HÖGSKOLAN SKÖVDE

Institutionen för kommunikation och information
Examensarbete i datavetenskap 25p
D-nivå
Vårterminen 2005

Using a Rule-System as Mediator for Heterogeneous Databases, exemplified in a Bioinformatics Use Case

Anna Schroiff (a04annsc@student.his.se)

HS-IKI-MD-05-003

Using a Rule-System as Mediator for Heterogeneous Databases, exemplified in a Bioinformatics Use Case

Submitted by Anna Schroiff to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics.

2005–09–01

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signed:_____

Using a Rule-System as Mediator for Heterogeneous Databases, exemplified in a Bioinformatics Use Case

Anna Schroiff

Abstract

Databases nowadays used in all kinds of application areas often differ greatly in a number of properties. These varieties add complexity to the handling of databases, especially when two or more different databases are dependent.

The approach described here to propagate updates in an application scenario with heterogeneous, dependent databases is the use of a rule-based mediator. The system EruS (ECA rules updating SCOP) applies active database technologies in a bioinformatics scenario. Reactive behaviour based on rules is used for databases holding protein structures.

The inherent heterogeneities of the Structural Classification of Proteins (SCOP) database and the Protein Data Bank (PDB) cause inconsistencies in the SCOP data derived from PDB. This complicates research on protein structures.

EruS solves this problem by establishing rule-based interaction between the two databases. The system is built on the rule engine ruleCore^{TM} with Event-Condition-Action rules to process PDB updates. It is complemented with wrappers accessing the databases to generate the events, which are executed as actions. The resulting system processes deletes and modifications of existing PDB entries and updates SCOP flatfiles with the relevant information. This is the first step in the development of EruS, which is to be extended in future work.

The project improves bioinformatics research by providing easy access to up-to-date information from PDB to SCOP users. The system can also be considered as a model for rule-based mediators in other application areas.

Keywords: ECA rules, SCOP, PDB, databases, update propagation

Contents

1 Introduction **1**
- 1.1 Starting Point: Current Situation . . . 1
- 1.2 Limitations of the Current Situation . . . 2
- 1.3 Potential Solution . . . 3
- 1.4 Aims and Objectives . . . 3
  - 1.4.1 Objectives . . . 3
  - 1.4.2 Structure . . . 4

2 Background **5**
- 2.1 The Protein Data Bank - PDB . . . 5
  - 2.1.1 Data . . . 5
  - 2.1.2 Architecture . . . 5
  - 2.1.3 Query Mechanism . . . 6
  - 2.1.4 Data distribution and update policies . . . 6
- 2.2 The Structural Classification of Proteins Database - SCOP . . . 6
  - 2.2.1 Data . . . 7
  - 2.2.2 Architecture . . . 7
  - 2.2.3 Query Mechanism . . . 7
  - 2.2.4 Data distribution and update policies . . . 8
- 2.3 Related Work . . . 8

3 Problem Description **9**
- 3.1 Use Cases . . . 9
- 3.2 Heterogeneities of Database Systems . . . 10
- 3.3 Alternative Solutions . . . 11
- 3.4 Limitations . . . 12

4 The Rule System **13**
- 4.1 ECA Rules . . . 13
- 4.2 Requirements . . . 13
- 4.3 ruleCore™ . . . 14
  - 4.3.1 Markup Language . . . 14
  - 4.3.2 Architecture . . . 14
  - 4.3.3 System Requirements . . . 15

5 The EruS Prototype **16**
- 5.1 PDB Event Connector . . . 16
- 5.2 ECA Rules . . . 17
- 5.3 SCOP Wrappers . . . 20
  - 5.3.1 Modified Entry in PDB . . . 21
  - 5.3.2 Obsolete Entry in PDB . . . 21
  - 5.3.3 Obsolete Entry in SCOP . . . 21
  - 5.3.4 Lookup Comment . . . 22

6 Evaluation **23**
6.1 Test Cases . . . 23
6.2 Results . . . 23
6.3 Discussion . . . 24
6.3.1 Use Cases . . . 24
6.3.2 Extendibility and Maintainability . . . 25
6.3.3 Limitations . . . 25
6.3.4 Heterogeneities . . . 25

7 Conclusion **27**
7.1 Contribution . . . 27
7.2 Future Work . . . 28

8 References **29**

A PDB Event Connector **32**

B ECA Rules in `rcml` Format **44**

C SCOP Wrappers **49**
C.1 Modified Entry in PDB . . . 49
C.2 Obsolete Entry in PDB . . . 51
C.3 Obsolete Entry in SCOP . . . 55
C.4 Lookup Comment . . . 58

1 Introduction

Information is nowadays often stored and managed by the use of database systems, in science as well as in industry. A wide spectrum of different databases is in use for this in a wide range of application areas. The database systems can differ greatly in a number of properties, for example purpose, underlying data models, supported output formats, or provided query mechanisms (Date 2003). These varieties add complexity to the handling of databases, especially when two or more different databases are used.

An example application area where this kind of situation occurs is bioinformatics. This is due to the fact that bioinformatics depends highly on the use of huge amounts of data, that is stored in different kinds of databases with a "variety of semantics, interfaces, and data formats used" (Lacroix & Critchlow 2003). Each January the journal "Nucleic Acids Research" publishes a list of molecular biology databases. In its 2005 update (Galperin 2005) the collection comprised more than 700 publicly available databases with heterogeneous properties and focusing on different information.

In practice many researchers store local copies of databases regularly needed on their computers (Alfares et al. 2005) in order to be independent from access to the remote sources and to comfortably use the data any time at any place. When performing a task, they need to obtain and integrate data from different databases, potentialy both locally and remotely maintained (Chung & Wooley 2003). Datasets acquired from distinct databases can include dependencies and connections as data stored in so called secondary databases is derived from data stored in primary databases.

1.1 Starting Point: Current Situation

The use of bioinformatics databases in practice can be illustrated by the following use case, which has been identified before by Alfares et al. (2005). It regards a common bioinformatics scenario consisting of the two databases PDB (Protein Data Bank) and SCOP (Structural Classification of Proteins). Figure 1 shows the current situation which is the starting point of this project.

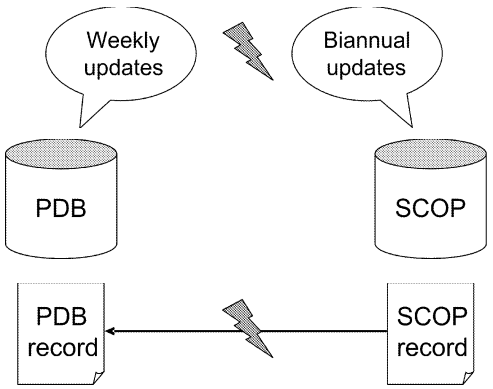

Figure 1: The starting point of this project

Protein Data Bank: PDB contains data about three dimensional structures of proteins. As Berman et al. (2000) state, it is the "single worldwide

archive of structural data of biological macromolecules". The database is updated weekly, including the addition of new datasets with new protein structures (RCSB 2004*a*).

Structural Classification of Proteins: SCOP consists of a hierarchy of protein structures which represents evolutionary relationships between structural domains of proteins (i.e., elements of protein structures). The current version of SCOP[1] has been released in July 2005. It is based on 25973 PDB entries and one literature reference; these are all proteins of known structure as of October 2004. Before that, the last update was released in February 2005 because the complex and mostly manually performed update procedure takes a couple of months, so that new releases are usually published biannually. SCOP provides parseable files for download which are used locally for large scale analysis as opposed to using the original remote version of SCOP or one of its mirrors.

As explained above, datasets in SCOP are derived from datasets in PDB. Experimental protein structures stored in PDB are classified in SCOP into a hierarchy representing evolutionary relationships between protein domains (details in section 2.2). Researchers use SCOP data for large scale analysis of protein domains to find evolutionary related proteins which are possibly used for further investigation (Murzin et al. 1995). SCOP data is used in the form of parseable flat files for different examinations and studies (Brenner et al. 1996, Selvam & Sasidharan 2004, Liu et al. 2004, Harlow et al. 2004, Bostick et al. 2004).

