]) but most of them concentrate on the functional part of Lisp. In NetCLOS the object-oriented part (CLOS) is focussed as an extendable part for parallel programming. With our new approach we introduce active objects in the spirit of actors [5, 1] in CLOS to make parallel object-oriented programming in Common Lisp possible.

3.1 Design decisions

Following [17] we discuss three dimensions of design issues for concurrent object-oriented programming: object model, internal concurrency, and interaction. The decisions are inspired by the concurrent object-oriented language ABCL/1 [21].

Object model. Because we extend an existing object-oriented language, where passive objects reside in the language, we use a heterogeneous object model with passive and active objects. Passive objects are normal CLOS objects, active objects are extended by a mail queue and a process. By buffering incoming messages in a mail queue active objects synchronize concurrent calls. Passive objects do not synchronize concurrent calls, i.e., they have to be saved by explicit synchronization calls or are used within a single-threaded active object.

Internal concurrency. Another design decision is whether an active object can process calls sequentially or in parallel. If calls are processed in parallel on the same active object, i.e. on one data source, a high communication rate will be necessary. Because of high communication costs in a workstation cluster, data and processes should reside on the same machine. Yet we decide to process tasks of one active object sequentially.

Interaction. In NetCLOS object identifications are used to determine the recipient of a message. Message passing can be done in three ways: *future*-messages, which are easy to integrate in a functional context, one-way messages (*past*-messages), as a more flexible but also more complicated tool for communication, and remote procedure calls (*now-messages*) for synchronous communication.

3.2 CLOS - the Common Lisp Object System

CLOS belongs to the ANSI Common Lisp standard [19] and defines the object-oriented part of the language. CLOS includes classes with multiple inheritance, generic functions, declarative method combination, and a metaobject protocol. Classes are defined by slots (instance variables or data fields) and some superclasses. All slots of all superclasses are inherited. Instead of having a message-passing concept as in other object-oriented languages, CLOS includes the more powerful concept of generic functions. A generic function describes a set of methods, i.e., a method is related to a generic function, not to one class. Because a generic function may have more than one discriminating argument, a generic function is related to a set of classes not to one specific class. Instead of passing a message to an object, the generic function is called. The classes of its arguments are used to determine (at runtime) which methods should be used to compute a value for the generic function. Declarative method combinations describe how several applicable methods should be ordered and how their results should be combined. This is done by defining different kinds of methods, e.g., *before-methods* are called before *primary-methods*, etc. The metaobject protocol is used to extend CLOS' behavior portably. For instance, the slot access can be modified to be a remote slot access. So called *metaclasses* can be defined by the user, which enhance the behavior of classes and objects (instances).

Fig. 2. Parts of an active object.

Sending messages and synchronization Active objects communicate only by using one of three message types (similar to ABCL/1): *past*, *now*, or *future*-messages. *Past*-messages are asynchronous one-way-messages. The caller can continue its work, after a message is sent. After calling the methods related to the message, the recipient does not send a reply message to the caller. Past-messages are declared by the keyword `:past` as in:

```
(defpargeneric <name> :past (<recipient-object> <argument1> ...))
```

Past-messages are used to realize complex request-reply frames.

Now-messages are remote procedure calls, i.e., the caller waits until the recipient accepts the message, computes the request, and sends the reply back. These messages are indicated by the keyword `:now`. Now-messages are used to ensure that the caller is inactive while processing the message. This can be used to realize a sequential interface to an active object or to ensure specific synchronization conditions.

When a *future*-message is sent, a *future* is created. Futures can be seen as simple active objects which can only deal with two messages: the past-message `write-result` and the now-message `touch`.

Fig. 3. Transfer of a query via a proxy.

Distribution The distribution model of NetCLOS is based on the notion of one *object space* residing on each workstation of a cluster. An object space contains all information to handle active objects, e.g., all necessary classes and functions are known to each object space.[2] Every active object resides on exactly one object space. But an active object can be referenced from each object space, not only the local one. Thus, the identity of an active object is guaranteed over all object spaces, i.e. each active object is unique and can be referenced from diverse object spaces. When messages are sent or an active object is passed as argument of a message, it makes no difference if the object is locally or remotely referenced. This holds only for active objects, other data types — like passive objects, lists, arrays, strings, or records — are only locally referenced. If objects of such data types function as arguments of a message, a copy is sent to the recipient, i.e., changes to those types made by the recipient are not known to the caller. Thus, no side effects on such datatypes are allowed. The copy

[2] Special features are defined for distributing new definitions of generic functions and classes and for defining systems (sets of files), which have to be known to all object spaces.

of such an object includes also nested objects (e.g. lists of lists). Cyclic data is handle correctly, by creating the same cycles in the remote object. Copying is done by generating a Lisp form, which when evaluated creates the appropriate objects, and sending the Lisp form to the recipient.

There are two alternatives for an application to distribute its active objects to object spaces: One is to move the object explicitly by calling the function `move`[3], the other is to use a predefined distribution class. There are two classes for distribution: one to realize a static distribution by deriving Task Interaction Graphs from the reference structure of the active objects to distribute and another class distributes the active objects dynamically when they are created. For the latter only a simple scheme is present for sending active objects to object spaces in a round robin manner. For distributing Task Interaction Graphs we use a combination of bisection and Kernighan-Lin (see section 4.1). But extensions of NetCLOS made by subclassing can be defined to realize more sophisticated distribution strategies.

For moving an active object explicitly the moving behavior of its slots can be specified when defining the active object. When specified with `:follow` the value of that slot (another active object) is moved in the same object space as the active object itself. When specified with `:stay` the value of the slot stays in the current object space. Instead of the value the moved active object contains a representant value (i.e., a *proxy*) as slot value.

Remote references Remote references to values is realized by proxies. A proxy knows the location of the original active object and sends a kernel message to the original active object on a proxy reference to get the referenced value. The necessary infrastructure is internal in NetCLOS. When moving an active object appropriate proxies are automatically created. If a slot is of type `:follow` a local proxy is created which referes to the remote slot value. If a slot is of type `:stay` a remote proxy is created which referes to the local slot value. Garbage collection is extended to handle proxy references, as the next paragraph describes.

Object spaces An *object space* is realized as a Lisp image and resides on one host; it is assumed that each host processes only one object space. An object space contains some features (some realized as light weight processes (lwp) inside one Lisp image), which realize the functionality of a virtual machine.

Object spaces communicate with each other by kernel messages. An object space contains one caller-lwp for each object space it wants to communicate with. The caller-lwp packs the message to be sent (i.e., creates a Lisp form which contains the message on evaluation) and sends it via TCP/IP to the other object space. There, the callee object space contains an lwp for realizing a recipient-lwp for each other object space. The recipient-lwp unpacks the message and evaluates the resulting Lisp form.

Each object space contains an object store, which contains local and remotely referenced objects. It ensures exactly one proxy for each remote object, and it realizes a garbage collection method for remote objects. This is necessary, because the internal garbage collection method of Lisp is image specific and references of proxies (residing in a remote image) to objects are not considered. Thus, with the internal garbage collection method an object would be garbage collected even if a proxy residing on another space refers to it. The remote garbage collection is carried out by counting remote references to each object. When no reference to a proxy and no local reference to the related object is present this object can be garbage collected or if the counter decrements to 0, the object can be garbage collected by the internal garbage collector contained in each Lisp image. A problem not yet attended to are garbage collecting cyclic reference structures.

Each object space contains furthermore an *object space manager* (or *object server*). Working with NetCLOS starts by loading NetCLOS in a Lisp image, which creates an initial object space on the host

[3] In the current implementation the function `move` can not be used in generic functions being performed in parallel on one active object, because the process synchronizing the mail queue is not moveable in Lisp.

the Lisp image is started — the master host — and an initial manager — the master[4]. This manager starts the virtual machine by giving it a number of hostnames[5]. On each host an object space is started, is initialized by some initialization forms and the communication links are established, which connect the object spaces to each other. Thus, a fully connected communication structure is created. Furthermore, the manager ensures an equal global context of classes and functions. When classes and functions are loaded in one object space, the manager sends an appropriate message to all object spaces which ensures a loading of the same classes and functions in those other spaces. If one object space stops working (e.g. because the Lisp image quits) it sends a specific message to each object space, which can react appropriately and can proceed working.

Integration of NetCLOS in CLOS There are two viewpoints to consider when integrating NetCLOS into CLOS: the implementors view and the application programmers view. From the implementors view NetCLOS is integrated in CLOS portably, i.e., without changing the implementation of CLOS. Even more, by extending the existing features, a small extension of the behavior of CLOS yields to big expressability. E.g., the slot access is extended by the possibility of defining moving behavior for slots. A slot access protocol inherent in CLOS is extended to handle this moving behavior and thus, every slot access for active objects is changed. From a programmer's point of view this is done by the same programming interface, i.e., the slot access function does not change to, e.g., special proxy access functions like `proxy-value`. Besides extending the slot access generic function metaclasses are integrated in NetCLOS for describing generic functions to be handled as messages, i.e. for each method call special methods for testing the active object's location (local or remote) and selecting the appropriate send style (now, past, future) are automatically integrated by these metaclasses. Furthermore, for each class *c*, whose instances can be moved, a subclass *proxy-c* is created. This class is of type *proxy-class*, a metaclass, which implements proxy behavior. For example, this metaclass creates only instances, which does not contain any slots, but sends slot references as messages to a remote instance, which contains the slots. Thus, with *proxy-c* the instance allocation protocol and the slot access protocol are extended.

This approach of extending CLOS is possible because of the existence of a metaobject protocol [11], which clearly specifies the behavior of diverse CLOS features, like slot access, method combination, and inheritance behavior. The extensions are portable in the sense that each CLOS implementation based on the metaobject protocol can be extended by NetCLOS. The usage of the metaobject protocol is different to a library approach where a number of functions have to be introduced and learned before a parallel program can be written. For further reading on this point see also [8].

Thus, from a programmer's point of view the extensions fit well in the programming style of CLOS. Even the programming of message passing instead of generic function calls are acceptable, because it comes as a special generic function call (i.e., to the first argument). Some Lisp specific features have to be handled with care, because they are not yet implemented in NetCLOS or are hard to integrate in a distributed environment. For instance, closures cannot be moved from one host to another and dynamically created functions are not yet handled correctly. This is due to the fact, that closures are not part of a metaobject protocol and thus, are not accessable without touching the implementation of Lisp. However, it is possible to define generic functions (i.e. named closures) and classes in NetCLOS, which are distributed to all object spaces, thus every space knows the same functions and classes. Neither are cyclic reference structures of active objects garbage collected. However, NetCLOS is used to implement parallel object-oriented programs based on CLOS.

4 Introducing parallel abstractions for programming AI applications

Parallel programming is a difficult task, because of the possibly big number of flows of control. In low-level parallel languages the handling of these flows is left to the programmer. To make parallel

[4] In the current implementation only the master can start object spaces, object spaces cannot connect to the master from outside. Thus, client-server structures on the object space level are not yet realizable.

[5] The current implementation does not include a user-password handling, thus, only trusted remote hosts of a workstation cluster can be given.

Fig. 4. Control abstractions and their integration in application classes.

4.1 An example structure — the relaxation net

Relaxation net is an abstraction implementing parallel discrete relaxation (see [7] for a similar approach). It consists mainly of

- a class of active objects (*value nodes*) acting as shared stores. Accesses to these stores are automatically synchronized, i.e. this is done by the NetCLOS level. These active objects can be used to implement the variables of a constraint net.

[6] i.e. the time when the application program is written down.

- a class of active objects (*function nodes*) which, when activated, computes a function of the content of a set of stores. These active objects can be used to implement constraints.
- a structure class which organizes stores and functional objects into a network and provides for iterated activation and parallel execution of the functional objects (i.e. a relaxation operation). This *relaxation net* can be used to implement a constraint net.

To distributed the *relaxation net* function and value nodes are modeled as tasks of a Task Interaction Graph. To distribute this graph on a workstation cluster we use a combination of bisection [18] and the Kernighan-Lin algorithm [10].

The main operation on relaxation nets is a function, which computes a fixed-point. This function can be processed in parallel if the domain of the function can be partitioned in parts and the function itself can be partitioned in independent component functions (see [20, 7] for details and Appendix A). To use the parallel abstraction *relaxation net*, the application programmer implements subclasses of the value and function node classes and the structure class *relaxation net*, redefining some methods, i.e. implements a normal object-oriented sequential interface. There is no need for any explicit parallel programming (see Appendix B).

4.2 Another example — implementing distributed AI applications with NetCLOS

In distributed AI besides others the concept of communicating agents is present. Agent structures are not yet implemented with NetCLOS, but can be realized as follows. To implement an agent an active object can be used. On which host an agent proceeds can be fixed by the user or can be decided by the system (realized by a simple distribution scheme of round robin, see section 3.4), e.g. each agent can reside on a distinct object space. Furthermore, it is possible to add new agents dynamically. For diverse agents communication schemes, e.g. direct communication of agents or blackboard architectures, necessary message protocols can be implemented by NetCLOS messages. Concrete steps may be as follows: a virtual machine consisting of n object spaces is started from the master host. Agents possibly of distinct types are created by the master and distributed to the object spaces. A past-message e.g. *do-work* starts the action of each agent, which may perform different problem solving tasks. The agents work in parallel and may communicate by further messages to each other.

5 Experimental results

We tested NetCLOS by implementing a parallel abstraction named *relaxation-net* (see also [7]), which contains a net-like reference structure of active objects and a fixpoint operation on that structure. The net is distributed on a workstation cluster by the abstraction and the operation is executed in parallel on diverse parts of the net, i.e., distribution and parallel processing is done by the abstraction. This abstraction is used for implementing a local propagation algorithm for constraint nets, i.e., on this level only the sequential interface of the parallel abstraction must be known to an application programmer.

To get a gain of parallel execution of a NetCLOS program, one has to take high communication costs into account which are related to the infrastructure of a workstation net, e.g., an ethernet or TCP/IP. Thus, as usual in such a case, the computation time on one host should be high enough to compensate the communication costs. This is also the result of experiments we made. When solving a line-diagram labeling problem [16], we only got a speed up for constraint propagation when raising the number of constraints (see Figure 5). The speed up strongly depends on the communication traffic on the ethernet and on the kind of workstations used, which are typically heterogeneous (e.g. from Sparc Classics to Ultra Sparcs). The distribution strategy does not yet consider such kind of information. Because of using a derived not a dynamic structure the constraint net is first created on one object space and than distributed to the other. This still yields to high distribution costs, which are not included in the presented results. Furthermore, all experiments are executable on only one machine. However, the experiments show that one can get a speed up for constraint propagation, when using NetCLOS.

Fig. 5. Speed-up when increasing problem size (given here in number of stairs n of diagrams like the right one) and number of workstations. For 1000 stairs we got a speed up of 3.6 for constraint propagation on 7 machines. The number of constraints is $6n + 7$ and of variables is $4n + 7$.

Because NetCLOS is integrated in Common Lisp its programming environment (profiler, debugger, editor etc.) can be used also for each object space separately. To illustrate parallel issues of programming (e.g. communication costs) specific environment extension would be useful but are still not realized (see e.g. [15]).