1.2 Limitations of the Current Situation

The current use of bioinformatics databases as described in the scenario above (section 1.1) comprises limitations and problems concerning the use and maintenance of the data sources.

First of all, the manual update handling of local copies as well as the original version of SCOP is not only inconvenient and time consuming, it also makes research error-prone since the actuality of the data used cannot be guaranteed.

Another problem that needs to be tackled lies in the inconsistencies arising from different update rates in PDB and SCOP: PDB is updated weekly whereas SCOP is updated only every six months. This fact leads to inconsistencies in SCOP as for instance references may have been updated or deleted in PDB since the last SCOP update. In case of the deletion of a PDB entry, the SCOP entry is connected to a non-existent PDB entry; in case of an update the classification based on structure information from PDB may no longer be correct.

The fact that even data in the remote, original version of SCOP cannot be guaranteed to be up-to-date with current research already integrated in PDB, leads to mistakes and slows down progress in research.

Research in the described field is thus inevitable to enhance and simplify the use of existing data as well as the capturing and classification of new data in bioinformatics in order to optimise research and yield better founded results in a shorter time. The communication of conducted analysis within the bioinformatics community can be improved and thus aims can be pursued more efficiently.

[1] http://scop.mrc-lmb.cam.ac.uk/scop/count.html#scop-1.69 (accessed 12 August 2005)

1.3 Potential Solution

The limitations and problems described above can be tackled by automating the synchronisation and update process between the data sources. As the systems in use can not be changed easily, an additional system is needed to act as a mediator between the existing systems.

The task to be performed in the given scenario is the handling of events (updates, deletes, inserts in PDB) by performing an appropriate action in a local copy of SCOP. This behaviour can be established by a rule system containing event-condition-action (ECA) rules of the form "on EVENT if CONDITION do ACTION". The rule system to be developed, "EruS - ECA rules updating SCOP", can achieve a more efficient interaction between the databases by replacing manual handling with an automatic mechanism.

The interaction accomplished by this can solve consistency and actuality issues by updating the data immediately. It makes the maintenance of local database copies more convenient and these data sources more reliable. Errors in research caused by out-of-date or incomplete data can be avoided and progress in bioinformatics research can be accelerated.

1.4 Aims and Objectives

The aim of the project is to develop a set of ECA rules for a rule system serving as mediator between two heterogeneous databases, exemplified in the case of two bioinformatics databases. The starting point is the scenario depicted above (section 1.1) consisting of the connected databases PDB and SCOP.

1.4.1 Objectives

The aim can be broken down in several objectives that need to be reached.

Specification of Use Cases: The first objective is to narrow down the limitations and problems to use cases to be addressed in this project. This objective thus defines the scope of the project and the requirements to be met.

Introduction of the Rule System: An existing rule system is used as a basis for implementing the ECA rules. This rule system is introduced and its functionality and use explored in this step.

Development of ECA Rules for the EruS Prototype: The development of the EruS prototype consisting of the rule engine, a set of ECA rules, and wrapper scripts follows. The ECA rules that handle the use cases identified before are defined according to the format demanded by the rule engine.

Evaluation and Testing of the EruS Prototype: Testing of the prototype based on test cases verifies the practical feasibility. EruS is set in an application environment to show that it works for the given use cases.

1.4.2 Structure

The objectives provide the structure of this thesis. Chapter 2 gives background information about the two databases used as example application. In chapter 3 the use cases, which will guide the work in the following chapters, will be specified along with the requirements for the rule system. In chapter 4 the rule system to be used is introduced. Chapter 5 describes the EruS prototype to handle the use cases specified in chapter 3. Chapter 6 covers the evaluation of the rule system with the developed ECA rules. Chapter 7 concludes the thesis with a discussion of the results achieved in this project and suggestions for future work.

2 Background

This section gives background information about the two databases relevant to the bioinformatics use case: PDB and SCOP.

2.1 The Protein Data Bank - PDB

PDB belongs to the group of structural databases in bioinformatics and hence stores data on protein structures. PDB was established 1971, it gained importance by the drastic increase of data in 1980s and in the early 1990s it made its way to a standard reference in bioinformatics as the majority of journals required PDB accession codes for publications (Berman et al. 2000). Since 1998 the Research Collaboratory for Structural Bioinformatics (RCSB), a non-profit consortium based in the United States working on three-dimensional structures of biological macromolecules, has been responsible for the maintenance of PDB.

2.1.1 Data

According to Berman et al. (2000) PDB is the "single worldwide archive of structural data of biological macromolecules", with public availability. The holdings as of August 2005 cover 32149 total macromolecular structures, including protein stuctures[2]. This number is increasing enormously, as new data is added constantly (e.g. 5507 new structures were released in the year 2004).

Data in PDB contain information about theoretical models as well as experimental structures of proteins, including structure factors and nuclear magnetic resonance (nmr) constraints (RCSB 2004*b*).

The data entered in PDB is processed as follows (Berman et al. 2000). Primary data is deposited from a researcher; the dataset is assigned a unique PDB identifier and is loaded to the core database. The new entry is then annotated and validated by the RCSB staff. After validation, the deposition's author must approve the entry before its distribution. The whole process from deposition to completion of data processing is performed within less than two weeks.

2.1.2 Architecture

The underlying architecture is an integrated system built of heterogeneous databases as described in Berman et al. (2000). Its main components are the core relational database and the ftp archive. The core database contains primary experimental and coordinate data; all deposited information is stored here. The ftp archive consists of ASCII files and contains final curated data files as well as data dictionaries.

Besides these two modules there are a Property Object Model-based database with indexed objects containing native and derived properties, the Biological Macromolecule Crystallization Database with literature derived information and an index of all textual content.

[2]http://www.rcsb.org/pdb/holdings.html (accessed 9 August 2005)

2.1.3 Query Mechanism

PDB provides five different query interfaces (RCSB 2004*b*):

PDB ID allows search simply by the unique PDB ID of an entry

QuickSearch provides keyword search in both the text of mmCIF (macromolecular Crystallographic Information File) files and the Web pages.

SearchLite offers keyword search in the text of mmCIF files.

SearchFields supports queries over different data items by a customizable query form.

Status Search allows access to information about unreleased entries by a number of attributes (e.g. author or release date).

Both SearchLite and SearchField are accessible to other data sources through a CGI application programming interface.

Query results are shown as general overview, but can be explored with detailed information and also downloaded as data files.

2.1.4 Data distribution and update policies

PDB data is distributed in three formats: the proprietary PDB file format for coordinate files,the XML (eXtensible Markup Language) file format and mmCIF (Deshpande et al. 2005). PDB data can be obtained from different sources; first of all the primary PDB sites, which are updated weekly. Also, PDB mirror sites can be accessed; the web-based mirror sites are also updated weekly whereas ftp-only mirror sites define their own update intervals. Besides, a snapshot of the database is published once a year and is available both via ftp and on DVD[3].

The updates of experimental structures are logged in the title sections of the PDB coordinate entry files. The records `OBSOLETE` for obsoleted structures and `REVDAT` for modified structures, respectively, contain information about the update (Callaway et al. 1996). For each update date distinct file lists[4] exist for added, obsolete and modified PDB, structure factor and nmr files. PDB provides a script[5] using these file lists to obtain data from any given update date.

2.2 The Structural Classification of Proteins Database - SCOP

SCOP contains all proteins of known structure; the entries are either derived from PDB data or from literature references. The protein structures are broken down into domains (i.e. the conserved structural features) which are hierarchically ordered "according to their evolutionary and structural relationships" (Andreeva et al. 2004). The current version (1.69) of SCOP was released in July 2005 and consists of 70859 protein domains, based on the 25973 PDB entries current on 1

[3]http://www.rcsb.org/pdb (accessed 7 August 2005)
[4]ftp://ftp.rcsb.org/pub/pdb/data (accessed 2 August 2005)
[5]ftp://ftp.rcsb.org/pub/pdb/software (accessed 2 August 2005)

October 2004 and one literature reference[6]. These domains are clustered in 2845 families, 1539 superfamilies, 945 folds and 8 classes.

2.2.1 Data

The smallest unit in the SCOP hierarchy is the protein domain (Conte et al. 2002). Each domain entry in SCOP is derived from one PDB structure and each PDB structure can be the source for several protein domains. Data sets in SCOP contain corresponding PDB entry names; new protein structures that are released by PDB also appear in the next revised version of SCOP. Nmr and structure factor files as well as theoretical models are not regarded for those updates (private communication with Alexey G. Murzin). According to their species the domains

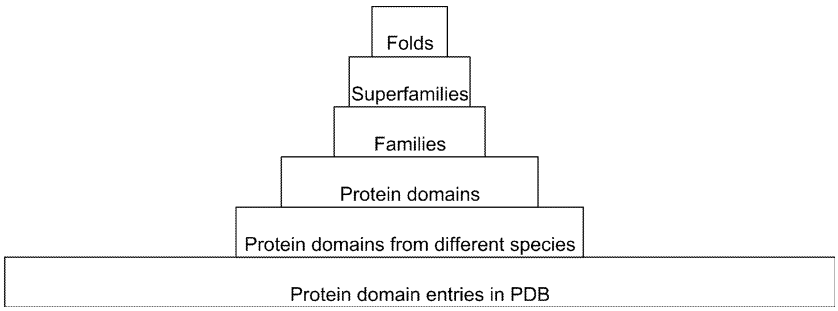

Figure 2: The levels of the SCOP hierarchy

are grouped and classified in the hierarchy of families, superfamilies, folds and protein domain classes as shown in figure 2 (Murzin et al. 1995).

The SCOP hierarchy is of dynamic nature. In the process leading to a new release of the database new protein structures are added and the hierarchy is revised. This may lead to changes in the hierarchy in a new release (Andreeva et al. 2004). Nevertheless, the entries in SCOP can be identified unambiguously, as each node has a unique identifier which is kept stable over the releases (Conte et al. 2002).

2.2.2 Architecture

SCOP data is provided in a set of hypertext pages on the world wide web (Murzin et al. 1995). The data behind these hypertext pages is stored in a flatfile and each node in the hierarchy can be displayed as one page. This facilitates the representation of more complex relationships as pages can be crosslinked. To allow the introduction of new classification levels, the SCOP implementation is built on a description of the data format's structure (Conte et al. 2002).

2.2.3 Query Mechanism

As mentioned above (section 2.2.1), each entry has a unique identifier, the *sunid*. This identifier can be used as keyword in the search engine provided along with the hypertext pages, but it is also possible to use other search keys as for example PDB identifiers or words that appear on the SCOP pages (Murzin et al. 2005).

[6]http://scop.mrc-lmb.cam.ac.uk/scop/count.html#scop-1.69 (accessed 12 August 2005)

2.2.4 Data distribution and update policies

Each new SCOP release is based on a PDB snapshot, which is processed by the SCOP authors for several months before the new version of SCOP can be released. The work on the current release started in October 2004 and was finished in July 2005. The long duration is due to the manual classification of the protein domains, performed by visual inspection and comparison of the structures assisted by automatic tools (Murzin et al. 1995). The domain definitions and classifications can change between releases as it is based on evolutionary relationships and new research results may give new information on interrelations.

Apart from the hypertext pages provided, SCOP data is also available in four parseable files which together contain all data available in SCOP: the hierarchy of nodes, classifications, descriptions as well as comments (Murzin et al. 2005).

2.3 Related Work

This project aims to provide a rule-based mediator to reactively process updates in heterogeneous databases. Different approaches have been taken to establish similar behaviour in databases and information system, mostly providing a new repository or application based on data from different sources. In Fahl et al. (1993), for instance, an architecture for active mediators is proposed, which implements an additional layer between a number of data sources and the applications using the data, hence a new system that offers transparent access to the underlying data sources. Similar to this, query-based approaches to data integration are taken in bioinformatics, for example the TINet system (Eckman et al. 2001), where several bioinformatics data sources are integrated behind a common query interface. The work on TINet goes towards integrating a larger number of data sources. This is not the aim for EruS, as only one database (SCOP) derived from another (PDB) has to be updated to provide consistency.

The ASTRAL compendium (Chandonia et al. 2004) constitutes an integrated repository for protein analysis. It provides query mechanisms as well as analysis tools to data partially derived from PDB and SCOP, including preliminary classifications of new PDB entries which otherwise would not be classified until a new SCOP release. However, ASTRAL data is restricted to the use by the tools provided in the ASTRAL compendium. It does not update the existing SCOP parseable files which are used for studies and analysis by other systems and applications as mentioned in section 1.1.

3 Problem Description

It is common practice in bioinformatics to keep local copies of frequently used databases to be independent of the connection to one of the remote servers (Alfares et al. 2005). The scenario regarded here consists of a local copy of the SCOP data and a local script that downloads update data from PDB. These two components are to be connected by a rule system providing ECA rules to handle the PDB updates. Changes in PDB are EVENTS in the sense of ECA rules and corresponding ACTIONS must be performed on the SCOP copy, if the CONDITION is true.

The use cases to be considered in the project concern only the use of a local copy of SCOP. A complete solution of the limitations that were described in section 1.2 exceeds the scope of this project. The resulting system can be further developed and adapted for use with the original SCOP version in future work.

3.1 Use Cases

The task of the rule engine is to detect events in PDB and handle them in the SCOP copy. The following events can occur:

Deletion of a PDB entry: In the case of a deletion in PDB, the corresponding SCOP entries include information about a PDB entry that no longer exists. This inconsistency must be handled by updating the SCOP entry and adding the information about the now obsolete PDB entry and its substitute (Alfares et al. 2005).

Update of a PDB entry: If new information about a protein structure is available or an error in a PDB entry is discovered, this entry will be updated in PDB. The handling of this update in SCOP is not trivial, as the change to the PDB entry might affect the classification of the structure domain in SCOP. The classification of domains is not fully automated, therefore it is not possible to change it as part of the ECA rule handling of the PDB update event. An alternative is to edit the SCOP entry and add the information about the modified PDB entry to indicate the potential change of its classification in the next SCOP release.

New entries in PDB: A more advanced use case would be not only to handle inconsistencies (i.e. updates or deletes of existing data in PDB), but also to assure the actuality of SCOP data (i.e. to handle inserts in PDB immediately to keep PDB and SCOP synchronised). This includes classification of new structures. As mentioned above, classification is not an automated procedure. Adding the information would naturally be possible, but only without any SCOP specific information, which would not be expedient since the added data would be of no use for analysis performed on SCOP data. Therefore, the handling of new entries is not feasible for the automatic rule engine developed at this stage of the project.

Thus, it can be summarised that the use cases considered in the development of the first version of EruS are the deletions and updates of PDB entries.

3.2 Heterogeneities of Database Systems

The limitations and problems described in 1.2 can be traced back to the fact that the given databases are heterogeneous in a number of properties. Bornhövd (2000) describes the following classification of heterogeneities in database systems in general:

1. **Heterogeneities concerning the Environment** are differences outside the data sources themselves. These concern:
   - Hardware
   - Software (e.g. Operating System)
   - Communication system (e.g. interfaces, protocols)
2. **Heterogeneities concerning the database management system** are differences that are independent of the data content. These concern:
   - Representation model (e.g. relational vs. object-oriented)
   - Query languages
   - Data management functionality, (e.g. access interfaces, additional functions such as consistency assurance or triggers)
3. **Heterogeneities concerning the modelling layer** are differences in the representation of the real world:
   - Structural heterogeneities appear, if different model constructs are used for equivalent aspects of the real world.
   - Semantic heterogeneities appear, if different semantic concepts are used for equivalent aspects of the real world.
   - Identification conflicts appear, if different properties are used for object identification.
4. **Heterogeneities concerning the data layer** are differences in instances that represent the same real world object:
   - Contradictory data sets have different values for semantically equivalent data objects.
   - Incomplete data sets have missing values for attributes in equivalent conceptualizations.