6 Related work

The work on NetCLOS is derived from concurrent object-oriented programming languages related to actors [5, 1, 21]. Thus, the notion of active objects, proxies, asynchronous and synchronous message passing etc. are similar. However, our main interest is to integrate such concepts in Common Lisp and CLOS as a language used for AI applications. In NetCLOS the integration of concurrent object programming is done in the CLOS programming style by introducing new subclasses, metaclasses, declarative method combination, slot options, and protocols. Thus, a CLOS programmer can use NetCLOS without learning a new parallel language.

The extension of CLOS by active objects enables parallel object-oriented programming, and thus, parallel abstractions. Other approaches [22] introduce mainly function-oriented parallel programming in Lisp by allowing parallel execution of functional arguments. A precondition for these approaches is a side effect free programming style, which is not realizable in realistic Lisp applications. Furthermore, functional approaches often generate a big number of small tasks, which increase the overhead.

Another Lisp related implementation for parallel programming is Kali Scheme [2]. Besides very similar features like adress spaces, proxies, diverse communication primitives the main difference is that in Kali Scheme first class continuations and first class procedures are supported for programming in continuation-passing style. The integration of these concepts in Lisp without non-portable access to the Lisp implementation is not possible, because the lack of first class continuations and a metaobject protocol for the functional part of Lisp. However, our interest is more a practical one: First, we use Common Lisp instead of Scheme because of its use in application programming for realizing e.g. simulation, configuration, diagnosing, and information managment systems. Second, we use Common Lisp and add the extension modul NetCLOS to it instead of defining a new language to make it possible that existing Lisp programs can still be used.

Other approaches like CMLisp [6] introduces data parallel abstractions. This showed that programming with abstractions can simplify parallel programming, but CMLisp is restricted to run on single instruction multiple data machines (i.e. the Connection Machine 2) and thus, is hardly usable for workstation clusters. This is similar to [7], where a relaxation operation is introduced to solve constraint problems, but the implementation is done on a Sequent Symmetrie, not on a more common workstation cluster.

How NetCLOS can be used for Internet programming and how CL-HTTP can support this, is part of our current work.

Netzwerkfähige Lisp-Anwendungen mit graphischer Oberfläche durch Einsatz von Standardkomponenten

Rolf Diekmann *Wolfgang Goerigk* *Ulrich Hoffmann*

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel, Preußerstraße 1-9, D-24105 Kiel
{rod | wg | fs}@informatik.uni-kiel.de

Zusammenfassung

Wir beschreiben ein Schnittstellen-Modul zwischen Common Lisp und Tcl/TK, das sowohl die interaktive Entwicklung ereignisorientierter Bedienoberflächen für Lisp-Anwendungen als auch die Portierung und Generierung eigenständig ablauffähiger Anwendungen mit dem Common-Lisp-Übersetzer CLiCC unterstützt. Basierend auf dem Software-Entwurfsmuster Model-View-Control lassen sich damit Lisp-Anwendungen als Client-Server-Anwendungen auch über Rechnernetze nutzen. Lisp als KI-Programmiersprache kann so auch modernen softwaretechnischen Anforderungen gerecht werden. Aufwendige Reimplementierung von KI-Anwendungen ist nicht zwingend erforderlich.

1 Einführung

Dieser Beitrag richtet sein Augenmerk auf die Handhabung von Lisp-Programmen oder Programmteilen in modernen softwaretechnischen Umgebungen mit gewachsenen Anforderungen an Flexibilität, Portabilität, Netzwerk- oder Client/Server- und Integrationsfähigkeit. Lisp ist eine klassische KI-Programmiersprache, in deren Umfeld viele auch deklarative KI-Methoden entstanden sind [23]. Da die KI unumstritten viele nutzbringende Methoden, ja Paradigmen, hervorgebracht hat und hervorbringt, sich die KI insgesamt aber ebenso unumstritten auf dem Weg „in die Normalität“ befindet [4], drängen sich softwaretechnische Fragestellungen bei der Nutzung von KI-Techniken stärker in den Vordergrund.

Sehr hohe, dynamische Programmiersprachen leisten einen wesentlichen Beitrag zur Beherrschung komplexer Anforderungen an Software-Systeme in der praktischen Anwendung. Eigene Implementierungsarbeit weicht zunehmend der Nutzung vorhandener, vorgefertigter Software-Bausteine, sog. Komponenten. Integration ist zentraler Bestandteil der Software-Entwicklungsarbeit. Dies wird um so wichtiger, je mehr Arbeit, auch Verifikationsarbeit, in die Komponentenentwicklung investiert wird. Teure Reimplementierung wird sich kaum jemand erlauben können.

Derzeitige Softwareentwicklung nutzt bereits heute de facto eine Sprachvielfalt: Sogenannte Skript-Sprachen finden Verwendung, aber auch C als Basis-Systemprogrammiersprache unterhalb der höheren Anwendungs-Programmiersprache.

Im BMFT-Projekt APPLY[1] [5, 6, 11] und in einer darauffolgenden Studie [12] in Zusammenarbeit mit dem Deutschen Forschungsinstitut für Künstliche Intelligenz (DFKI) haben wir einen Weg aufgezeigt, wie sich die Sprache Lisp, die sich gerade bei der Implementierung hochkomplizierter, mathematisch fundierter Software-Komponenten, aber auch bei der Entwicklung experimenteller, eher forschungsorientierter neuer Problemlösungsverfahren, erhebliche Meriten verdient hat, in dieses Szenario einbetten kann. Bislang lassen sich in Lisp implementierte Software-Komponenten, wenn über-

[1] Das Verbundvorhaben APPLY (GMD, ISST, VW-GEDAS, CAU Kiel) wurde von 1991 bis 1994 durch das BMFT gefördert.

haupt, nur sehr schwer in vorhandene heterogene Software-Umgebungen integrieren, unter anderem weil die Lisp-Hersteller bis heute nur an der Integration Lisp-fremder Software nach Lisp hinein, nicht aber umgekehrt an der Integration von Lisp-Komponenten in Lisp-fremde Software-Umgebungen gearbeitet haben.

In diesem Papier beschreiben wir eine Integration von Common Lisp [21, 22] mit Tcl/TK [19] (ToolCommandLanguage/ToolKit). Diese Arbeit ist im Rahmen der Weiterentwicklung des Common-Lisp-Übersetzers CLiCC [7, 8, 16] entstanden, mit zweierlei Zielsetzung: Zum einen wollten wir die einfache, interaktive Entwicklung graphischer, ereignisorientierter Bedienoberflächen für Lisp-Software unterstützen und dabei auf ein Standardwerkzeug zurückgreifen, das leicht erlernbar und auf verschiedenen Plattformen verfügbar ist. Zum anderen war einfache Portierbarkeit und Anwendungsgenerierung Ziel. Beides ist gelungen. Die Schnittstelle zwischen Lisp und Tcl/TK ist sowohl für ein interaktives Common-Lisp-System als auch für CLiCC implementiert [10], so daß Anwendungen auf Zielrechner portiert werden können, die über Tcl/TK und einen Standard-C-Übersetzer verfügen. Ein Lisp-System muß auf der Zielarchitektur nicht verfügbar sein.

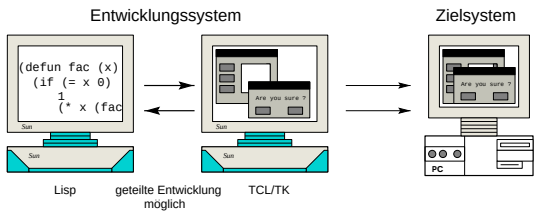

Abbildung 1: Entwicklung und Portierung von Lisp-Client/Server-Anwendungen mit graphischer Tcl/TK-Bedienoberfläche

Schließlich lassen sich Lisp-Anwendungen, z.B. in Lisp implementierte Server, über Standard-Netzwerkdienste wie TCP/IP in Rechnernetzen verfügbar machen. Dies beschreiben wir im letzten Abschnitt. Sie können dann beispielsweise durch Tcl/TK-Clients bedient werden, auch über das World Wide Web. Reimplementierung ist nicht nötig.

2 Lisp und Tcl/TK

Bei dem Entwurf der Schnittstelle zwischen Lisp und Tcl/TK haben wir uns von einem klassischen Software-Entwurfsmuster (*design pattern*) leiten lassen, dem *Model-View-Control*. Dabei gehen wir davon aus, daß die Kernanwendung, das Modell, in Lisp implementiert ist, während graphische Oberfläche (Sicht) und ereignisorientierte Kontrolle im wesentlichen in Tcl/TK geschrieben sind.

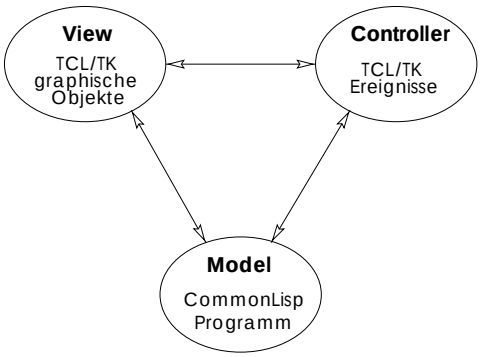

Abbildung 2: Software-Entwurfsmuster Model-View-Control

Damit sind Kernanwendung und Bedienoberfläche lose gekoppelt. Kommunikation geschieht, wie in Tcl/TK üblich, über Zeichenketten, was insbesondere einer Verteilung der Anwendung im Sinne einer Client-Server-Architektur entgegenkommt, da Kommunikation ohne weiteres auch über Standard-TCP/IP-Verbindungen geschehen kann (vgl. Abschnitt 5).

2.1 Die Tcl/TK-Schnittstelle

Eine typische Verwendung der Lisp-Tcl/TK-Schnittstelle, die wir im folgenden etwas genauer beschreiben wollen, läßt sich wie folgt kurz umreißen:

1. Der Lisp-Teil der Anwendung baut alle nötigen Lisp-Datenstrukturen auf und initialisiert sie.

2. Das Lisp-Programm startet eine Tcl-Interpreter-Instanz und stößt ein Tcl-Skript an, das die graphische Oberfläche (View und

Controller) der Anwendung aufbaut. Insbesondere werden Tcl-Kommandos etabliert, die Lisp-Funktionen ausführen werden.

3. Das Lisp-Programm übergibt die Kontrolle an die Event-Schleife des Tcl-Interpreters. View und Control werden von Tcl gesteuert.

4. Durch spezielle Benutzerinteraktionen verläßt die Kontrolle die Event-Schleife und Lisp baut den Tcl-Interpreter ab und beendet die Anwendung.

Dabei sind das Vorgehen, ja sogar der Programmcode des Lisp- und des Tcl/TK-Teils identisch, unabhängig davon, ob die Anwendung in einem klassischen interaktiven Common-Lisp-System (wie AllegroCL) oder als Kompilat des Common-Lisp-Übersetzers CLiCC abläuft.

2.2 Funktionsumfang

Aus Sicht der Anwendung stellt sich die Tcl/TK-Schnittstelle als ein Lisp-Modul dar, das Funktionalität beinhaltet, die die Kommunikation der Lisp-Anwendung mit einer Instanz eines Tcl-Interpreters ermöglicht. Es stehen Funktionen zur Verfügung, die es erlauben,

- eine Tcl-Interpreter-Instanz zu starten und zu beenden,
- Tcl-Kommandos zu installieren, die Lisp-Funktionen ausführen,
- Tcl/TK-Befehle auszuführen und
- globale Variablen in Lisp und Tcl aneinander zu binden.

Letzteres ermöglicht die Verwaltung eines globalen Zustandes, der dem Lisp-Teil der Anwendung über Lisp-Variablen und dem Tcl/TK-Teil über Tcl-Variablen zugreifbar ist, deren Werte synchronisiert werden.

Die Tcl/TK-Schnittstelle stellt im wesentlichen drei globale Variablen und eine Reihe von Funktionen zur Verfügung, mit denen Lisp-Programme Funktionalität einer dynamisch erzeugten Instanz eines Tcl-Interpreters nutzen können. Nach Initialisierung ist diese Instanz über einen Deskriptor, der in der

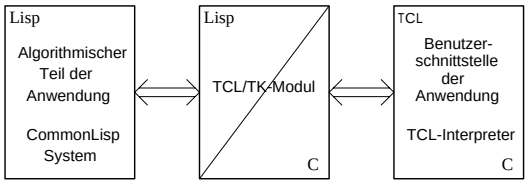

Abbildung 3: Die Tcl/TK-Schnittstelle

Variablen `*TclInterp*` abgelegt ist, ansprechbar. Die beiden anderen globalen Variablen, `*TraceVars*` und `*TclLispCallable*`, enthalten Hash-Tabellen, die Tcl-Variablen bzw. Tcl-Befehlsnamen mit zugehörigen Lisp-Aktionen assoziieren.

Die Variable `*TraceVars*` wird zum Verbinden (Synchronisieren) von Tcl- und Lisp-Variablen verwendet, `*TclLispCallable*` enthält die Lisp-Funktionen für installierte Tcl-Kommandos, die aus Tcl heraus aufgerufen werden können. So kann Tcl Lisp-Funktionen aufrufen (*call-in*). Andersherum geschieht der Aufruf von Tcl-Funktionen aus Lisp durch Aufrufe der Interpreterinstanz mit dem auszuführenden Tcl-Kommando als Argument.

Funktionsübersicht

Bei den weitaus meisten der zur Verfügung gestellten Funktionen handelt es sich um Wrapper-Funktionen, d.h. Aufrufhüllen für Funktionen der Tcl-Bibliothek. Die Funktionen lassen sich wie folgt gliedern:

- `tcltk-interp, tcl-init, tk-init, tcl-delete-interp, get-tcl-result-string, get-interp-result-string, null-c-string-p`

  sind Funktionen zur Verwaltung des Tcl-Interpreters. Sie erlauben, eine Tcl-Interpreter-Instanz erzeugen, zu löschen, zu initialisieren usw.

- `tcl-setvar, tcl-getvar, tcl-unsetvar`

  erlauben dem Lisp-Programm, Tcl-Variablen in verschiedenen Kontexten des Tcl--Interpreters anzulegen, zu setzen, auszulesen und auch wieder zu löschen.

- `tcl-bind-lisp, tcl-unbind-lisp`

erlauben die oben erwähnte Verbindung von Lisp- mit Tcl-Variablen und damit die Verwaltung eines zwischen Modell und View/Controller synchronisierten globalen Zustandes.

- `tcl-eval, tcl-global-eval, tcl-eval-file`
  erlauben die Ausführung von Kommandozeilen oder Skript-Dateien durch den Tcl-Interpreter.
- `tcl-create-command, tcl-delete-command, tcl-get-command-info`
  dienen der Definition von Lisp-Funktionen als Tcl-Kommandos.
- `tk-create-window-from-path, tk-name-to-window, tk-pathname, tk-destroy-window, tk-main-window, tk-show-main-window, tk-make-window-exist`
  dienen dem Aufruf von Fensteroperationen des TK-Toolkits.
- `tk-main-loop, tcl-do-one-event, tcl-do-queued-events`
  schließlich ermöglichen die Handhabung und gesteuerte Abarbeitung von Tcl-Ereignissen (Events). Realisiert wird dieses vom im Tcl-Interpreter realisierten Tcl-Event-Loop, der die Behandlung aller vom System und dem Benutzer eingehenden Ereignisse (Events) erledigt. Die Funktionen ermöglichen es nun, die Kontrolle in verschiedenen Granularitäten an die Ereignis-Verarbeitungsinstanzen von Tcl abzugeben. Während `tk-main-loop` die Kontrolle an Tcl abgibt, bis es keine Quelle für Ereignisse mehr gibt, geben `tcl-do-one-event` bzw. `tcl-do-queued-events` die Kontrolle nur für ein bzw. alle bisher aufgelaufenen Ereignisse an Tcl ab.