The properties mentioned above (section 3.2) can now be limited to the ones that are applicable in this project as shown in figure 3. The problems here are both the maintenance and the usage of information as the connections on the data level (i.e. between data sets in distinct sources) are handled manually (Murzin et al. 1995). This concerns local copies of the data sources as well as the original source of SCOP and its derivation from PDB. The underlying idea is to resolve possible inconsistencies and to keep the data up-to-date. These two aspects concern the data sets in the data sources, hence the heterogeneities on the data layer (4) need to be overcome. An update of a PDB entry can lead to a contradictory dataset in SCOP. The cause of this problem is the fact that two distinct, independent

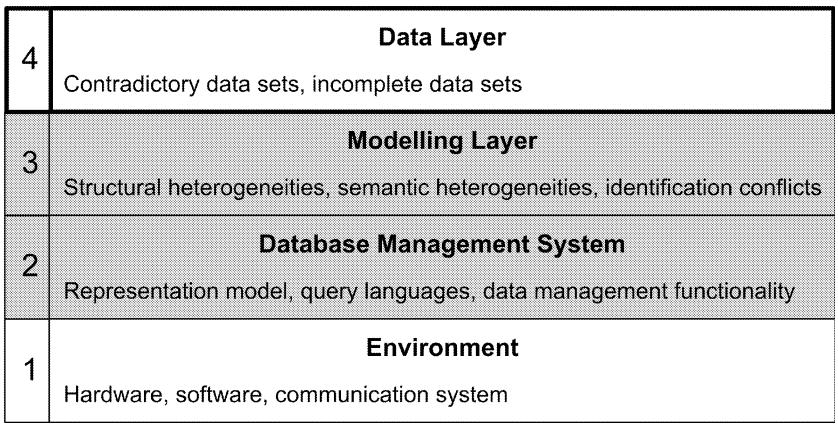

Figure 3: Heterogeneities of database systems. Shaded areas show layers especially relevant for EruS

databases have an overlap on the data level as SCOP contains data derived from PDB.

However, this problem cannot be tackled by regarding the data layer only; it is a necessary requirement to deal with the heterogeneities on the other layers in order to allow a solution on the data layer. For this project especially the database management system (2) and the modelling layer (3) are relevant issues to take into account. Whereas the heterogeneities concerning the environment (1) regard a lower communication level than is aimed at here and therefore this technical layer can be abstracted and does not need to be regarded further.

The heterogeneities in the database management systems rule out the option to just merge the two systems. We are not dealing with two relational databases where a union of tables may solve the problem. Furthermore the fact that there are no direct interfaces between PDB and SCOP requires a mediator between the systems. The demands on this mediator are increased as the SCOP system to be handled only consists in flatfiles and therefore of course does not provide any functionality to handle triggers or rules. The heterogeneities on the modelling layer put another requirement on the mediator. Both systems use different identifiers for their data and one entry in PDB may have several corresponding entries in SCOP. This discrepancy needs to be overcome by the connecting system EruS.

3.3 Alternative Solutions

As already indicated in section 2.3, a possible solution can be the use of a database integration system. This, however, does not lead to the result aimed at here, since these systems provide a common interface for a number of databases or integrate data in a new repository rather than updating existing data as in this case the SCOP flatfiles.

The functionality of a mediator can also be provided by ad-hoc scripts written individually for a given scenario. The problem here is that the functionality and the processing rules are part of the code and not stored independently and structured. This makes those scripts hard to maintain and extend, whereas a rule system can easily be adapted and extended by adjusted or additional rules. This is of major importance here since this project will only be a starting point and tackle only a subset of the use cases the scenario consists of as described in section

3.1.

The new approach in EruS is to use a rule-based mediator for update propagation and processing between the two bioinformatics databases. The goal, namely to provide updated SCOP data, and the functionality needed can be provided better by a rule system than by the approaches mentioned above.

3.4 Limitations

The local version of SCOP which is being updated by EruS can be seen as a database which consists of the latest SCOP release and additional information from PDB updates. It is not a complete derivation and especially not complete with regard to the current version of PDB. However, the updates performed to the local version will not contradict the evolution of SCOP with the next upcoming release since the classification will not be altered and all data is always synchronised with PDB which is the basis for the development of a new SCOP release. On publication of a new release the local copy can be deleted and the update can be run for all PDB updates since the date of the frozen PDB version on which the new SCOP release is based. This way no update information will get lost.

4 The Rule System

The approach taken in EruS to solve the problem described in the previous chapters is based on rules implementing reactive behaviour. As foundation for the prototype a rule system is used which handles ECA rules.

4.1 ECA Rules

An ECA rule is a rule of the format on EVENT if CONDITION do ACTION (Alfares et al. 2004). The event can be either primitive or composite (i.e. consisting of several events). On occurrence of an event the condition is checked and if it is true, the action is executed. ECA rules provide the possibility to implement reactive functionality on different kinds of data sources and applications. In the given scenario the rules are used to react to updates in PDB (events) and execute actions to update the local SCOP files.

Also, the rules are held in a clearly structured format in a single rule base instead of being scattered and hidden in the code of a program that can be substituted by the rule system (Papamarkos et al. 2003). This improves modularity and hence makes the software easier to maintain and extend. This is of major importance for this project as the resulting system is to be further developed in future work.

4.2 Requirements

The rule system to be used in this project must fulfil the following requirements:

1. ECA rule support is needed for reactive behaviour needed for SCOP to react to events in PDB. This feature is also important for a clear and straightforward definition of the rule set as otherwise the rules are concealed within the program code.

2. Handling of different data sources as it is needed for the two heterogeneous systems PDB and SCOP. Support for remote sources is also desirable as this may be required for further stages of the project.

3. The possibility to execute an external program as action is needed to update the SCOP data. As the SCOP data consists only of flatfiles the functionality to update these flatfiles must be provided by an additional script to be called as the action part of a rule.

4. Support for composite events is needed to avoid constraints for future work. Even though only primitive events are needed at this stage, further development will certainly require composite rules. Besides, the common notion of ECA rules includes composite events (Alfares et al. 2004, Papamarkos et al. 2003) and a system should comply with this standard.

5. Free availability of the rule system for this project is necessary.

4.3 ruleCore™

The rule system ruleCore™ supports ECA rules and provides a GUI for straightforward rule definition which includes composite events. Also, it can handle numerous data sources and offers different data input interfaces. The action to be executed on event detection can be an external program or a new event to be triggered. Furthermore, ruleCore™is free for academic research. Thus, ruleCore™ fulfils all the above (section 4.2) described requirements.

4.3.1 Markup Language

The configuration for ruleCore™ which includes the rule definitions is specified in the XML based ruleCore™ Markup Language (rCML) (Seiriö & Berndtsson 2005). The necessary configuration file can be easily constructed using the ruleCore™ Designer. Since the file is in ordinary XML format it can also be edited manually. The term "situation" is used for composite events in ruleCore™.

4.3.2 Architecture

ruleCore™ consists of four modules: the Designer, the Event Generator, the Engine and the Monitor (MS Analog Software kb 2003). Rules can be defined using the ruleCore™ Designer which provides a GUI for configuration and stores the configuration including the rules in an rCML file (see 4.3.1). When creating rules with the Designer, the ruleCore™ Event Generator can be used to test and debug rule definitions. The core of the ruleCore™ architecture is the ruleCore™ Engine, which processes incoming events based on rule definitions. Figure 4 illustrates event processing in ruleCore™.

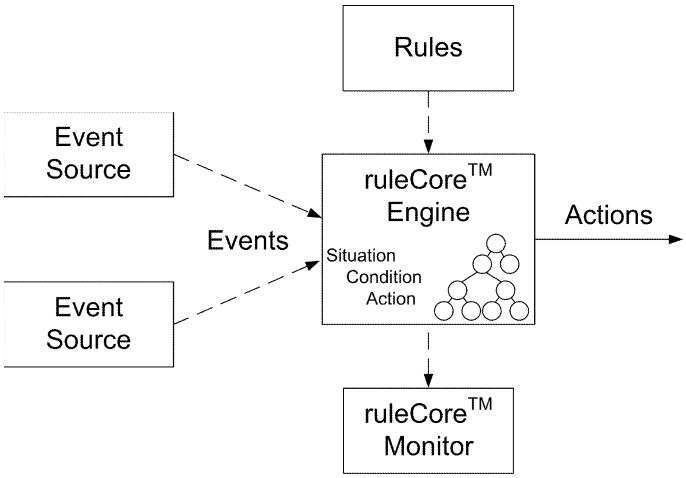

Figure 4: Rule processing in ruleCore™

The ruleCore™ Engine provides four event input interfaces for external event sources. These are implemented with Event Input Servers which can be internal or stand-alone processes:

- The Socket Event Input Server requires programming to be usable.
- The XML-RPC Event Input Server also requires programming which can be based on XML-RPC standard libraries.

- The Tibco Rendezvous Event Input Server is a stand-alone input server which does not require programming but must be started manually.
- IBM WebSphere Event Input Server is a distributed message queue which does not require custom programming either.

Besides the Engine, the Event Generator and the Designer, ruleCore™ also includes the ruleCore™ Monitor, a GUI to monitor event occurences, rule instances and the internal state of the Engine.

4.3.3 System Requirements

ruleCore™ runs on Windows 2000/XP, Linux and Solaris systems (MS Analog Software kb 2003). It requires postgreSQL, which is included in the installation executables for Windows.

5 The EruS Prototype

The EruS prototype developed is a local system with an installation of ruleCore™, holding and processing the defined ECA rules (section 5.2) as basis. The data to be updated by the rule system is stored in the form of the SCOP parseable files holding all information covered by SCOP (section 2.2.4). The system is invoked by running the PDB event connector (section 5.1) which downloads the update information, generates the events and sends them to the rule engine. In processing the events the rules are used and the actions are executed in terms of the SCOP wrappers (section 5.3) which update the SCOP data. Figure 5 shows the basic architecture of the EruS system.

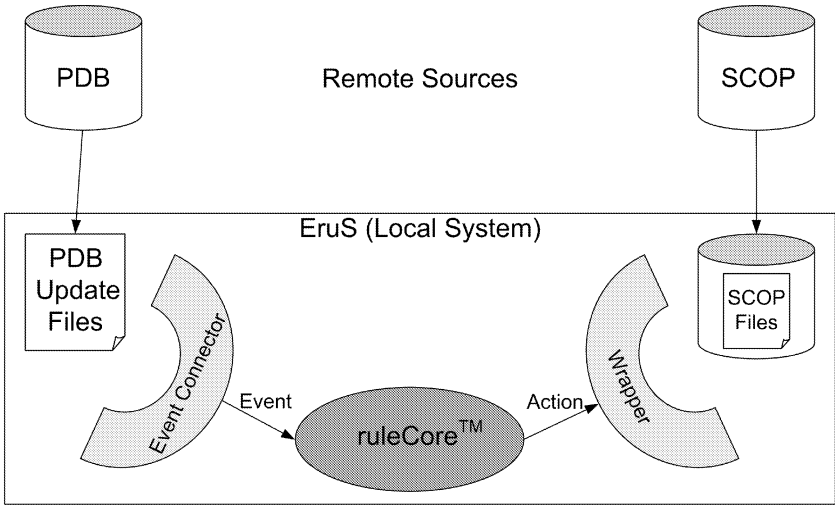

Figure 5: Architecture of EruS

5.1 PDB Event Connector

PDB provides the following files with each update:

- `{modified|added|obsolete}.pdb`,
- `{modified|added|obsolete}.nmr`,
- `{modified|added|obsolete}.sf`,
- `models_{modified|added|obsolete}.pdb`

These files contain lists of updated PDB structures (`.pdb`), nuclear magnetic resonance constraints (`.nmr`), structure factors (`.sf`) and theoretical models (`models_`)

The PDB event connector (for source code see appendix A) is based on a PDB update script available from the RCSB ftp server[7]. It has been altered and extended and now downloads only the update files relevant for the SCOP updates. These are the modified and obsolete PDB structures, because nuclear magnetic resonance constraints, structure factors and theoretical models do not affect the

[7] ftp://ftp.rcsb.org/pub/pdb/software (accessed 2 August 2005)

SCOP entries as stated in section 2.2.1 and added entries are not handled here as specified in section 3.1. After downloading the two file lists, the listed PDB files are downloaded and unpacked.

The information needed about modified and obsolete entries for further processing is then extracted from the updated files. For all updates, the date and PDB identifier are extracted. In a modified PDB entry the type of modification (in the example below: `1`) and the modification comment (in the example below: `JRNL DBREF REMARK`) are included.

```
REVDAT   2   19-JUL-05 1D4V     1        JRNL   DBREF  REMARK
```

Whereas for obsolete entries it is necessary to obtain the identifier of the entry now replacing it (in the example below: `1VRY`).

```
OBSLTE      26-JUL-05 1ZHD       1VRY
```

With this information an event is generated and sent to the rule engine via the Socket Event Input Server, which accepts a string of the following format (MS Analog Software kb 2003):

```
<integer>,type:<value><integer>,param,<name>:<value>
```

The integers provide the length of the value strings; the event needs exactly one type and can have several parameters. The following example is an event sent to ruleCore^{TM} by the PDB event connector in the case of an obsolete entry occurring. The PDB entry with identifier `1zhd` has been replaced by the new PDB entry `1VRY` with the PDB update of 25 July 2005 (lower and upper case of PDB identifiers are equivalent).

```
11,type:obsoletePDB8,param,DATE:200507254,param,PDBID:1zhd4
   ,param,SUBST:1VRY
```

The PDB event connector accepts different command line arguments. It can be called with `perl pdb_download.pl latest` to process the changes made with the most recent PDB update, whereas `perl pdb_download.pl <YYYYMMDD>` uses the files of the particular update provided. If the user is unsure about the valid dates, `perl pdb_download.pl dates` provides a list of all update dates. When a new SCOP update has been released or EruS is used for the first time, `perl pdb_download.pl since<YYYYMMDD>` is useful as it processes all updates since the date provided, which should than be the date on which work on the SCOP release used started.

5.2 ECA Rules

The ECA Rules to be used as basis for the event handling by ruleCore^{TM} are defined in the ruleCore^{TM} Designer and stored in the file `erus.rcml` (see appendix B). The event strings coming in to ruleCore^{TM} through the Socket Event Input Server are processed based on this configuration file. If the action executed is an external program or script, ruleCore^{TM} sends the event, which triggered the action, with its parameters to the standard input in XML format.

In the current stage of the project the following four ECA rules are defined:

modifiedPDB

```
ON modification in PDB
```

```
IF modid != 0 and modid != 1
DO update SCOP
```

This rule is instantiated on occurrence of a modification event in PDB. This event has four parameters which contain all necessary information about the update: `DATE`, `PDBID`, `MODID` and `MODCOM`. A condition is evaluated to check if the modification identifier is not 0 or 1, as only the other types of modifications are possibly relevant for SCOP. If the condition is true, the action taken is the execution of the SCOP Wrapper `SCOP_modifiedPDB.pl` (section 5.3.1).

obsoletePDB

```
ON deletion in PDB
DO update SCOP and trigger new event
```

This rule is an event-action rule. It is instantiated when an event arrives from PDB which notifies about an obsolete entry. The event consists of the parameters `DATE`, `PDBID` and `SUBST`, the last being the PDB identifier of the new entry now replacing the obsolete entry with PDB identifier `PDBID`. On occurrence of this event type the SCOP wrapper `SCOP_obsoletePDB.pl` (section 5.3.2) is executed as action

obsoleteLeaf

```
ON possibly obsolete leaf node in SCOP
DO update SCOP and trigger new event
```

This event-action rule is instantiated when a leaf node in SCOP is marked as "possibly obsolete" by the SCOP wrapper `SCOP_obsoletePDB.pl` (section 5.3.2). The wrapper `SCOP_obsoleteNode.pl` (section 5.3.3) is called with the parameters `DATE`, `NODE` and `PARENT` as action part of the rule to take the update one level up the hierarchy.

obsoleteNode

```
ON possibly obsolete child node in SCOP
IF number of child nodes equals number of obsolete
   child nodes
DO update SCOP and trigger new event
```

This rule is also invoked on an event coming from a SCOP wrapper. When a SCOP node has been marked as possibly obsolete by `SCOP_obsoleteNode.pl` (section 5.3.3), it also sends this event consisting of five parameters: a `NODE`, its `PARENT`, the parent's number of `CHILDNODES` and obsolete child nodes (`OBSCHILDNODES`) and the `DATE`. The condition evaluated checks if the number of child nodes equals the number of obsolete child nodes as in this case the update cascades up the hierarchy and the script `SCOP_obsoleteNode.pl` is executed with a new set of event parameters in the action part of this rule.