Drei Funktionen im Detail

Die wesentlichen drei Funktionen sollen im folgenden etwas genauer beschrieben werden. Es sind `tcl-eval`, `tcl-create-command` und `tcl-bind-lisp`. Es sind die Funktionen, die die drei wesentlichen Konzepte der Schnittstelle implementieren,

- die Ausführung von Tcl-Kommandozeilen aus Lisp heraus,
- den Aufruf von Lisp-Funktionen aus Tcl heraus und
- die Verbindung globaler Lisp- und Tcl-Variablen.

Die Funktion `tcl-eval` dient der Auswertung eines Tcl-Ausdrucks durch den Tcl-Interpreter und realisiert somit einen *call-out*, d.h. den Aufruf eines Tcl-Kommandos aus Lisp heraus. Als Parameter erwartet `tcl-eval` einen gültigen Tcl-Ausdruck in Form einer Zeichenkette, die vom Tcl-Interpreter ausgewertet werden soll. Die Funktion liefert, ebenfalls als Zeichenkette, das Resultat (oder `NIL` im Falle eines Fehlers). Zunächst besorgt sich `tcl-eval` aus der globalen Lisp-Variable `*TclInterp*` das abstrakte Handle der aktuellen Tcl-Interpreter-Instanz und testet, ob dieser bereits gültig, d.h. ob der Tcl-Interpreter existent und initialisiert ist. Das Argument ist eine Lisp-Zeichenkette, die durch eine Funktion des FFI des Lisp-Systems in eine entsprechende Zeichenkette in C-Repräsentation transformiert wird. Über den in C realisierten Teil der Lisp-Tcl/TK-Schnittstelle wird diese Zeichenkette dem laufenden Tcl-Interpreter zur Auswertung übergeben. Das Resultat schließlich wird in eine Lisp-Zeichenkette transformiert.

Die Funktion `tcl-bind-lisp` ermöglicht die Erstellung einer 'Verbindung' zwischen einer Tcl-Variablen und einer Lisp-Variablen. Diese Verbindung ist wie folgt zu charakterisieren: Der Wert der Tcl-Variablen spiegelt sich immer im Wert der assoziierten Lisp-Variablen wider, aber nicht automatisch umgekehrt. D.h. wird der Wert einer Tcl-Variablen verändert, so erhält die mit dieser Tcl-Variablen assoziierte Lisp-Variable, falls es eine solche gibt, den gleichen Wert zugewiesen, allerdings als Lisp-Datum. Dabei spielt es keine Rolle, ob die Tcl-Variable zu dem Zeitpunkt, an dem diese Verbindung mit `tcl-bind-lisp` aufgebaut wurde, schon existierte oder nicht. Die Lisp und die Tcl-Variable werden bei der Erstellung aus dem jeweiligen aktuellen Bindungskontext von Lisp bzw. Tcl ge-

nommen und sozusagen 'statisch' aneinander gebunden. Realisiert wird diese Verbindung durch eine globale Hashtabelle in Lisp und mit der Funktion `Tcl_TraceVar` der Tcl-Bibliothek. Diese ermöglicht es, dem Tcl-Interpreter eine Funktion bekanntzumachen, die immer dann aufgerufen wird, wenn sich der Wert der Tcl-Variable ändern soll. Die installierte Funktion wird dann, noch bevor der Wert zugewiesen wird, aufgerufen und übernimmt die Synchronisation des Werts der entsprechenden Lisp-Variablen.

Die Funktion `tcl-create-command` ermöglicht der Lisp-Anwendung, den Tcl-Interpreter um eine in Lisp realisierte Tcl-Funktion dynamisch zu erweitern. Dies realisiert einen *call-in* registrierter Lisp-Funktionen aus Tcl heraus. `tcl-create-command` hat zwei Argumente. Im ersten Argument steht der Name der neu einzurichtenden Tcl-Funktion (eine Lisp-Zeichenkette), das zweiten Argument enthält die Lisp-Funktion, die durch Tcl aufgerufen werden soll. Konnte kein neuer Tcl-Befehl angelegt werden, so liefert `tcl-create-command NIL`. Die Lisp-Funktion `tcl-lisp-dispatcher` kümmert sich dann darum, daß im Fall eines *call-in* die richtige Lisp-Funktion mit schon nach Lisp konvertierten Daten als Argumente aufgerufen und daß entsprechend der Rückgabewert der Lisp-Funktion in ein C-Datum konvertiert wird.

3 Entwicklung

Zur Entwicklung von Anwendungen steht eine Implementierung der Tcl/TK-Schnittstelle für Allegro-Common-Lisp zur Verfügung, die sich auf die Fremdsprachen-Schnittstelle (FFI) von AllegroCL stützt. Die Einbettung der Schnittstelle ist in Abbildung 4 dargestellt.

Um das übersetzte C-Programm in AllegroCL laden und benutzen zu können, muß die aus dem C-Programm generierte Objekt-Datei zu einer dynamischen 'shared-library', d.h. zu einer zur Laufzeit nachladbaren und für mehrere Programme nutzbaren Bibliothek gebunden werden. Dies wird auf Unix-Systemen durch den System-Linker ermöglicht. Auf anderen Systemen gibt es entsprechende Entwicklungswerkzeuge, die die Generierung von 'shared-library' (oder DLL's) erlauben.

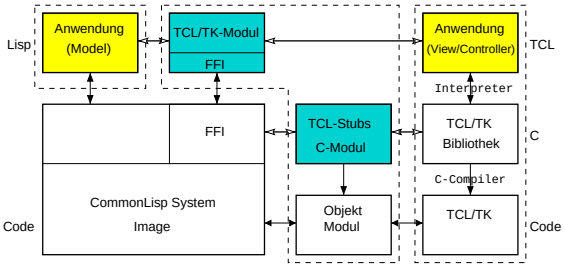

Abbildung 4: Die Tcl/TK-Schnittstelle mit einem Common-Lisp-Entwicklungssystem

Die Bibliothek wird vom Tcl/TK-Modul mit der Anweisung (`load "tcltk-allegro.so"`) in das interaktive Lisp-System geladen und anschließend werden die in dieser Bibliothek enthaltenen Funktionen mit Hilfe von Funktionen der Fremdsprachen-Schnittstelle von AllegroCL an spezielle Lisp-Funktionen gebunden. Die Namen werden durch eine Signaturbeschreibung festgelegt, in der auch die Typen für Parameter und Wert der Funktionen deklariert werden.

4 Anwendungsgenerierung und Portierung mit CLiCC

Die Schnittstelle ist ebenfalls als Modul für den Common-Lisp-Übersetzer CLiCC implementiert. Tatsächlich ist nur der Code geändert, der die CLiCC-Fremdsprachen-Schnittstelle bedient. Der Rest des Codes ist gleich. CLiCC kompiliert vollständig nach C und läßt dynamisches Nachladen von Code im allgemeinen nicht zu. Die Integration geschieht auf der Ebene von C-Quellcode oder Objektcode, der statisch oder zur Ladezeit als *shared library* dynamisch gebunden wird. Die Lisp-Tcl/TK-Schnittstelle ist Bestandteil des CLiCC-Laufzeitsystems.

Wie die Bilder in den Abbildungen 4 und 5 bereits suggerieren, muß zur Portierung mit CLiCC an dem Code der Anwendung nichts geändert werden; der Lisp- und auch der Tcl/TK-Teil können vollständig unverändert bleiben.

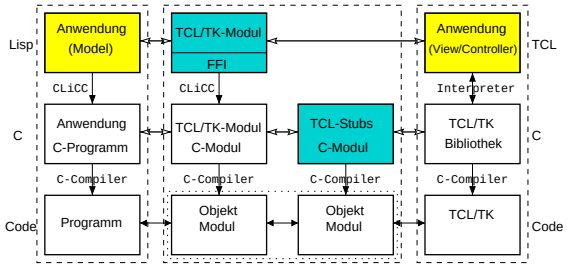

Abbildung 5: Die Tcl/TK-Schnittstelle mit dem Common-Lisp-Übersetzer CLiCC

5 Client-Server-Anwendungen

Mit zunehmender Vernetzung von Computern entsteht auch der Wunsch, KI-Anwendungen über Kommunikationsnetze zu benutzen. Als Standard hat sich die Internet-Kommunikation basierend auf dem Protokoll TCP/IP etabliert. Um KI-Anwendungen netzwerkfähig zu machen, ist es nicht zwangsläufig notwendig, diese in Java zu reformulieren, da zur Realisierung von verteilten KI-Anwendungen auf bestehende Internetdienste und -techniken zurückgegriffen werden kann. Üblicherweise basieren derartige verteilte Systeme auf dem Client-Server-Prinzip. Ein Server stellt die Anwendung für viele Clients bereit, die je nach Systemkonzeption auch Aufgaben selbständig erledigen können, aber in der Regel zur Visualisierung und Steuerung der Anwendung eingesetzt werden. Vom Server wird die gleichzeitige Bedienung von möglicherweise mehreren Clients verlangt, was einen entsprechenden Kontrollfluß (Prozesse, Threads oder sequentieller Dispatch) voraussetzt. Typische KI-Programmiersysteme haben über sog. foreign-function-interfaces Zugriff auf Kommunikations-Bibliotheken, etwa unter Unix auf die BSD-Socket-Bibliothek [9]. Mit ihrer Hilfe können KI-Anwendungen Kommunikation über das Internet vornehmen.

Basierend auf der BSD-Socket-Bibliothek gibt es Zusatzprogramme (etwa [17]), die die Standard-Ein/Ausgabe beliebiger Programme mit Kommunikations-Sockets verbinden können. Sie erlauben damit eine sehr einfache Anbindung von KI-Anwendungen an das Netz, wenn diese nur über Standard-Ein/Ausgabe kommunizieren. Die Server-Kommunikation mit mehreren Clients wird in diesem Fall durch das Gründen weiterer Prozesse durch das jeweilige Betriebssystem übernommen. Jeder Client erhält dadurch einen eigenen Server, der unabhängig von den restlichen Servern arbeitet. Geeignete Synchronisation beim Zugriff auf gemeinsame Daten ist natürlich zu gewährleisten.

Ein von überall über das Internet erreichbarer Server muß gewisse Sicherheitseigenschaften besitzen, um die Vertraulichkeit der von ihm direkt erreichbaren Daten zu gewährleisten. Eine netzwerkfähige KI-Anwendung wird, je nach Sicherheitsbedürfnis, Authentifizierungen und verschlüsselte Datenübertragungen verwenden. Bestehende Internet-Standards können übernommen werden, bzw. bieten gute Vorbilder.

KI-Anwendungen, die gemäß des Modell-View-Controller-Musters entworfen sind, besitzen natürliche Schnittstellen zwischen dem algorithmischen KI-Programmteil (Modell) der Anwendung und ihrer Bedienoberfläche (View und Controller). Um eine solche Anwendung netzwerkfähig zu machen, wird die Schittstelle zwischen Modell und View/Controller so definiert, daß einem Text-Anwendungs-Protokoll folgend über Text-Repräsentationen kommuniziert wird. Das Protokoll umfaßt sinnvollerweise neben dem Austausch von Nutzinformation auch die Behandlung von Authentifizierungs-, Diagnose- und von Fehlermeldungen. Die Schnittstelle zwischen Modell und View/Controller läßt sich in diesem Fall einfach über eine Kommunikationsnetz hinweg ausdehnen, d. h Modell und View/Controller können verteilt auf unterschiedlichen Computern ablaufen. Für die Definition von Text-Anwendungs-Protokollen gibt es unter den etablierten Internet-Protokollen zahlreiche Vorbilder, etwa SMTP [20] für die Übertragung elektronischer Post. Klartext-Protokolle bieten den Vorteil, daß die Kommunikation interaktiv mit Standardwerkzeugen überprüft werden kann. Sie schließen die Möglichkeit von Verschlüsselung und/oder Komprimierung kommunizierter Daten nicht aus.

Beispiel: Ein KI-Programmiersystem mit klassicher Text-Schnittstelle, und einer graphischen auf Tcl/TK [18] basierenden graphischen Bedienumgebung. Die Kommunikation zwischen Pro-

grammiersystem und Bedienumgebung erfolgt über ein Klartext-Protokoll. Eine verteilte Version dieses Systems erlaubt einem in Tcl/TK formulierten Bedien-Client via Internet die Verbindung mit dem Server aufzunehmen, der das Programmiersystem realisiert. Damit kann von beliebigen an das Internet angeschlossenen Rechnern dieses Programmiersystem bedient werden. Aus einem WWW-Browser heraus erfolgt dies durch ein Tcl/TK-plugin. Alternativ kann direkt ein entsprechendes Tcl/TK-Script eingesetzt werden, das auf dem Client abläuft.

6 Schlußbemerkungen

Wir haben ein Konzept zur Nutzung von Tcl/TK zur Implementierung ereignisorientierter graphischer Bedienoberflächen für Lisp-Anwendungen vorgestellt sowie Implementierungen einer Lisp-Tcl/TK-Schnittstelle vorgestellt, die sich gleichermaßen für den interaktiven Entwicklungsprozeß als auch für die Anwendungsgenerierung und Portierung eignen, ohne daß Code modifiziert werden muß. Tragendes Konzept ist die Verwendung von Standardkomponenten. Dies gilt auch für die Netzwerkfähigkeit derartiger Anwendungen auf der Basis von TCP/IP-Verbindungen.

Diese Arbeiten sind im Zusammenhang mit der Weiterentwicklung des in Kiel entwickelten Common-Lisp-Übersetzers CLiCC entstanden, motiviert durch eine Studie zur Anwendbarkeit von CLiCC, die wir in Kooperation mit dem DFKI durchgeführt haben [12]. In dieser Studie wurde CLiCC auf RELFUN [3] angewendet. In einem Experiment haben wir RELFUN mit dem in Abschnitt 5 beschriebenen Verfahren auch über das Internet verfügbar gemacht, allerdings ohne entsprechende Sicherheitsvorkehrungen, so daß diese Implementierung z.Zt. nicht permanent im Netz verfügbar ist. Es zeigte sich jedoch, daß man nicht notwendigerweise Java-Reimplementierungen komplexer Lisp-Software vornehmen muß, wenn man Forderungen nach Netzwerkfähigkeit gerecht werden will.

Ein herausragendes Merkmal von CLiCC ist, daß sich auch das Hauptprogramm als Modul in bindefähigen Objektcode übersetzen läßt. Damit können Lisp-Module auch in Nicht-Lisp-Anwendungen gebunden werden, eine Richtung der Integration, die klassische Lisp-Implementierungen bis heute nicht adäquat bieten. Dies ist in dem hier vorgeführten Standard-Vorgehen nicht ausgenutzt, läßt aber weitere Konzepte der Integration von Lisp-Anwendungen in Standard-Softwareumgebungen zu.