In ruleCore™ the event part of a rule is always specified as "situation", the concept for composite events in ruleCore™ (Seiriö & Berndtsson 2005). Even though the events described here are all primitive, they still are "situations" in the rcml file. An example of rule definition in rcml can be seen in listings 1 to 5.

Listing 1: Rule to handle an obsolete SCOP node

```
<rule create="single" delivery="all" limit="None" parameter=
   "append" name="obsSCOPNode">
    <description>
        Handles obsolete non-leaf nodes in SCOP and calls
           update script for SCOP.
    </description>
    <event-ref enabled="yes">obsNode</event-ref>
    <condition-ref enabled="yes">CHILDNODES</condition-ref>
    <action-ref enabled="yes">SCOPobsNode</action-ref>
    <minus-action-ref enabled="no"/>
    <instance-limit/>
    <filter/>
</rule>
```

Listing 2: Situation detected when a node is obsolete

```
<event-def type="composite" name="obsNode">
    <detect-event/>
    <no-detect-event/>
    <event-selector name="">
        <condition/>
        <scope>global</scope>
        <events/>
    </event-selector>
    <detector>
        <event-ref type="event">obsoleteNode</event-ref>
    </detector>
</event-def>
```

Listing 3: Event constituting the situation “obsNode”

```
<event-def type="basic" name="obsoleteNode">
    <parameters>
        <parameter type="string" name="DATE"/>
        <parameter type="string" name="NODE"/>
        <parameter type="string" name="PARENT"/>
        <parameter type="string" name="CHILDNODES"/>
        <parameter type="string" name="OBSCHILDNODES"/>
    </parameters>
</event-def>
```

Listing 4: Condition evaluated on occurrence of the situation “obsNode”

```
<condition-def always-true="no" composite="no" name="
   CHILDNODES">
    <parameters>
        <parameter name="CHILDNODES">
            <item-ref type="event"/>
            <param-ref>CHILDNODES</param-ref>
            <instance>all</instance>
            <function/>
            <key>no</key>
        </parameter>
```

```
        <parameter name="OBSCHILDNODES">
            <item-ref type="event"/>
            <param-ref>OBSCHILDNODES</param-ref>
            <instance>all</instance>
            <function/>
            <key>no</key>
        </parameter>
    </parameters>
    <expressions>
        <expression name="CHILDNODES">
            <lhs>
                <param-ref>CHILDNODES</param-ref>
            </lhs>
            <operator>equal</operator>
            <rhs>
                <param-ref>OBSCHILDNODES</param-ref>
            </rhs>
        </expression>
    </expressions>
    <composite-condition/>
</condition-def>
```

Listing 5: Action executed on occurrence of "obsNode" if "CHILDNODES" is true

```
<action-def name="SCOPobsNode">
    <action-item type="script" name="SCOPobsNode">perl SCOP\
       _obsoleteNode.pl</action-item>
</action-def>
```

5.3 SCOP Wrappers

There are three different SCOP wrappers, written in Perl, called as part of the different ECA rules and one Perl script to lookup a SCOP comment. All wrappers read and write from and to SCOP parseable files. These are a set of four flatfiles each containing different information about the SCOP entries and hierarchy[8]:

`dir.cla.scop.txt_1.69` holds an entry for each SCOP domain containing identifiers, domain definitions and detailed information on the domain classification. It is used here to find the *sunids* connected to a certain PDB identifier.

`dir.com.scop.txt_1.69` holds both automatically generated and manually added comments for SCOP nodes. This file is written to by the SCOP wrappers to add information about the PDB updates.

`dir.des.scop.txt_1.69` holds a list of all SCOP nodes including their identifiers, types, descriptions and limited information on their classification. The file is not used in the SCOP wrappers.

[8] http://scop.mrc-lmb.cam.ac.uk/scop/release-notes.html (accessed 2 August 2005

dir.hie.scop.txt_1.69 represents the SCOP hierarchy only in the form of identifiers. From this file information about parent nodes of possibly obsolete SCOP nodes is obtained.

5.3.1 Modified Entry in PDB

The script SCOP_modifiedPDB.pl (see source code in appendix C.1) receives an event in XML format with the parameters DATE, PDBID, MODID and MODCOM. The event is parsed and all *sunids* connected to the given PDBID are looked up in dir.cla.scop.txt_1.69. The file dir.com.scop.txt_1.69 is then searched with each of these *sunids*. If there is already a line with this *sunid*, the new comment

```
PDB ID <PDBID> modified in PDB update <DATE> (modification
   type <MODID> <description of MODID>) <MODCOM>
```

is added to the line, otherwise, a new line for the *sunid* including the comment is added at the end of the file as shown in listing 6.

Listing 6: Line added to dir.com.scop.txt_1.69 by SCOP_modifiedPDB.pl

```
73394 ! PDB ID 110d modified in PDB update 20050801 (
   modification type 3 - coordinates or transformations)
   DBREF  SEQADV REMARK SCALE1
```

5.3.2 Obsolete Entry in PDB

The script SCOP_obsoletePDB.pl (see source code in appendix C.2) receives an event with the parameters DATE, PDBID and SUBST, it also starts with parsing the XML input and obtaining information from dir.cla.scop.txt_1.69 to find the respective lines in dir.com.scop.txt_1.69. The comment added to existing or new lines now is:

```
PDB ID <PDBID> is obsolete with PDB update <DATE>, replaced
   by PDB entry <SUBST>
```

An example is given in listing 7.

Listing 7: Line added to dir.com.scop.txt_1.69 by SCOP_obsoleteDB.pl

```
98815 ! PDB ID 1se1 is obsolete with PDB update 20050711,
   replaced by PDB entry 1VRS
```

Then the file dir.hie.scop.txt_1.69 is accessed to obtain each *sunid's* parent node. With this information, new events are generated for each *sunid* and sent to ruleCore^{TM} to check for cascading.

5.3.3 Obsolete Entry in SCOP

The script SCOP_obsoleteNode.pl (see source code in appendix C.3) receives an event in XML format with the parameters DATE, NODE, PARENT and for non-leaf nodes also CHILDNODES and OBSCHILDNODES, which at first are parsed for further handling. If the parameter CHILDNODES exists we are dealing with a non-leaf node whose child nodes are all possibly obsolete and this information is added as comment

```
all <CHILDNODES> child nodes are possibly obsolete
```

in `dir.com.scop.txt_1.69`, an example is given in Listing 8.

Listing 8: Line added to `dir.com.scop.txt_1.69` by `SCOP_obsoleteNode.pl`

```
81820 ! possibly obsolete child node: 79781 (due to PDB
   update 20050502) ! possibly obsolete child node: 79783 (
   due to PDB update 20050502) ! all 2 child nodes possibly
   obsolete
```

In any case the `PARENT`'s *sunid* is looked up in `dir.com.scop.txt_1.69` to add to an existing or new line (as in 5.3.1) the comment:

```
The only child <SUNID> of this node may be obsolete since <
   DATE> due to PDB update
```

Again, the `PARENT`'s number of child nodes and its parent node are obtained from `dir.hie.scop.txt_1.69` and a new event is generated and sent to ruleCore™ to check for further cascading of the update. The event parameters have new values now, corresponding to the next higher level of the hierarchy.

5.3.4 Lookup Comment

The script `SCOP_findcomment.pl` (see source code in appendix C.4) is a simple perl script to make the check for updates easier for the user if the application used with the SCOP flatfiles does not include the handling of the comments file `dir.com.scop.txt_1.69`. It simply prints out the comment line of the *sunid* given as command line argument when calling the script.

6 Evaluation

In this chapter the testing of EruS is described and the test results are analysed with respect to the use case definition in chapter 3.

6.1 Test Cases

EruS has been tested both for individual update dates and a series of update dates. As the work on the current SCOP release 1.69 was based on a PDB snapshot as of 1 October 2004[9], only update dates after this were considered. For individual update dates samples were taken from the whole range of update dates between October 2004 and August 2005. The effect of an update is only representable if all previous updates have been performed as well because otherwise information is missing from the local SCOP copies, which affects the cascading of obsolete information. Therefore the chronological succession of all updates since October is preferable for the analysis of the result and is described below in more detail.

6.2 Results

The results were taken from the update series including all updates between 1 October 2004 and 9 August 2005. Table 1 shows figures from the PDB Event

Table 1: Results: PDB Event Connector

	45 weeks	avg. per week
data downloaded		
number of files	3966	88.1
data amount	483.8 MB	10.8 MB
event calls	3959	88.0
modified	3883	86.3
obsolete	76	1.7

Connector. For 45 weeks it downloaded almost 4000 packed .pdb-files, comprising 483.8 MB of data. Based on this data, 3959 event strings were sent to ruleCore™ through the Socket Event Input Server. Thus, an average of 88 events per week has reached ruleCore™. The difference between files downloaded and events created is due to entries that were modified and obsolete during the same week.

The rule system processed events coming from the PDB Event Connector and the SCOP wrappers (figures in table 2). Alltogether 4274 rule instances were created by ruleCore™ on incoming events. Only 301 of them led to the execution of actions, the remaining events did not fulfil the specified conditions. The script `SCOP_modifiedPDB.pl` was called 47 times, `SCOP_obsoletePDB.pl` and `SCOP_obsoleteNode.pl` were called 76 and 178 times, respectively.

As a result of the actions called 423 comments, related to 299 SCOP nodes, were added to `dir.com.scop.txt_1.69` (table 3). The number of sunids overall affected is not the sum of the sunids affected with each comment type as sunids

[9]http://scop.mrc-lmb.cam.ac.uk/scop/count.html#scop-1.69 (accessed 12 August 2005)

Table 2: Results: ruleCore™

	Events detected		**Actions called**	
Rule	45 weeks	avg. per week	45 weeks	avg. per week
modifiedPDB	3883	86.3	47	1.0
obsoletePDB	76	1.7	76	1.7
obsoleteLeaf	137	3.0	137	3.0
obsoleteNode	178	4.0	41	0.9
Σ	4274	95.0	301	6.6

Table 3: Results: SCOP Wrappers

	comments added		sunids affected	
comment type	45 weeks	avg. per week	45 weeks	avg. per week
mod. PDB entry	67	1.5	67	1.5
obs. PDB ID	137	3.0	137	3.0
obs. child node	178	4.0	95	1.4
all child nodes obs.	41	0.9	41	0.9
overall	423	9.4	299	6.6

with "all child nodes obsolete" also have "obsolete child nodes" comments. The comments added did not excessively increase the size of the file or the number of the comments as it already contained comments for 1597 SCOP nodes (of 94442 existing nodes) before. The file is still the smallest of the SCOP flat files with a file size of 583 KB compared to 538 KB before the update.

The update for 10 months took approximately 20 hours to download and process, due to the slow download procedure provided in the PDB update script. The script has been improved and only data not locally available is downloaded. If already present data is used, the Event Connector is slowed down by deliberate delays, as ruleCore™ showed problems with processing many events received almost at the same time. However, the performance of the event connector was not a major criterion here as this extensive update needs only to be performed on the first use of EruS or when a new version of SCOP is released. Afterwards only small updates need to be obtained regularly.

6.3 Discussion

The results described in 6.2 are discussed here with respect to the description of the application scenario given in chapter 3

6.3.1 Use Cases

The use cases to be handled by the EruS prototype were specified as the deletion and update of existing PDB entries.

Deletion of a PDB entry This is handled by the system in several steps. Comments are added to the flatfile. In the test case 137 SCOP leaf nodes were marked as possibly obsolete due to PDB updates. The potential cascading of the deletion is also considered and 178 comments were added about obsolete child nodes of higher level nodes which resulted in 41 of them being marked as possibly obsolete themselves because all of their child nodes were marked as such. In any case the user is made aware of inconsistencies potentially arising from deletion of PDB entries by the comments added.

Update of a PDB entry The system adds information about modified PDB entries to the correspoding SCOP nodes. Only the SCOP relevant modifications lead to action execution. This resulted in 67 SCOP nodes marked with data about the PDB update. The processing of modifications takes longer than the processing of obsolete PDB entries as the condition needs to be evaluated. However, this does not pose a problem for results achieved by EruS.

6.3.2 Extendibility and Maintainability

A main reason to use a rule system was the maintainability and extendibility of the prototype. The rcml file developed here as configuration for the ruleCore™ Engine can easily be adjusted either using the ruleCore™ Designer or with any texteditor, according to the rcml specifications. Hence, the rule base can easily be extended and maintained as the different components of the rules are clearly defined in a modular way.

Different wrapper scripts were defined as action parts of different rules. If the actions are to be changed in future work, each individual script can easily be replaced. In this granularity the wrapper scripts are also easily maintainable. On a lower level of granularity the maintainability and extendibility of the wrappers decreases since then the actual code needs to be changed.

6.3.3 Limitations

The data being updated is part of a local system, it is not to be confused with the original version of SCOP. The files modified by EruS at this stage of the project can not be compared one-to-one to the files of a new SCOP release. This is due to the fact that a preliminary version of the new release is not generated but rather information added for the time in between SCOP releases. However, the local updated SCOP copy in the form of the four parseable files can be used in any application based on the format description of those flat files because the SCOP wrappers edit the file `dir.com.scop.txt_1.69` and the resulting file changes comply with the file specification in the SCOP release notes. EruS is of most use together with a system or application that already involves the comment file because otherwise the comments need to be checked separately, for example with the script provided.

6.3.4 Heterogeneities

At this stage we can now see how the heterogeneities of databases according to Bornhövd (2000), explained in section 3.2, are overcome by EruS, working as

mediator between the two heterogeneous databases PDB and SCOP.

1. Heterogeneities concerning the Environment: As explained before (section 3.2), these are not considered here.

2. Heterogeneities concerning the database management system: Regarding the database management system the mediator provides an interface between the two databases SCOP and PDB, which obtains updates from PDB on the one side and deals with flatfiles from SCOP on the other side and also processes the data obtained based on rules.

3. Heterogeneities concerning the modelling layer: When it comes to the modelling layer the problem is the different use of identifiers in the two databases. EruS handles this by mapping PDB identifiers to a set of SCOP *sunids* within the wrapper scripts based on information provided in SCOP.

4. Heterogeneities concerning the data layer: The result of the connection established between PDB and SCOP by the mediator EruS is less heterogeneity on the data layer as contradictory data sets are avoided and the two systems are now consistent regarding updates to existing entries.

7 Conclusion

The general problem tackled here is the interaction of two heterogeneous databases. In bioinformatics this is a very common problem, especially because it is a mainly data-driven science (Backofen et al. 2004).

The aim of this project was to develop a set of ECA rules to be used in a rule system serving as mediator between two heterogeneous databases. This has been exemplified in the bioinformatics scenario consisting of the Structural Classification of Proteins database which is derived from the Protein Data Bank.