Auch andere Lisp-Implementierungen basieren auf C-Übersetzung: Am bekanntesten ist Austin Kyoto Common Lisp (AKCL), ein klassisches interaktives Common-Lisp-System, dessen inkrementeller Compiler auf Übersetzung nach C und dynamischem Nachladen des generierten Objektcodes beruht. Am ehesten ist unsere Zielsetzung vergleichbar mit G. Attardis „Embeddable Common Lisp“ (EcoLisp, [2]), wo ebenfalls das Hauptaugenmerk auf Einbettbarkeit gelegt wird.

Literatur

[1] Andreas Abecker, Harold Boley, Knut Hinkelmann, Holger Wache, and Franz Schmalhofer. An Environment for Exploring and Validating Declarative Knowledge. DFKI Technical Memo TM-95-03, DFKI GmbH, November 1995. Also in: Proc. Workshop on Logic Programming Environments at ILPS'95, Portland, Oregon, Dec. 1995.

[2] Giuseppe Attardi. The Embeddable Common Lisp. *LISP Pointers*, VIII(1):30–41, 1995.

[3] Harold Boley. Extended Logic-plus-Functional Programming. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the 2nd International Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, volume 596 of *LNAI*. Springer, 1992.

[4] Wilfried Brauer. KI auf dem Weg in die Normalität. *KI*, Sonderheft:85–91, August 1993.

[5] Harry Bretthauer, Thomas Christaller, Horst Friedrich, Wolfgang Goerigk, Winfried Heicking, Ulrich Hoffmann, Dieter Hovekamp, Heinz Knutzen, Jürgen Kopp, E. Ulrich Kriegel, Ingo Mohr, Rainer Rosenmüller, and Friedemann Simon. Das Verbundvorhaben APPLY: Ein modernes und bedarfsgerechtes Lisp. *KI*, 2:50–54, June 1992.

[6] Harry Bretthauer, Thomas Christaller, Horst Friedrich, Wolfgang Goerigk, Winfried Heicking, Ulrich Hoffmann, Andreas Kind, Bert Klude, Heinz Knutzen, Jürgen Kopp, E. Ulrich Kriegel, Ingo Mohr, Rainer Rosenmüller,

and Friedemann Simon. Von der APPLY-Methodik zum System. Abschlußbericht des BMFT-Projektes APPLY APPLY/XIII/10, CAU/GMD/ISST/VW-GEDAS, Sankt Augustin, June 1994.

[7] O. Burkart, W. Goerigk, and H. Knutzen. CLiCC: A New Approach to the Compilation from Common Lisp Programs to C. In *Workshop "Alternative Konzepte für Sprachen und Rechner"*, Bericht Nr. 8/91-I des Instituts für angewandte Mathematik und Informatik der Univ. Münster, 1991.

[8] Olaf Burkart. Das Frontend eines Compilers von Common Lisp nach C. Master's thesis, Institut für Informatik, CAU, Kiel, 1991.

[9] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume III: Client-Server Programming wand Applications, BSD Socket Version.* Prentice Hall, Engelwood Cliffs, NJ, 1993.

[10] Rolf Diekmann. Plattformübergreifender Entwurf ereignisorientierter Benutzeroberflächen für Lisp-Anwendungen: Konzept und technische Realisierung. Master's thesis, Institut für Informatik, CAU, Kiel, 1998.

[11] W. Goerigk and F. Simon. Zielsetzungen im Verbundvorhaben APPLY. In *Workshop "Alternative Konzepte für Sprachen und Rechner"*, Bericht Nr. 8/91-I des Instituts für angewandte Mathematik und Informatik der Univ. Münster, 1991.

[12] Wolfgang Goerigk, Harold Boley, Ulrich Hoffmann, Markus Perling, and Michael Sintek. Komplettkompilation von Lisp: Eine Studie zur Übersetzung von Lisp-Software für C-Umgebungen. *KI*, 10(2), 1996.

[13] Wolfgang Goerigk, Ulrich Hoffmann, and Heinz Knutzen. CommonLisp$_0$ and CLOS$_0$: The Language Definition. APPLY-Arbeitspapier APPLY/CAU/II.2/2, Institut für Informatik, CAU, Kiel, September 1993.

[14] Wolfgang Goerigk and Friedemann Simon. Migration und Kompilation in Lisp: Ein Weg von Prototypen zu Anwendungen. *Proc. Workshop "Neuere Entwicklungen der deklarativen KI-Programmierung"*, DFKI Research Report RR-93-35:31–52, September 1993.

[15] Chestnut Software Inc. Lisp-to-C Translator: Technical Specification. Technical report, Chestnut Software Inc., Boston, MA, 1991.

[16] Heinz Knutzen. Codegenerierung und Laufzeitsystem eines Compilers von Common Lisp nach C. Master's thesis, Institut für Informatik, CAU, Kiel, 1991.

[17] Jürgen Nickelsen. Socket-1.1. Free Software, August 1995.

[18] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, Reading Massachusetts, 4 edition, 1994.

[19] John K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, Reading, MA, USA, 1993.

[20] J. B. Postel. RFC 821: Simple Mail Transfer Protocol. Technical report, University of Southern California Information Sciences Institute, August 1982.

[21] G.L. Steele. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

[22] G.L. Steele. *Common Lisp: The Language. Second Edition.* Digital Press, Bedford, MA, 1990.

[23] P. H. Winston and B. K. P. Horn. *Lisp Second Edition.* Addison-Wesley, Reading, 1984.

Die $\mathcal{MORE}$ Architektur

Ein Objekt-Orientiertes Architekturmodell für Komplexe und Netzwerkfähige Softwaresysteme.

Abdel Kader Diagne
DFKI GmbH
Stuhlsatzenhausweg 3
66123 Saarbrücken
Deutschland
diagne@dfki.de

1 Einleitung

Die Architektur eines komplexen Systems ist maßgebend dafür, inwieweit bzw. wie schnell und resourcenschonend dieses an veränderte Anforderungen angepaßt werden kann. Eine solche aktuelle Anforderung ist beispielsweise die erleichterte Integration und Nutzung nicht nur von komplexen Softwaresystemen sondern auch von Softwarekomponenten im Netz.
In dieser Arbeit wird das objekt-orientierte Architekturmodell $\mathcal{MORE}$ [1] beschrieben, das die Entwicklung und Implementierung von komplexen aber flexiblen Systemen, sowie deren Verteilung und Bereitstellung im Netz erheblich erleichtert.
Hauptmerkmal des $\mathcal{MORE}$ -Modells ist das REQUEST-MANAGER-RESPONSE Architekturmuster (architectural pattern), das in dieser Arbeit eingeführt wird. Der Ansatz besteht darin, daß jede Komponente in einem sog. *Manager* eingekapselt wird, der diese nach außen als Server erscheinen läßt. Ebenso werden Input- und Outputspezifikationen durch *Request*- und *Response*-Klassen definiert, d.h. konkrete Ein- und Ausgabedaten von Komponenten in Objekten eingekapselt.
Ergänzt wird dieser Ansatz einerseits durch die Spezifikation einer generischen Serverschnittstelle, GSI (Generic Server Interface), für die Kommunikation zwischen Client und Server, und andererseits durch die Einführung des Konzepts einer virtuellen Systemarchitektur, VSA (Virtual System Architecture), die es einem Server ermöglicht mehrere Aufträge parallel zu verarbeiten.
Die $\mathcal{MORE}$ -Architektur wurde bereits in verschiedenen Systemen erfolgreich implementiert [2], [3]. Diese Implementierungen werden wir weiter als Beispielanwendungen benutzen, um einige Varianten der Architektur vorzustellen.
Desweiteren werden Erweiterungs- und Anpassungsmöglichkeiten der Architektur angesprochen und sowie Anforderungsprofile für geeignete Systeme gegeben. Zum Schluß wird das $\mathcal{MORE}$ -Modell in die Landschaft der gängigen Architekturschemata [6] eingeordnet und entsprechend bewertet.

[1] $\mathcal{MORE}$ steht für Manager-Based, Object-Oriented, and Request/Response-Based

2 Motivation und Konzept

Die $\mathcal{MORE}$ -Architektur resultiert aus einer Reihe von Systementwicklungsaktivitäten, bei denen die Entwicklung heterogener Systeme im Vordergrund stand, die bestimme Mindestanforderungen erfüllen sollten:

- Bereits existierende Komponente und Subsysteme sollten integriert werden.
- Komponenten sollten leicht austauschbar sein.
- Komponenten können in verschiedenen Programmiersprachen implementiert werden und auf unterschiedlichen Rechnern laufen.
- Das system sollte als Server im Netz verfügbar sein und sollte in der Lage sein, mehrere Aufträge gleichzeitig zu verarbeiten.

$\mathcal{MORE}$ ist vom Konzept her objekt-orientiert. Das Hauptmerkmal des Architekturschemas wird durch das Architekturmuster (architectural pattern) Request-Manager-Response beschrieben. Daher liegt es nahe, die Architektur nach dem *Context-Problem-Solution*-Schema [1], zu präsentieren. Es werden zuerst allgemeine, bekannte, Design-Situationen *Context* angegeben, in denen bestimmte Anforderungen an die Systemarchitektur gestellt werden bzw. in denen bestimmte Design-Probleme (*Problem*) auftreten, für deren Bewältigung ein Lösungsschema (*Solution* vorgestellt wird. Die Struktur des Patterns wird dann als UML-Diagram dargestellt, und Beispielanwendungen, sowie Architekturvarianten werden vorgestellt.

3 Anwendungskontext

Es wird hier nicht den Versuch unternommen, alle möglichen Design-Situationen anzugeben, in denen die $\mathcal{MORE}$ -Architektur eingesetzt werden kann. Die bisherigen Erfahrungen haben jedoch gezeigt, daß insbesondere skalierbare, heterogene und netzwerkfähige Systeme einen geeigneten Kontext für den Einsatz ihren bieten.

4 Anforderungsprofil

In diesem Abschnitt wird, ohne Anspruch auf Vollständigkeit, wesentliche Anforderungen an Systemarchitekturen aufgelistet, die sich mit der hier vorgestellten Architektur sehr gut bewältigen lassen.

- Bereits existierende Komponente und Subsysteme sollten integriert werden.
- Komponenten können in verschiedenen Programmiersprachen implementiert werden und auf unterschiedlichen Rechnern laufen. Sie sollten eine stabile Schnittstelle aufweisen und leicht austauschbar sein.
- Das System muß als Server im Netz verfügbar sein und muß in der Lage sein, mehrere Aufträge gleichzeitig zu verarbeiten.
- Sowohl asynchrone als auch inkrementelle Verarbeitung müssen unterstützt werden.

- Die Kommunikation zwischen Komponenten sollte auf Client/Server-Basis funktionieren und darf nicht von einem einzelnen IPC-Protokoll (Inter-Process Communication) abhängen.
- Nicht objekt-orientierte Subsysteme müssen in einem objekt-orientierten System integriert werden.
- Das System muß erweiterbar, anpaßbar dynamisch konfigurierbar sein.
- Eine wissensbasierte Kontrolle bzw. Steuerung des Systems sollte möglich sein.

5 Ansatz

Der Lösungsansatz basiert auf folgender Leitidee:

- Für jede Komponente wird ein *Manager* definiert, das in erster Linie ein Object darstellt, das bestimmte Koordinations- und Kontrollaufgaben im System übernimmt. Er wird durch die Spezifikation seiner Klasse, seiner Verantwortung im System (Dienste, die er zur Verfügung stellt) und durch die Angabe der Komponenten bzw. der Manager, mit denen er kommuniziert, definiert.

  Manager kapseln Komponenten ein und lassen sie nach außen als Server erscheinen Dadurch werden Zugriffe auf die Komponente durch Zugriffe auf den Manager ersetzt, der dem Client eine stabile und anforderungsorientierte Schnittstelle zur Verfügung stellt. So kann sich die Schnittstelle der eigentlichen Komponente ändern bzw. die Komponente kann sogar ausgetauscht werden, ohne daß die Schnittstelle nach außen sich ändert; entsprechende Anpassungen werden dann vom Manager vorgenommen.

  Ein Manager ist auch ein Mediator [4]. Er reduziert gegenseitige Abhängigkeit von Komponenten im System und regelt die Kommunikation zwischen interagierenden Komponenten, sodass diese nicht explizit aufeinander referieren. Dadurch werden komplexe Interaktionsprotokolle aus den Komponenten genommen. Diese werden auf ihre Kernfunktionalität reduziert. So haben Änderungen meistens nur eine lokale Wirkung, und die Interaktionsprotokolle können oft ohne Änderung der beteiligten komponenten variiert werden.

- Für jede Komponente wird eine *Request*-Klasse definiert, die Inputspezifikation enthält. Die Klassendefinition kann Steuerungs- und Konvertierungsoptionen und weitere Methoden enthalten. Eine Komponente kann, je nach Client, Anzahl und Art der geleisteten bzw. geforderten Dienste, mit mehreren *Request*-Klassen bzw. Subklassen assoziiert werden.

  Ein *Request* stellt eine Erweiterung des *Command-Pattern* dar [4], das ebenfalls darauf basiert, Parameter in Objekte einzukapseln, um die Flexibilität des Systems zu erhöhen: Priorisierungen von Anfragen (z.B. Queuing), Abspeicherung von Prozessinformationen (Logs), Rückangigmachen von Verarbeitungsschritten (Undo), Callback, etc.

- Für jede Komponente wird eine *Response*-Klasse definiert, die Outputspezifikation enthält.

  Ein *Response*-Object enthält (eine Referenz auf) das *Request*-Objekt, dessen Lösung es darstellt. Damit wird eine Anfrage zusammen mit ihrer Lösung und alle relevante Schritte des Problemlösungsprozesses in einem einzigen Objekt einkapselt.

5.1 Struktur des Request-Manager-Response -Patterns

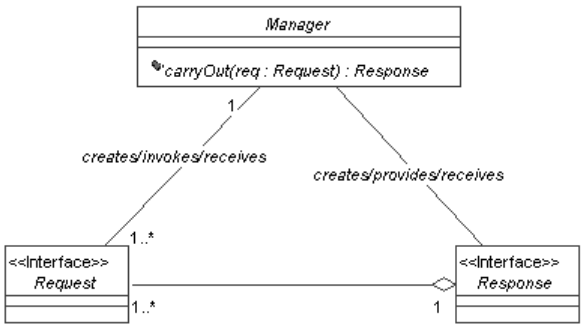

Abbildung 1: **Basisstruktur des Request-Manager-Response -Patterns**

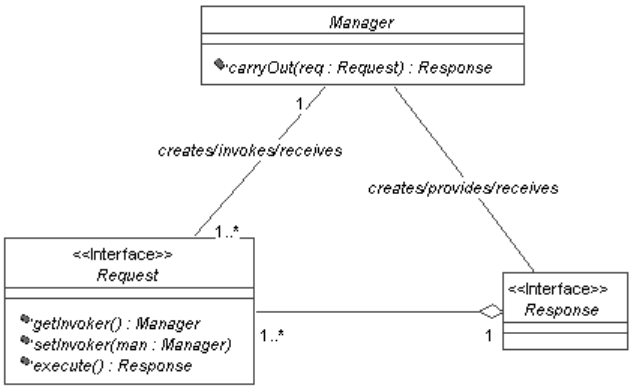

Abbildung 2: **Variante 1: keine direkte Interaktion zwischen Managern**

Request, *Response* und *Manager* sind abstrakte Klassen. Komponenten werden in Managern eingekapselt, und diese kommunizieren unter sich auf Client/Server-Basis durch Absenden und Empfangen von Requests. Für konkrete Komponenten und Schnittstellenspezifikationen werden entsprechende Subklassen definiert. In der Basisversion (Abbildung 5.1) haben interagierende Manager direkte Referenzen aufeinander. Dies ist aber nicht immer der Fall. Die in den Abbildungen 5.1 und 5.1 dargestellten Varianten erlauben einen höheren Grad an Unabhängigkeit zwischen Managern. Da müssen Manager keine Referenzen auf Interaktionspartner haben. Jedes *Request*-Objekt enthält relevante Informationen über Auftraggeber und Auftragnehmer. So werden Anfragen und erzeugte Lösungen vom Anfrage-Objekt selbst weitergeleitet. Variante 2 (Abbildung 5.1 unterscheidet sich von Variante 1 (Abbildung 5.1, indem sie asynchrone Verarbeitung über *Callback* unterstüzt. Der Auftraggeber (Client) muß nicht auf eine Anwort warten, sondern diese wird durch eine Callback-Funktion (*receive()*) geliefert. Variante 1 eignet sich aber besser für den Einsatz von objekt-orientierter

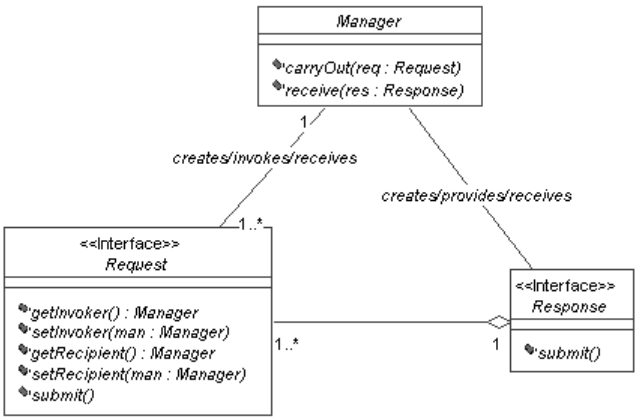

Abbildung 3: **Variante 2: Interaktion mit Callback**

Middleware wie CORBA oder RMI, da in den meisten Fällen lediglich *Request* und *Response* als Top-Level-Remote-Objekte gebraucht werden.

5.2 Die generische Serverschnittstelle GSI

Um die Kommunikation mit Systemen im Netz zu erleichtert wurde die $\mathcal{MORE}$ -Architektur um ein Kommunikationskonzept erweitert, der darauf abzielt, den Overhead der Kommunikationsverwaltung aus den Fachkomponenten des Systems rauszuhalten und gleichzeitig eine Plattform für den Einsatz verschiedener Inter-Prozess-Kommunikationsprotokolle zu bieten. Dafür werden die speziellen Module CCI und CCL definiert.

5.2.1 Die Kommunikationsschnittstelle CCI

CCI ist objekt-orientiert und definiert einen allgemeinen Rahmen für die Inter-Prozeß-Kommunikation. Der Grungansatz besteht darin, sog. Kommunikationsschnittstellenobjekte (CCI -Objekte) bzw. -Klassen zu definieren, die dann für bestimmte Protokolle weiter spezialisiert werden. Ein CCI -Objekt stellt eine Reihe von Mechanismen zur Verfügung, die erlauben, eine Kommunikation zwischen verteilten Systemen und Komponenten komplett zu verwalten. Einige dieser Mechanismen sind:

- die Priorisierung von Aufträgen,
- die Benutzung von Ein- und Ausgabepuffer,
- die Authentifizierung von Clients und die Festlegung von Service-Codes,
- die Hierarchisierung von Kommunikationsstufen: Sitzung, Auftrag, Aufgabe, etc.

Um ein (Sub)System zu einem im Netz verfügbaren Server zu Verwandeln, braucht man lediglich ein CCI -Objekt zu erzeugen, das dem System zugeordnet wird. Dieses Objekt verwaltet dann alle Kommunikationsaktivitäten.

Folgender Common Lisp/CLOS Code-Ausschnitt zeigt eine Beispielimplementierung einer allgemeinen CCI -Klasse mit einer spezialisierten Subklasse, die auf TCP/IP-basierte Kommunikation über Sockets basiert:

```
(defclass cci* ()
  ((name
    :reader name :initarg :name :initform :cci-0)
   (host
    :reader host :initarg :host :initform (cw::hostname))
   (address
    :reader address :initarg :address :initform nil)
   (input-buffer
    :reader i-buffer :initarg :i-buffer :initform nil)
   (output-buffer
    :reader o-buffer :initarg :o-buffer :initform nil)
   (input-counter
    :reader i-counter :initform 0 :type integer)
   (output-counter
    :reader o-counter :initform 0 :type integer)
   (verbose-p
    :reader cci-verbose-p :initform nil)
   (stream
    :reader cci-stream :initarg :stream :initform T) )
  (:documentation
   "Foundation class for a client/server communication interface (cci)."
   ))

(defclass server-cci* (cci*)
  ((clients ;; (<name> <address>)
    :reader clients :initarg :clients :initform nil))
  (:documentation "Server side of cci." ))

(defclass client-cci* (cci*)
  ((service-channel ;; <address>
    :reader service-channel :initarg :service-channel :initform nil))
  (:documentation "Client side of cci." ))

(defstruct (streamsock-daemon-process (:copier nil) (:include mp::process))
  "Embodies the leightweight lisp process for accepting clients' requests for
   connection."
  cci )

(defclass streamsock-basic-cci* ()
  ((protocol-identifier
    :reader protocol-identifier :initarg :protocol-identifier
    :initform "StreamSock~N\n"
    :allocation :class)
```

```
   (protocol-version
    :reader protocol-version :initarg :protocol-version
    :initform 0 :allocation :class)
   (protocol-version-size
    :reader protocol-version-size :initarg :protocol-version-size
    :initform 1 :allocation :class)
   (tcp-protocol-number
    :reader the-tcp  :allocation :class)
   (socket-descriptor ;; the socket's descriptor 'sd'
    :reader socket-descriptor :initform nil)
   (service-id
    :reader service-id :initarg :service-id :initform "NLS!Connect" )
   (connection-counter
    :reader connection-counter :initform 0)

   ;; Constants from <sys/socket.h>
   (af-inet
    :reader af-inet :initform 2 :allocation :class)
   (sock-stream ;; :initform is 1 on SunOS 4.x.y (e.g. 4.1.1)
    :reader sock-stream :initform 2 :allocation :class)
   (sol-socket
    :reader sol-socket  :initform #xffff :allocation :class)
   (so-reuseaddr
    :reader so-reuseaddr :initform #x4 :allocation :class)
   (tcp-nodelay
    :reader tcp-nodelay :initform #x01 :allocation :class)

   ;; Constants from <netinet/in.h>
   (inaddr-any
    :reader inaddr-any :initform 0 :allocation :class)

   ;; Constants from <errno.h>
   (eintr ;; interrupted system call
    :reader eintr :initform 4 :allocation :class)

   ;; Constants from <fcntl.h>
   (f-setfl
    :reader f-setfl :initform 4 :allocation :class)
   (o-nonblock
    :reader o-nonblock :initform #x4000 :allocation :class)
   (info-file
    :reader info-file :initarg :info-file)
   ))

(defclass streamsock-server-cci* (server-cci* streamsock-basic-cci*)
  ((streamsock-daemon
    :reader streamsock-daemon :initarg :streamsock-daemon :initform nil
```

```
    :type streamsock-daemon-process)
   (daemon-sleep-time
    :reader daemon-sleep-time :initarg :daemon-sleep-time :initform 1
    :documentation "Sleep time after accepting a client's request.")
   (daemon-idle-p
    :reader daemon-idle-p :initarg :daemon-idle-p :initform nil
    :documentation "if T daemon goes in an idle state and doesn't poll for new
                    connect requests.")
   (port
    :reader port :initarg :port :initform 10848 :type integer)))

(defclass streamsock-client-cci* (client-cci* streamsock-basic-cci*)
  ((service-host
    :reader service-host :initarg :service-host :initform nil)
   (service-port
    :reader service-port :initarg :service-port :initform nil)
   (server-name
    :reader server-name :initarg :server-name :initform nil)))
```

5.2.2 Die Spezifikationssprache CCL

CCL ist eine deklarative und Client/Server-basierte Aufgabenbeschreibungssprache. Die grundlegende Idee besteht in der Trennung Kommunikationsprotokoll und kommunizierte Daten. Die vorhandene Version von CCL ist eine Weiterentwicklung einer deklarativen Spezifikationssprache, die für Anwendungen im Bereich der Sprachverarbeitung konzipiert wurde, wird aber hier zur Veranschaulichung des Konzepts herangezogen. Sie basiert auf Merkmalstrukturen und erlaubt die Spezifikation von Aufträgen in Form von Attribut-Wert-Paaren. Primär werden lediglich Ausgangsdaten und erwartete Zielinformation beschrieben. Der Server verfügt dann über einen Workflow-Manager, der aus den gegebenen und erwünschten Daten einen Prozessablauf erzeugt und ausführt. Folgendes Beispiel zeigt eine Anfrage in der aktuellen Implementierung, in der ein Text gegeben wird, dessen semantischen Repräsentation erzeugt werden soll:

```
[(dialogue-id 1) (task-id 3) (text ``Wir treffen uns am 12.10.98 um 15 Uhr.'')
(sem ?)]
```

5.3 Die virtuelle Systemarchitektur VSA

Ein virtuelles System ist ein (objekt-orientiertes) Aggregat, das dadurch entsteht, daß Komponenten über Manager-Objekte angesprochen werden. Durch die Einfürung von Mechanismen, die die Definition und Verwaltung von 1:N-Beziehungen zwischen komponenten und Managern erlauben, können mehrere Instanzen eines Systems entstehen, die hier als virtuelle Systeme bezeichnet werden. Denn tatsächlich existieren nicht mehrere Intanzen von reellen Komponenten, sondern mehrere Manager teilen sich die Dienste einer Komponente. Es werden Sperrmechanismen eingeführt, die verhindern müßen, daß mehrere Manager gleichzeitig auf dieselbe Komponente zugreifen. Durch den Einsatz von virtuellen Systemen können mehrere Tasks gleichzeitig verarbeitet werden, indem jedem Auftrag ein viruelles System zugewiesen wird.

6 Anwendungsbeispiele

In diesem Abschnitt werden zwei Anwendungen der $\mathcal{MORE}$ -Architektur für die Realisierungen von großen und komplexen Online-Systemen kurz vorgestellt.

6.1 Cosma

6.1.1 Kontext

COSMA ([2]) ist ein agenten-basiertes, natürlichsprachliches System für die email-basierte Terminvereinbarung. Kernkomponenten von COSMA sind ein Agentensystem und ein Sprachverarbeitungssystem, der als Online-Server fungiert.

6.1.2 Anforderungen an die Systemarchitektur

Einige wichtige Anforderungen an die Architektur lauten wie folgt:

- Ein hohes Grad an Wiederverwendung von NLP Ressourcen und Verarbeitungsmechanismen sollte gewährleistet werden.
- Die Schnittstelle sollte robust sein, und deklarative Aufgabenspezifikation sollten möglich sein.
- Clients sollten über eine Programmierschnittstelle (API) den Server konfigurieren können.
- Inkrementelle, verteilte und asynchrone Verarbeitung müßen unterstützt werden. Insbesondere sollte inkrementelles Verarbeiten simulierbar sein.
- Parallele Dialogverarbeitung muß gewährleistet werden.
- Das System muß als Server im Nezt vefügbar sein.

6.1.3 Lösungsansatz und Realisierung

In COSMA wurde eine eher manager-zentriete Version des REQUEST-MANAGER-RESPONSE -Patterns implementiert, bei der Manager direkt miteinander kommunizieren. Das System besteht aus einem *Team von Managern*, das von einem *Workflow-Manager* koordiniert wird. Dieser analysiert deklarative Benutzeranfragen, zerlegt sie in Teilaufgaben, ordnet diese einzelnen Managern zu und koordiniert der Problemlösungsprozeß.
Komponentenmanager in COSMA haben lokale Zwischenspeicher, die sie verwenden, um Ausgaben von Komponenten abzulegen. Dadurch können sie Ergebnisse selektiv oder stückchenweise weitergegeben werden, und auf dieser Weise, nicht-inkrementelle Komponenten inkrementell erscheinen yu lassen.
GSI und VSA wurden in COSMA implementiert. Dadurch wurde es möglich unterschiedliche Dialoge von mehreren Agenten gleichzeitig zu verarbeiten, bzw. mehrere parallel geführte Dialoge von einem einzelnen Agenten zu verwalten. Jeder Dialog wird von einem einzelnen vituellen System verarbeitet.
Die $\mathcal{MORE}$ -Architektur von COSMA wurde in Common Lisp und CLOS implementiert.

6.2 Mulinex

6.2.1 Kontext

MULINEX ist ein translinguales Information-Retrieval-System für das Internet, das auch die Zusammenfassung, Kategorisierung und Übersetzung von Dokumenten ermöglicht.

6.2.2 Anforderungen an die Systemarchitektur

Neben einer ständigen Online-Verfügbarkeit, einer hohen Performanz und eine ausgewiesene Robustheit des Systems sind noch folgende wichtige Anforderungen zu nennen:

- Das System muß leicht erweiterbar und anpaßbar sein.
- Neue Komponenten sollen hinzugefügt bzw. vorhandene entfernt werden können, ohne deren Interaktionspartner im System ändern zu müssen.
- Schnitstellen müßen stabile und dennoch leicht erweiterbar sein.

6.2.3 Lösungsansatz und Realisierung

In MULINEX wurde in einem ersten Prototyp Variante 2 des REQUEST-MANAGER-RESPONSE -Patterns mit *Callback* implementiert. Später wurde Variante 1 übernommen, da viele Komponenten verteilt sind und mittlerweile über die Java Middleware-Lösung RMI kommunizieren. Miteinander kommunizierende Manager arbeiten autonom und haben keine direkte Referenz zueinander. Ein Client braucht lediglich eine Anfrage durch Instantiierung eins *Request* zu formulieren, und diese über das Objekt selbst ausführen zu lassen. Jedes *Request*-Objekt pflegt relevante Informationen über den für seine Verarbeitung zuständigen Manager.

Für die Realisierung von CORBA- oder RMI-basierte Lösungen für die verteilte Verarbeitung werden lediglich *Request* und *Response* als *remote objects* definiert.
Die $\mathcal{MORE}$ -Architektur von Mulinex ist komplett in Java implementiert.

7 $\mathcal{MORE}$ und andere Architekturschemata

Unterschiedliche Architekturschemata haben unterschiedliche Wirkungen auf die Qualität und Performanz eines Systems. Daher sollte die Verwendung von Architekturmodellen zu einem besseren Verständnis von Systemen verhelfen, genauso wie das Verstehen von Systemarchitekturen es erst möglich macht, gute Systeme zu entwickeln, die helfen können, praktische Probleme zu bewältigen. In diesem Sinne wurde die $\mathcal{MORE}$ -Architektur konzipiert. Sie definiert, im Gegensatz zu gängigen Architekturschemata wie Pipes & Filters, Blackboard-, und Schichtenarchitekturen (Layers) kein starres Strukturierungsmodell, sondern bietet eher Konzepte und Methoden für die Erhöhung der Flexibilität, Modularität, Wiederverwendbarkeit und Erweiterung von Systemen und Komponenten an, sowie für ein besseres Verständnis des zugrundeliegenden Systems und seiner Anwendungsdomäne.
Die $\mathcal{MORE}$ -Modell steht nicht unbedingt in einem Entweder-Oder-Verhältnis zu anderen Architekturschema, sondern es ergänzt sie in vielen Fällen. So kann ein Schichtenmodell als $\mathcal{MORE}$ -Architektur implementiert werden, dadurch daß Schichten und Komponenten in Managern eingekapselt werden. Das gleiche gilt für die meisten Repository-basierte Architekturen (z.B. Blackboard) und Andere, wie einige Experimente bereits gezeigt haben.

Wie Manager strukturiert werden und wie sie interagieren, bzw. ob, wie und inwieweit *Request*- und *Response*-Objekte strukturiert und eingesetzt werden, hängt von den vorliedengen Systemanforderungen ab.
Das $\mathcal{MORE}$ -Architekturmodell bringt aufgrund des objekt-orientierten REQUEST-MANAGER-RESPONSE -Patterns alle Vorteile der Objekt-Orientierung mit sich: Vererbung, Polymorphismus, Einkapselung der Daten, Wiederverwendbarkeit, etc. Aber gerade deswegen ist dieses Modell meistens nur dort sinnvoll einsetzbar, wo ein Objekt-Orientierung auch Sinn macht.
Weitere heterogene und komplexe Systeme werden momentan basierend auf der $\mathcal{MORE}$ -Architektur entwickelt. Diese werden mit Sicherheit einige Erweiterungen und Konsolidierung mit sich bringen. Das Kernkonzept von $\mathcal{MORE}$ wird weiterhin aber daurauf beruhen, Input, Output und Verarbeitung differenzierter zu behandeln.

Literatur

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stahl. *Pattern-Oriented Software Architecture: A System of Patterns.* Addison-Wesley, 1996.

[2] Stephan Busemann, Thierry Declerck, Abdel Kader Diagne, Luca Dini, Judith Klein, and Sven Schmeier Natural Language Dialogue Service for Appointment Scheduling Agents. Proceedings of Fifth Conference of Applied Natural Language Processing, Washington DC., USA, April 1997, pp. 25–32. Available on the Net at the Computation and Language Archive ("http://xxx.lanl.gov/abs/cmp-lg?9702007") and as DFKI Research Report RR-97-02, February 1997.

[3] Gregor Erbach, Günter Neumann and Hans Uszkoreit. Mulinex (http://mulinex.dfki.de) Multilingual Indexing, Navigation and Editing Extensions for the WWW. AAAI Spring Symposium on Cross-Language Information Retrieval, Menlo Park, CA, 1997, pp. 22-28 [also in: Proceedings of the Third DELOS workshop – Cross-Language Information Retrieval, European Research Consortium for Informatics and Mathematics, 1997, pp. 17-24]

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] T. W. Malone and K. Crowston. Toward an interdisciplinary theory of coordination. Technical report ccs tr 120, Center for Coordination Science, Sloan School of Management, MIT, MIT, Cambridge, MA, 1991.

[6] Mary Shaw, David Garlan. Software Architecture. Perspectives on an Emerging Discipline. Prentice Hall, New Jersey 1996.

[7] UML, The Unified Modeling Language (http://www.rational.com/uml/index.shtml). OMG, UML Specification, November 19th 1997

Techniques and Rule Patterns for Declaratively Querying Web Data with FLORID

Bertram Ludäscher Rainer Himmeröder Wolfgang May
Institut für Informatik, Universität Freiburg, Germany
{ludaesch,himmeroe,may}@informatik.uni-freiburg.de

Abstract

FLORID is an implementation of the deductive object-oriented database language F-logic and has recently been extended to provide a declarative semantics for querying the Web. By means of several illustrative examples, we show how FLORID's rule-based logical language can be used to extract, query, and analyze data from the Web.

1 Introduction

Models and languages for querying the Web, for handling semistructured data, and for integration and restructuring of information have recently attracted a lot of interest [MV98]. We argue that *dood* languages, i.e., supporting deductive and object-oriented features, are particularly suited in this context: *Object-orientation* provides a flexible common data model for combining information from heterogeneous sources and for handling partial information. Techniques for navigating in object-oriented databases can be applied to semistructured databases as well, since the latter may be viewed as (very simple) instances of the former. *Deductive rules* provide a powerful framework for expressing complex queries in a high-level, declarative programming style. WebLog [LSS96] and ADOOD [GMNP97], for example, build upon the dood language F-logic [KLW95]. FLORID [FLO] is an implementation of F-logic, which has been extended to provide a declarative semantics for querying the Web. The proposed extension allows extraction and restructuring of data from the Web and a seamless integration with local data. A main advantage of the approach is that it brings together the above-mentioned issues in a unified, formal framework and supports rapid prototyping and experimenting with all these features. In particular, FLORID programs may be used (simultaneously) as wrappers, mediators, or for "ordinary" deductive queries.

In this paper we illustrate, by means of several examples, the different techniques and rule patterns for declaratively querying the Web with FLORID. The examples substantiate the claim that a dood framework is suited for querying and management of semistructured and/or Web data. In Section 2, we briefly introduce the basics of F-logic, FLORID's Web model, and the data-driven style of accessing Web documents. Section 3 deals with *structure-based* queries, i.e., involving only the hyperlink structure of Web documents. In contrast, Section 4 focuses on *content-based* queries, i.e., where also the textual representation of documents has to be analysed in order to extract data (in general, FLORID programs involve both structure and contents of Web documents). Some concluding remarks and further FLORID references are given in Section 5.

2 Exploring the Web with FLORID

We briefly review the basic constructs of F-logic and its extension by path expressions.

A Glimpse of F-Logic. Consider the following fragment of an F-logic program:

```
person[name ⇒string; children@(integer) ⇒⇒person].      % signature of class person
employee::person.                                        % subclass relationship
john:employee[                                           % instance relationship and
    name →"John Smith"; children@(1998) →→{mary,bob}].   % ... some example data
X.father:man ←X:person.                                  % object creation by ...
X.mother:woman ←X:person.                                % ... path expressions
```

First, the *signature* of class person is specified: The *single-valued* method name yields instances of class string, whereas the *multi-valued* (and parameterized) method children yields instances of person. The *subclass* relation employee::person states that all members of employee are also members of person. Next, john:employee defines that the object named john is an *instance* of class employee; the specification inside [...] defines the actual data values for name and children.

Path Expressions and Object Creation. The last two rules demonstrate how path expressions in the head can be used to create new object identifiers (oids): If X is bound to an instance of person, then the single-valued method father becomes defined for X. The newly "created" father is referenced by the *path expression* X.father and is made an instance of man. In particular, john[father→john.father] and (john.father):man hold (similarly for mother and woman). Thus, the dot "." corresponds to navigation along single-valued methods (→) like name and father, while ".." is used to navigate along multi-valued methods (→→) like children, e.g., as in john..children@(Y)[surname→john.surname].

The use of path expressions for object creation is crucial for our approach to data-driven exploration of the Web using F-logic. Object creation has to be used with care in order to avoid infinite universes and nontermination: e.g., if a rule X.father:man ← X:man were added to the program, an infinite number of objects like john.father, john.father.father, etc. would be created.

2.1 The Web Model

As usual, we conceive the Web as a graph-like structure consisting of documents and links between them. More precisely, we distinguish between the class url of (potential) urls, and webdoc of (accessed) Web documents (Fig. 1). Urls are instances of string, for which a special method get is definable (see below). Typical elements of class webdoc are HTML pages, but other document types may also be included (e.g., BibTEX). In particular, one may define a subclass sgmldoc such that the parse-tree of fetched SGML documents can be analyzed:[1] If a Web document has been accessed, a number of *system-defined* methods may become defined for it: e.g., url (the url of the Web document), modif (time of last modification), type, and—most notably—the multi-valued method hrefs@(*label*) representing the outgoing links of the Web document (see Fig. 1). Note that hrefs is parameterized with the label of the link. If the Web access fails, error returns the reason of the failure (e.g., *page not found*).

[1] This feature is currently being incorporated into FLORID.

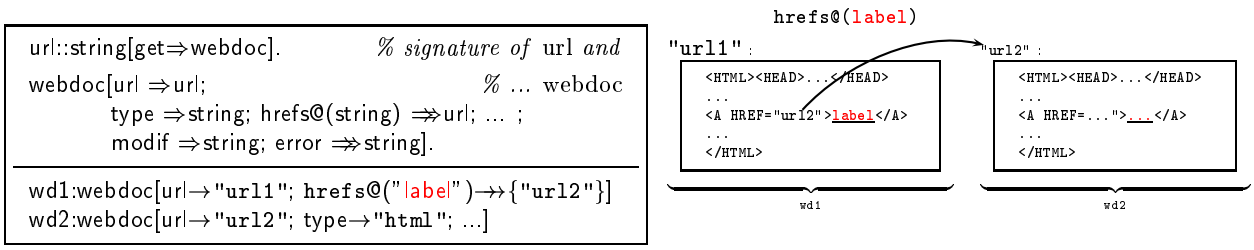

Figure 1: F-logic Web model: signature and example data

2.2 Data-Driven Web Exploration

A Web document is accessed and added to the local F-logic database by defining the method get for an instance u of class url in the head of a rule, thereby creating the new oid "u.get" of the fetched Web document. After the oid u.get has been created, system-defined methods are automatically "filled in" by FLORID, and the Web document named u.get becomes an ordinary F-logic object (conceivable as a large string). Thus, u.get is "cached" and the url u is accessed only once. Note that the *potentially* system-defined methods for class webdoc are given by the FLORID system—the *actually* system-defined (i.e., "filled") methods for u.get depend on the result of accessing the url u. For example, if error is defined, then hrefs@(*label*) will be undefined. Here, the advantages of using an object-oriented framework like F-logic become apparent: although the instances of a certain class may typically define a certain set of methods, some (or all) of these methods may be omitted. Thus, the use of NULL values as in the relational model can be avoided.

Apart from this first analysis of the document done by the system, the main power of the approach lies in the possibility to specify arbitrary *user-defined* methods for fetched documents using all features of F-logic (and path expressions). Consider, e.g., the following F-logic program:

```
"http://www.informatik.uni-freiburg.de/~dbis/":url.get.      % (1) define and access start url
U:explored ←U:url.get.                                       % (2) remember explored documents
U1.get ←U:explored.get[hrefs@(Lbl) →→{U1}],                  % (3) transitively access url U1 ...
        substr(U,U1).                                        %  ... if U is a prefix of U1
```

The path expression in (1) defines a particular string as an instance of url and, since the method get is defined for this url, accesses the corresponding document. Due to (2), the urls of all accessed documents become instances of class explored. (3) uses the power of deduction with recursion and transitively accesses all documents reachable from an already explored url U, provided that the referenced url U1 is a substring of U. Thus, only urls starting with http://...~dbis will be accessed; for other links, get remains undefined. Therefore, explored and unexplored urls can be distinguished using the queries ?-U:explored and ?-U:url, not U:explored, respectively. In particular, only a finite number of new oids u.get is created.

Since exploration of the Web is completely data-driven, a seamless integration with the bottom-up evaluation strategy of FLORID and a declarative semantics of the language is achieved [HLLS97, LHL+98].

3 Querying Structure

When considering Web documents, one may distinguish between *structure* (i.e., how pages are interlinked) and *contents* (i.e., the actual data provided on pages; see Section 4). The hyperlink structure of a collection of Web pages (cf. Fig. 2) can be explored in FLORID by navigating along the multi-valued method hrefs@(*label*). Indeed, the link structure of a set of Web documents is the prototypical example of what is called a *semistructured database* (*ssdb*), but also the data found on individual pages can often be considered as semistructured.

3.1 Semistructured Databases

The enormous success of the Web has recently lead to an increasing interest in models and languages for *ssd* [Abi97, AQM+97, BDHS96, Suc97]. Typical features attributed to *ssd* include the following: the structure is irregular, partial, unknown, or implicit in the data, and typing is not strict but only indicative [Abi97]. Since the distinction between schema and data is often blurred, semistructured data is sometimes called "self-describing" [Bun97].

In general, there may be very little structure in semistructured data, or the structure may be contained within the data and has to be discovered. Therefore, the underlying data model has to be simple and flexible. Here, we adopt the fairly standard model where *ssd* is represented as a labeled graph:

Definition 1 Let $\mathcal{N}$ be a set of *nodes* and $\mathcal{L}$ a set of *labels*. A *semistructured database*[2] (*ssdb*) $\mathcal{D}$ is a finite subset of the set of *labeled edges* $\mathcal{E} = \mathcal{N} \times \mathcal{L} \times \mathcal{N}$. □

Like in other object-oriented data models, data in an F-logic database instance can be conceived as a labeled graph: e.g., the F-logic atom $X[M@(A_1, \ldots, A_k) \twoheadrightarrow \{Y\}]$ corresponds to an edge from object X to Y, which is labeled with the parameterized method $M@(A_1, \ldots, A_k)$. The *ssd* model is a special case of this graphical representation:

Nodes of a *ssdb* $\mathcal{D}$ correspond to oids, whereas labeled edges correspond to multi-valued method applications. More precisely, for a labeled edge in $\mathcal{D}$, an equivalent graph notation and a representation in F-logic syntax can be given as follows:

$$(x, \ell, y) \in \mathcal{D} \qquad \Leftrightarrow \qquad x \overset{\ell}{\twoheadrightarrow} y \qquad \Leftrightarrow \qquad x[\ell \twoheadrightarrow \{y\}] \ .$$

Thus, for a given *ssdb* $\mathcal{D}$, we obtain the following natural representation in F-logic:

```
X:node, Y:node, L:label, X[L↠{Y}] ← ssdb(X,L,Y).
```

Here, it is essential that L is viewed as a *multi-valued* method, since there may be several distinct edges emanating from x which share the same label ℓ.

Web Skeleton. Clearly, the link structure of a set of Web document may be conceived as a *ssdb*: nodes correspond to Web documents and edges to labeled hyperlinks between documents. If nodes are opaque, i.e., when no information apart from the link structure is available, we speak of the *(Web) skeleton* of a set of Web documents. For example, the labeled graph depicted in Figure 2 represents a fragment of the skeleton of the DBLP server [DBL]: In the skeleton view, the only information available is contained in labels (represented as strings), whereas nodes are opaque. Thus, the *skeleton* covers the *structural* aspect of Web documents but not their *contents*. We will deal with extracting contents in Section 4.

[2] Sometimes also called *graph database* [BDFS97].

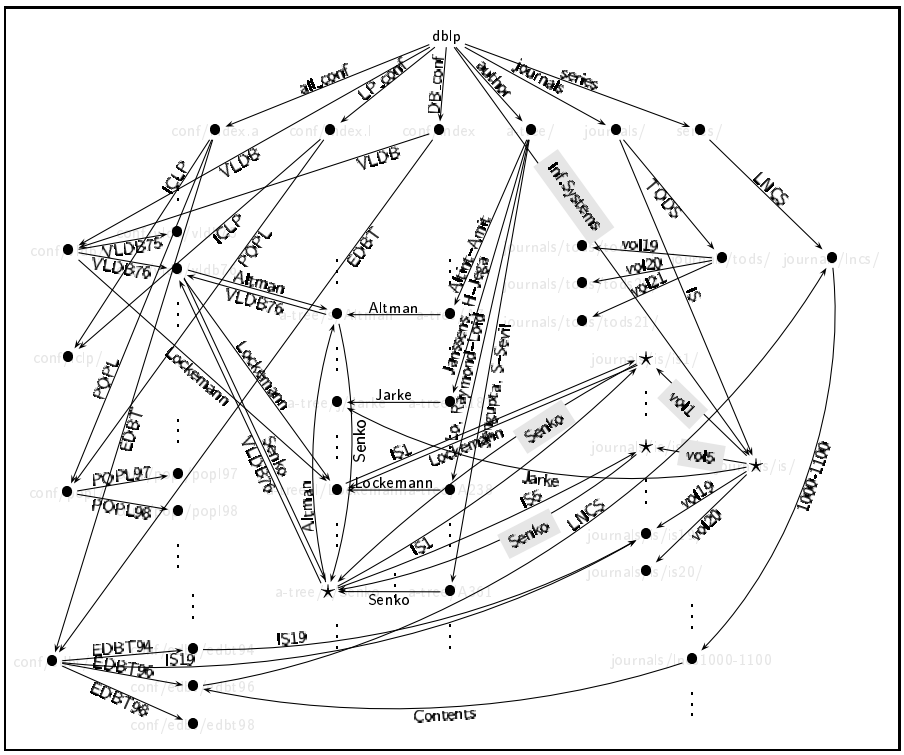

Figure 2: Fragment of the skeleton of the DBLP database

3.2 A Generic Web Skeleton Extractor

A Web skeleton like the one depicted in Fig. 2 can be automatically extracted with FLORID using the following generic skeleton extractor P_{ext}:

P_{ext}:

```
root[src↠{u_1, ... , u_n}].                                  % (1)
node :: url.                                                 % (2)
U : node.get ← root[src↠{U}].                                % (3)
---------------------------------------------------------------
Y : node, L : label, X[L↠{Y}] ←                              % (4)
        X : node.get[hrefs@(L)↠{Y}], φ.
Y.get ← Y : node, ψ.                                         % (5)
```

Figure 3: A generic skeleton extractor for FLORID

First, the relevant source url's are defined (1), and the class node is declared a subclass of url (2). Every source url u is made an instance of node (and thus of url), and the single-valued method get is defined for u (3): u is accessed, u.get is assigned the respective Web document and some additional methods for u.get are defined, among them hrefs@(...). Then, an exploration cycle is started: Given an url (node) x, in rule (4), for all labels ℓ and referenced url's y s.t. x.get[hrefs@(ℓ)↠{y}], the labeled edge $x \overset{\ell}{\twoheadrightarrow} y$ is added to the F-logic *ssdb* and y is made a node. By constraining (4) with an additional goal φ, only those labeled edges are defined which are considered relevant. Finally, (5) fetches the new Web document y.get of

the node y, provided the condition ψ holds. Thus, φ is a first constraint limiting the number of strings which are considered as nodes (and thus url's). Additionally, only those url's y are actually accessed and their contents retrieved in y.get, for which ψ holds.

The rule pattern given by P_{ext} allows straightforward extraction of a Web skeleton, simply by instantiating the source url's, and φ and ψ appropriately:

Example 1 (DBLP Skeleton Extractor) A fragment of the DBLP server skeleton (Fig. 2) is retrieved when starting the skeleton extractor with

```
root[src↠{dblp}].        dblp = "http://www.informatik.uni-trier.de/~ley/db/".
```

and the constraints

- φ = substr("trier",Y), and (*consider only url's containing "trier"*)
- ψ = substr("/db/journals/is/",Y). (*restrict to is (=Information Systems) journal*)

After all relevant documents have been accessed, we may query the skeleton: For example, we may be interested in all authors whose name contains a certain substring (say: "sen"), and who had a paper in *Information Systems*. Using the extracted hyperlink structure of the DBLP skeleton (cf. Fig. 2), we can directly issue the following query:

```
?– dblp.."Inf. Systems"..V..A, substr("sen",A).
  A/"Christian S. Jensen" V/"Volume 21, 1996"
  A/"Christian S. Jensen" V/"Volume 19, 1994"
  A/"Arun Sen" V/"Volume 20, 1995"
  A/"Arun Sen" V/"Volume 11, 1986"
  A/"Georg Lausen" V/"Volume 8, 1983"
    ⋮
  A/"Michael E. Senko" V/"Volume 1, 1975"
16 output(s) printed
```

As this query reveals, the links emanating from the *Information Systems* page are (typically) volume pages; the links emanating from volume pages are (typically) author pages. In this way, the link structure of Web sites can be explored and analysed using FLORID's powerful ad-hoc queries. □

Simply by instantiating P_{ext} differently, we obtain the skeleton of the weather encyclopedia of the *Schweizer Fernsehen (SF DRS)*:

Example 2 (Meteo Skeleton Extractor) We start P_{ext} as follows:

```
root[src↠{meteo}].       meteo = "http://www.sfdrs.ch/sendungen/meteo/lexikon/index.html".
```

and use the identical constraints

- φ, ψ = substr("/lexikon/",Y). (*restrict to "lexikon" pages*)

In order to find the names of all entries in the weather encyclopedia, we ask:[3]

[3] FLORID has different output modes for displaying answers: In Example 1 this mode is set to *variable bindings* (as in Prolog), while it is set to *ground instances* here (and in other examples below).

```
?– meteo..L.
   meteo.."absolute Feuchte".
   meteo.."Absorption".
      ⋮
   meteo.."Weiterführende Literatur".
131 output(s) printed
```
□

It should be clear that, apart from the above substr-predicate, there are many other ways of limiting the set of explored url's in P_{ext}. For example, one may define a method depth for each node, such that source nodes have depth zero and a document referenced from another one with depth n has itself depth at most $n+1$. Based on this, one can easily restrict Web exploration to documents with depth $\leq n_0$.

3.3 Mining Links

Once a Web skeleton has been extracted, FLORID's deductive query language allows to gather new information from the given data sources. We illustrate some features based on the weather encyclopedia program from Example 2. Operating on the skeleton of such an encyclopedia is particularly interesting, since its hyperlink structure mirrors—to a large extent—semantic relationships between the corresponding notions of the encyclopedia. For example, one may discover related notions using path expressions, or one may estimate the importance of certain notions based on the frequency of their use (which requires the use of aggregation):

Path Expressions. Take some entry from the weather encyclopedia, say "Ozonschicht" (*ozone layer*). Then, the query

```
?– meteo..L.."Ozonschicht"..M.
```

finds all entries L and M "in the context" of this entry, i.e., starting from the root page meteo, there is a link labeled L leading to the "Ozonschicht"-page, and from this page there is a link labeled M:

```
?– meteo..L.."Ozonschicht"..M.
   meteo.."Absorption".."Ozonschicht".."Atmosphäre".
   meteo.."FCKW".."Ozonschicht".."Atmosphäre".
      ⋮
   meteo."O 3"."Ozonschicht"."FCKW".
   meteo."UV-Strahlung"."Ozonschicht"."absorbiert".
49 output(s) printed
```

If we were only interested in "back-loops", i.e., labels which occur before *and* after the "Ozonschicht"-page, we could require that L=M, resulting in only four answers for L.[4]

General Path Expressions. Apart from the (simple) path expressions considered so far, so-called *general path expressions* have been suggested for querying and navigating on semistructured data (see e.g. [AQM+97]). Given labels L and M, these expressions allow to follow "generalized" labels like $(L{\cdot}M)$ (sequence), $(L|M)$ (disjunction), $(L)^*$ (iteration), $(L)^{-1}$ (inversion), etc. In [LHL+98] it is shown how such general path expressions can be evaluated with FLORID.

[4] "FCKW", "Ozon", "Stratosphäre", and "UV-Strahlung"

Aggregation. To find the "most important" entries, we can use aggregation and count the *fan-in*, *fan-out*, and *edge-count* of nodes and labels, respectively. This is accomplished by adding the following rules to the Meteo skeleton extractor:

```
e(X,L,Y)[] ← X:node[(L:label)↠{Y:node}].          % define all "edge-objects"
X[fan_out→N] ← N=count{Y [X]; e(X,_,Y)[]}.   % aggregation: count outgoing...
Y[fan_in→N] ← N=count{X [Y]; e(X,_,Y)[]}.          % ... and incoming edges
L[edge_count→N] ← N=count{Z [L]; Z=e(X,L,Y)[]}.     % count edges per label
```

The first rule defines, for each labeled edge $x \stackrel{\ell}{\twoheadrightarrow} y$, a distinct object (note the use of "[]" to distinguish the *object* e(X,L,Y)[] with empty specification from the ternary *predicate* e(X,L,Y)). The second and third rule use the built-in aggregation count: the expressions with curly braces define the aggregation: for example, "count{Y [X] ; ...} " means *"count all Y's, group by X"*, the expression behind the semicolon ";" is the aggregation goal.

In our example, when querying the edge-count, we find that "Luft" and "Temperatur" are the "most important" entries:

```
?– N = L.edge_count.
  1 = "absolute Feuchte".edge_count.
    ⋮
 35 = "Temperatur".edge_count.
 41 = "Luft".edge_count.
252 output(s) printed
```

Graph Algorithms. Since the Web skeleton is a directed labeled graph, general graph algorithms can be used to reveal interesting structural properties of the skeleton. For example, we may be interested in the *strongly connected components* (*scc*) of the given skeleton: two nodes x and y belong to the same *scc* iff they are reachable from each other in the given graph. To this end, we simply add the generic program for computing *scc*'s on *ssdb*'s (Figure 4) to the program above:

```
tc(X,Y) ← X:node[(_:label)↠{Y}].                  % on the given labeled graph...
tc(X,Y) ← tc(X,Z), tc(Z,Y).                  % ... compute the transitive closure
X:scc_id(X) ← X:node.                  % initially, each node belongs to his own scc
scc_id(X) = scc_id(Y) ← tc(X,Y), tc(Y,X).   % ...but scc's may be fused if mutually reachable!
?– sys.strat.dolt.                          % sys-command to enforce stratification
Z[size→N] ← N = count{X [Z] ; X:Z, Z = scc_id(C)}.     % determine the size of each scc
```

Figure 4: P_{scc}: Computing strongly connected components in FLORID

The program P_{scc} makes essential use of *derived equalities*—a special feature of F-logic (and FLORID). Also, note the user-defined stratification (?-sys.strat.dolt.) before the last rule: In order to ensure that the final aggregation produces the intended results, the rules above the aggregation must have been evaluated completely beforehand.
Given the rules above, we can now determine the number of *scc*'s and their sizes:[5]

[5]Here we have switched back to FLORID's *variable bindings* output mode.

```
?- M = count{Z [N] ; Z[size →N]}.
   M/1    N/96
   M/25    N/1
2 output(s) printed
```

Thus, there are 26 *scc*'s: 1 large *scc* with 96 nodes and 25 trivial *scc*'s with a single node. Observe that the argument U of the oid scc_id(U) is an instance of node and hence of url.[6] Therefore, we may apply the method get to it. In this way, we can extract the *titles* of pages in the large *scc* as follows:

```
?- scc_id(_U)[size→96], _U.get[title→T].
   T/"Wetterlexikon: Index"
   T/"Wetterlexikon: Absolute Feuchte"
      ⋮
   T/"Wetterlexikon: Wolken"
96 output(s) printed
```

4 Querying Contents

Apart from extracting the skeleton of a set of Web documents (i.e., their *link structure*), also their *contents* may be queried. To this end, built-in predicates for extracting and analyzing data from accessed Web documents have to be provided. A simple, yet flexible and powerful approach used in FLORID, is to view Web documents as (large) strings and then apply *regular expressions* to extract data. Regular expression can be used, for example, to extract all strings between pairs of HTML tags like <h2> and </h2> (level-2 headings), or to analyze tables or lists.[7] The regular expressions employed in FLORID include groups and format strings, thereby providing a very expressive language: The predicate

pmatch(*Str*, *RegEx*, *Fmt*, *Res*)

finds all strings in the input string *Str* which match the pattern given by the (Perl) regular expression *RegEx*. The *format string Fmt* describes how the matched strings should be returned in *Res*. This feature is particularly useful when using *groups* (expressions enclosed in (...)) in regular expressions. For example, we have:[8]

```
?- pmatch("A man's only as old as he feels", "/(.*) (he .*)/", "$1 the woman $2", X).
   X/"A man's only as old as the woman he feels"
1 output(s) printed
```

Since for an url u, the reference u.get denotes the fetched Web document, u.get can be used as first argument to the pmatch predicate.

4.1 Syntactical Queries: CIA World Factbook

We illustrate simple content-based queries using the pages of the *CIA World Factbook*, a collection of Web pages providing information about the countries in the World (e.g., on geography, people, government, and economy). Although one may argue that the data in the World Factbook is highly regular and should be put on the Web as a database in the

[6] See (2) in Figure 3.

[7] Another possibility currently being incorporated is to use a general SGML parser, whose output is directly mapped into some F-logic structure.

[8] The answer for X is a quote from Groucho Marx, see `http://www.bmacleod.com/groucho.html`

first place, there are some idiosyncrasies and irregularities to consider (see below for a simple example, i.e., the "pseudo-values" of the attribute capital). Moreover, the actual database is *not* available from the Web, whereas the semistructured HTML pages are.

Note that the following rules make up a *self-contained* program, i.e., there are no separate languages for wrappers or mediators. The program is roughly organized as follows: After a certain root page has been accessed, several outgoing links to relevant country pages are followed and the corresponding country pages are accessed. Thus, data-driven Web exploration (Section 2.2) is used in conjunction with querying structure (navigation along hyperlinks; Section 3). Finally, the actual data is extracted from the country pages, which corresponds to querying contents:[9]

First, the urls of the World Factbook homepage, of the page for countries in Europe, and of a local mirror are defined for the object cia, our "root" object for the Factbook:

```
cia[world →"http://www.odci.gov/cia/publications/nsolo/factbook/global.htm" ;
    europe →"http://www.odci.gov/cia/publications/nsolo/factbook/eur.htm" ;
    mirror →"file:/home/dbis/flogic/data/ciawfb/global.htm" ].
```

To allow for easy substitution of the data source, a generic name cia.src is defined:

```
cia[src →cia.world].      % use the main Factbook page here
```

Alternatively, cia.src may be set to cia.europe, or may even be rule-defined: e.g., if access to cia.world fails, cia.src can be defined as cia.mirror.

The string represented by cia.src is made an instance of class url and accessed via get:

```
cia.src:url.get.          % ⇔ (cia.src):url ∧ (cia.src).get
```

Thus, cia.src.get is the name (logical oid) of the accessed Web document and hrefs@(*label*) is defined for it by the system (unless an error has occurred). The source page cia.src.get of the Factbook contains links to the individual country pages. These links are used to populate the class country with instances C and the urls U of C:

```
C:country[url →U] ←                          % remember the url U of country C after ...
    cia.src.get[hrefs@(Lbl) →→{U}],          % ... extracting all labels Lbl and urls U ...
    pmatch(Lbl, "/(.* ) \([0-9]/","$1", C).  % ... and removing excess parts from Lbl
```

The labels Lbl in cia.src.get are the names of the countries followed by the size of the page in KBytes (e.g., "Spain (32 KB)"). Here, the built-in predicate pmatch is used to strip off this size information: e.g., match("Spain (32 KB)", ..., ..., C) yields C="Spain".

The individual pages of the extracted countries are accessed by defining get for the corresponding urls as usual:

```
U.get ← C:country[url→U].      % retrieve all country pages
```

Observe that the name url can be used for both, the predefined class of urls, and the user-defined method url: In F-logic, also methods and classes are objects; thus, it is possible to reason about schema information.

Finally, data from the country pages can be extracted and stored in the F-logic database. For example, among lots of other data, the following can be extracted (recall that the pmatch predicate treats Web documents as strings):

[9] For a more detailed example in which also data from different sources is integrated, see [HKL+98].

```
C[capital →X]       ←  pmatch(C:country.url.get, "/Capital:.*\n(.*)/", "$1", X).
C[total_area →X]    ←  pmatch(C:country.url.get, "/total area:.*\n(.*sq km)/", "$1", X).
C[external_debt →X] ←  pmatch(C:country.url.get, "/External debt:.*\n(.*)/", "$1", X).
```

These rules show a strong regularity. Thus, one can take advantage of the meta-programming facilities of F-logic (here: variables at method position) and replace the code by *a single generic rule* and facts describing the used patterns:

```
C[Method→X] ← pattern(Method, RegEx), pmatch(C:country.url.get, RegEx, "$1", X).
pattern(capital,"/Capital:.*\n(.*)/").
pattern(total_area,"/total area:.*\n(.*sq km)/").
pattern(external_debt,"/External debt:.*\n(.*)/").
```

Clearly, such a "pattern-base" may be extended easily for other methods.

Querying the Data. Once the data has been extracted, it can be queried, restructured, and integrated with data from other sources, using all features of F-logic and FLORID, respectively:

```
?- N = count{C ; C:country}.     % (Q1) use aggregation to count the countries
?- C:country[capital→CA].        % (Q2) name all countries and their capitals!
?- C:country, not C.capital.     % (Q3) which countries do not have a capital?
```

For cia.src=cia.world, query (Q1) yields N=266 countries. However, (Q2) outputs a binary relation (Country,Capital) with only 256 entries. (Q3) reveals the 10 "countries" for which the method capital is not defined, e.g., "Antarctica", "Atlantic Ocean", and "World". It turns out that there are some more "countries" which have the method capital defined, yet do *not* have a proper capital. For example, the fact

```
"Bouvet Island":country[capital→"none; administered from Oslo, Norway"]
```

can be derived by FLORID. Thus, we may specify the class of *real* countries as follows:

```
C:real_country ← C:country[capital→CA], not substr("none", CA).
```

Now, the query ?- C:country, not C:real_country discloses 25 more "false countries" (apart from the 10 above) including, e.g., "Bouvet Island", "Clipperton Island", and "Western Sahara".

Schema Browsing and Discovering Structure. Since method and class names are first-class citizens in F-logic, reasoning about schema information is possible. Consider the following queries:

```
?- _:country[M→_].               % (Q4) what methods are defined for countries?
?- _:country.M, C:country, not C.M.   % (Q5) return countries with undefined methods
```

Query (Q4) yields all single-valued methods (capital, total_area, land_area, etc.) potentially defined for countries (i.e., defined for at least *some* country). The different occurrences of the anonymous (don't care) variable "_" denote *distinct* ∃*-quantified* variables. The first literal _:country.M of (Q5) is a syntactic variant of (Q4); together with the rest of (Q5), "countries" C with undefined methods M are reported: e.g., for C="Ashmore and Cartier Islands", the method M="labor_force" is undefined.

Looking at the CIA World Factbook pages, we discover that important attributes of countries are printed in **boldface**. Hence, we can automatically reveal potentially relevant attributes of countries by extracting all data between ⟨B⟩...⟨/B⟩. Unfortunately, this yields many irrelevant answers. These can be eliminated by *intersecting* the boldface expressions over all country pages. Logically, we use double negation to find the notions which are present on *all* country pages:[10]

```
% for all real countries, extract all bold expressions:
C[bold_ex→↠{R}] ← pmatch(C:real_country.url.get, "m!⟨B⟩(.*?)⟨/B⟩!g", "$1", R ).

% what bold expressions are not defined for all real countries?
not_all_bold(M) ← _:real_country[bold_ex→↠{M}], C:real_country, not C[bold_ex→↠{M}].

% what bold expressions are defined for all real countries?
all_bold(M) ← _:real_country[bold_ex→↠{M}], not not_all_bold(M).
```

After evaluating the above program, we obtain the desired answers:

```
?– all_bold(M).
  M/"Location:"
  M/"Description:"
  M/"Area:"
    ⋮
75 output(s) printed
```

4.2 Querying Semantic Tags: A FLORID-XML Parser

With the current state of the art, extracting data from "ordinary" HTML pages requires the often quite tedious and time-consuming task of writing an appropriate wrapper. In particular, this is true when the data source offers only syntactic hints for the *presentation* of the data (e.g., format tags: boldface, italics, etc.) and no information about the *meaning* and/or *context* of data.[11] Thus, a better approach for supporting information gathering from the Web is the use of *semantically meaningful tags*. For example, the *Extensible Markup Language XML* is an effort within W3C to support structured document interchange on the Web [XML97]. XML allows the definition of customized markup languages with application-specific tags. The data model for XML is very simple and corresponds to a tree-like structure; XML documents are (quite verbose) linerizations of this data structure. Since in a *well-formed* XML document

- all tags must be balanced (elements must have both start and end tags present), and
- elements must nest inside each other properly (no overlapping markup),

XML documents have a highly regular structure and can be parsed and analysed very easily.

Consider, for example, the XML representation of a relational database [XML97]: A relational database consists of a set of tables, where each table is a set of records. A record in turn is a set of fields and each field is a pair field-name/field-value. This description of the database suggests a simple nesting of fields inside records inside tables inside databases:

[10] Here, we tacitly assume a stratification (?-sys.strat.dolt.) after each rule.

[11] Nevertheless, as can be seen from the Section 4.1, simple "wrappers" may still be easily specified by FLORID rules (provided the Web source shows enough regularity).

Fig. 5 is an example of a single database in XML with two tables authors and editors.[12] Every element (i.e., expression of the form ⟨*tag ...*⟩*...*⟨*/tag*⟩) induces a *box*. Based on this box model, a simple XML-parser can be defined very elegantly and concisely in FLORID:

A Simple FLORID-XML Parser. Similar to the access of entry pages of the skeleton extractor (Fig. 3), we first get the root page(s):

```
root[src↠{"http://www.informatik.uni-freiburg.de/~dbis/florid/xml_sample}].
U:url.get ← root[src↠{U}].
```

Once the pages have been accessed, we may query their contents: First, we extract all tags from the documents and populate the class tag with this data:

```
T:tag ← pmatch(_.get,"m!⟨/(\w+)⟩!g","$1",T).
```

Next, we define for every found tag T, the (Perl) regular expression for matching strings of the form ⟨T⟩⟨*data*⟩⟨/T⟩:[13]

```
T[regex→P] ← T:tag, pmatch(T,"/(.*)/","m!⟨$1⟩ (.*?)⟨/$1⟩!gis",P).
```

For every url U whose Web document U.get matches the regular expression T.regex for some tag T, we create a "box" B containing the matched data, and link this box via the method T to the original url U:

```
U[T↠{B:box}] ← T:tag, pmatch(U.get,T.regex, "$1", B).
```

The actual crux of the parser is the *recursive "dissection"* of boxes into sub-boxes and their interlinking by the following rule: For every box B and every tag T, if B contains data matching T.regex, then B is a *complex box* (cbox), and the newly found data NewB is a box, which is accessible from B via T:

```
B:cbox[T↠{NewB:box}] ← B:box, T:tag, pmatch(B,T.regex, "$1", NewB).
```

Finally, provided these rules have been evaluated, we can determine the class of *atomic boxes* (abox), i.e., containing no further sub-boxes:

```
B:abox ← B:box, not B:cbox.
```

The presented simple parser extracts the XML data and maps it into an F-logic database. Using this representation, complex queries can be expressed in a clear and intuitive way with FLORID: For example, for the XML document in Fig. 5 we may be interested in all atomic boxes and the names of their "parent boxes":

```
?- _..X = B:abox.
  _.."address" = "10 Tenth St, Decapolis":abox.
  _.."address" = "2 Second Av, Duo-Duo":abox.
  _.."address" = "1 Premier, Maintown":abox.
  _.."name" = "Robert Roberts":abox.
    ⋮
  _.."born" = "1960/05/26":abox.
  _.."telephone" = "7356":abox.
13 output(s) printed
```

[12] In the actual text document, the author-entries are arranged vertically instead of horizontally.
[13] Here, for simplicity, we do not deal with attributes inside of tags.

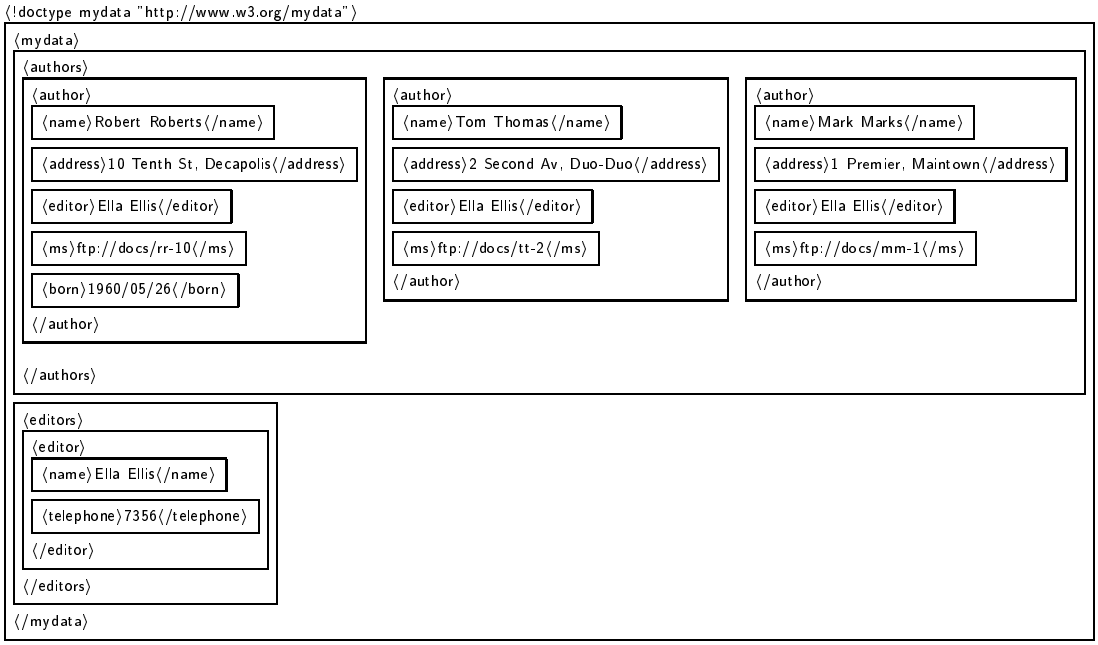

Figure 5: XML representation of a relational database [XML97]

Or we may ask: *"What tags can be found inside of the* authors' *box?"*

```
?– _.."authors"..X.
   _.."authors"."name".
   _.."authors"."address".
   _.."authors"."editor".
   _.."authors"."ms".
   _.."authors"."born".
   _.."authors"."author".
6 output(s) printed
```

Since the above program links direct *and indirect* sub-boxes to the box containing the sub-boxes, also transitive links are present: For example, we can ask for authors' names as follows:

```
?– _.."authors"."name" = X.
   _.."authors"."name" = "Robert Roberts".
   _.."authors"."name" = "Tom Thomas".
   _.."authors"."name" = "Mark Marks".
3 output(s) printed
```

5 Conclusion

We have shown, by means of several illustrative examples, how Web data can be queried in an intuitive and declarative way using FLORID: A generic skeleton extractor has been presented, which allows to automatically extract the hyperlink structure of collections of Web documents. Based on FLORID's logical query language, this structure may be further analysed, e.g., using (general) path expressions and aggregation. In addition to structure-based queries, FLORID also supports content-based queries: In the current implementation,

(Perl) regular expressions are used to work on poorly structured data (e.g., plain text), or to operate on highly structured data (like XML). A general built-in SGML parser is going to be incorporated in the near future and will map SGML documents to F-logic databases.

The extension of FLORID's semantics for querying the Web is described in [HLLS97]; the papers [HKL+98] and [LHL+98] focus on integration of different sources and management of semistructured data with FLORID, respectively. The latter contains also a short introduction to F-logic and path expressions.

References

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 1–18. Springer, 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997.

[BDFS97] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In F. Afrati and P. Kolaitis, editors, *6th Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 336–350, Delphi, Greece, 1997. Springer.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrandt, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1996.

[BRR97] F. Bry, K. Ramamohanarao, and R. Ramakrishnan, editors. *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, Montreux, Switzerland, 1997. Springer.

[Bun97] P. Buneman. Semistructured Data (invited tutorial). In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 117–121, Tucson, Arizona, 1997.

[DBL] DBLP Computer Science Bibliography. `http://www.informatik.uni-trier.de/~ley/db/`.

[FLO] The FLORID Home Page. `http://www.informatik.uni-freiburg.de/~dbis/florid/`.

[GMNP97] F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Datalog++: A Basis for Active Object-Oriented Databases. In Bry et al. [BRR97].

[HKL+98] R. Himmeröder, P.-T. Kandzia, B. Ludäscher, W. May, and G. Lausen. Search, Analysis, and Integration of Web Documents: A Case Study with FLORID. In *Proc. Intl. Workshop on Deductive Databases and Logic Programming (DDLP'98)*, pp. 47–57, Manchester, UK, 1998. GMD Report 22.

[HLLS97] R. Himmeröder, G. Lausen, B. Ludäscher, and C. Schlepphorst. On a Declarative Semantics for Web Queries. In Bry et al. [BRR97], pp. 386–398.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with *FLORID*: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8), 1998. to appear.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *Proc. Sixth International Workshop on Research Issues in Data Engineering (RIDE)*, 1996.

[MV98] U. Masermann and G. Vossen. Suchmaschinen und Anfragen im World Wide Web. *Informatik Spektrum*, 21(1):9–15, 1998.

[Suc97] D. Suciu, editor. *Proc. of the Workshop on Management of Semi-Structured Data (in conjunction with SIGMOD/PODS)*, Tucson, Arizona, 1997. `http://www.research.att.com/~suciu/workshop-papers.html`.

[XML97] Extensible Markup Language (XML). `http://www.w3.org/XML/`, 1997.