The rule-based mediator EruS (ECA rules updating SCOP), built on the rule engine ruleCore^{TM} has been developed in this project to establish reactive behaviour between the databases. The rule engine is complemented by wrapper scripts obtaining data from PDB and SCOP and modifying data in SCOP. A set of ECA rules has been defined in the XML-like `rcml` format. These are used by ruleCore^{TM} to process the incoming events. EruS handles updates and deletions of existing PDB entries which are processed by the rule system and lead to modifications of the SCOP parseable files containing all data included in SCOP, which can be used in local applications. Tests proved EruS to be functioning well and fulfilling the needs posed by the application scenario, but also showed some possibilities for enhancements in future work (section 7.2).

7.1 Contribution

EruS is a project that combines two areas of research. It is a step to bring forward the use of rules in order to enhance information propagation in bioinformatics and also to improve database systems in general. In bioinformatics, rules can assist to simplify the use of the vast data amounts and by this optimise research leading to better founded results. From a database perspective, bioinformatics is an ideal field to apply reactive rules and deepen research by using technologies in a limited application area before taking further steps.

For the specific use case considered, EruS overcomes some of the limitations the use of SCOP and PDB has. Inconsistencies arising from deletions in PDB are no longer a problem as the deletions are noted in SCOP. Information about modifications of PDB entries is also propagated to SCOP and can easily be checked by the bioinformatics researcher for influences on the analysis conducted. Adding comments to SCOP nodes based on information about obsolete and modified entries in PDB helps to avoid flaws in research and makes additional information more convenient to obtain.

The rule system ruleCore^{TM} has not been used in a context like this before. It can now be seen that this software is not only of use in industrial application environments but also in academia, as here in bioinformatics research. In general, it can be said so far that rule-based mediators may provide a valuable approach for bioinformatics update problems, as has been shown in this exemplifying scenario.

Moreover, the system developed here puts only minimal requirements on the data sources themselves. In particular the underlying rule system ruleCore^{TM} is very flexible and the approach taken here for the bioinformatics databases can be transferred to different application areas.

7.2 Future Work

The future work suggested for the EruS system arises from both the restrictions regarding the use cases to be handled (determined in section 3.1) and the results achieved during evaluation (section 6).

In the evaluation EruS showed good functionality on the one hand but on the other hand the performance needs to be improved for practical use. In order to achieve results here, a revision of the PDB Event Connector is most promising.

As defined in the application scenario (section 3) EruS currently only handles deletion and modification of existing PDB entries. This solves inconsitencies in SCOP resulting from changes in PDB, but actuality of the data with respect to new PDB entries is not considered. However, the full functionality of the underlying rule system is far from being exhausted by the system's current stage of development and, as intended, leaves options for further extension and improvement.

In a future scenario, EruS should be extended with composite events and the updates to SCOP can be applied to all parseable files, not only the comments file. The existing SCOP hierarchy has to be used, deletions can cascade in the hierarchy but should not affect it in a way that it can contradict new SCOP releases.

In more advanced stages, semantics should be added to the EruS rule base in order to include the handling of new PDB structures by preliminary classification within the SCOP hierarchy. For this step, the ASTRAL compendium's (Chandonia et al. 2002) handling of PDB updates can be surveyed as it includes a preliminary classification of new PDB structures. Here it has to be kept in mind that this is only a tentative classification which may later be changed in the manual update. The automatic processes regarding SCOP can only enhance but not replace the manual work based on human expert knowledge since this is an important characteristic of the SCOP database.

The final goal to be reached by further development of EruS is a completely updated SCOP copy which is consistent and complete with regard to the latest PDB update and which can be used in the same way as the original SCOP release. Once this is achieved for local copies, the last remaining step is to apply the system to the original SCOP version.

In general, more research in the use of rule-based systems as mediators for update propagation in bioinformatics databases may give useful results for the improvement of bioinformatics databases.

8 References

Alfares, J. J., Bailey, J., Berndtsson, M., Bry, F., Dietrich, J., Kozlenkov, A., May, W., Pătrânjan, P. L., Pinto, A., Schroeder, M. & Wagner, G. (2004), State-of-the-art on evolution and reactivity, Technical report, REWERSE.

Alfares, J. J., Berndtsson, M., Bry, F., Eckert, M., Henze, N., May, W., Pătrânjan, P. L. & Schroeder, M. (2005), Use-cases on evolution, Technical report, REWERSE.

Andreeva, A., Howorth, D., Brenner, S. E., Hubbard, T. J., Chothia, C. & Murzin, A. G. (2004), 'Scop database in 2004: refinements intergrate structure and sequence family data', *Nucleic Acids Research* **32**, 226–229.

Backofen, R., Badea, M., Burger, A., Fages, F., Lambrix, P., Nutt, W., Schroeder, M., Soliman, S. & Will, S. (2004), State-of-the-art in bioinformatics, Technical report, REWERSE.

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N. & Bourne, P. E. (2000), 'The protein data bank', *Nucleic Acids Research* **28**, 235–242.

Bornhövd, C. (2000), Semantikbeschreibende Metadaten zur Integration heterogener Daten aus dem Internet, PhD thesis, Fachbereich Informatik der Technischen Universität Darmstadt. In German.

Bostick, D. L., Shen, M. & Vaisman, I. I. (2004), 'A simple topological representation of protein structure: Implications for new, fast, and robust structural classification', *PROTEINS: Structure, Function, and Bioinformatics* **56**, 487–501.

Brenner, S. E., Chothia, C., Hubbard, T. J. P. & Murzin, A. G. (1996), [37] understanding protein structure: Using scop for fold interpretation, *in* R. F. Doolittle, ed., 'Methods in Enzymology', Vol. 266, Academic Press, pp. 635–643.

Callaway, J., Cummings, M., Deroski, B., Esposito, P., Forman, A., Langdon, P., Libeson, M., McCarthy, J., Sikora, J., Xue, D., Abola, E., Bernstein, F., Manning, N., Shea, R., Stampf, D. & Sussman, J. (1996), 'Pdb format description version 2.2', Website.
URL: *http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2_frame.html*

Chandonia, J.-M., Hon, G., Walker, N. S., Lo Conte, L., Koehl, P., Levitt, M. & Brenner, S. E. (2004), 'The astral compendium in 2004', *Nucleic Acids Research* **32**(Database Issue), 189–192.

Chandonia, J.-M., Walker, N. S., Lo Conte, L., Koehl, P., Levitt, M. & Brenner, S. E. (2002), 'Astral compendium enhancements', *Nucleic Acids Research* **30**(1), 260–263.

Chung, S. Y. & Wooley, J. C. (2003), *Bioinformatics: Managing Scientific Data*, Morgan Kaufmann Publishers, chapter 2, pp. 11–34.

Conte, L. L., Brenner, S. E., Hubbard, T. J. P., Chothia, C. & Murzin, A. (2002), 'Scop database in 2002: refinements accomodate structural genomics', *Nucleic Acids Research* **30**(1), 264–267.

Date, C. J. (2003), *An Introduction to Database Systems*, Addison-Wesley Longman Publishing Co., Inc.

Deshpande, N., Addess, K. J., Bluhm, W. F., Merino-Ott, J. C., Townsend-Merino, W., Zhang, Q., Knezevich, C., Xie, L., Chen, L., Feng, Z., Green, R. K., Flippen-Anderson, J. L., Westbrook, J., Berman, H. M. & Bourne, P. E. (2005), 'The rcsb protein data bank: a redesigned query system and relational database based on the mmcif schema', *Nucleic Acids Research* **33**(D), 233–237.

Eckman, B. A., Kosky, A. S. & Leonardo A. Laroco, J. (2001), 'Extending traditional query-based integration approaches for functional characterization of post-genomic data', *Bioinformatics* **17**(7), 587–601.

Fahl, G., Risch, T. & Sköld, M. (1993), Amos - an architecture for active mediators., *in* 'NGITS', Haifa, Israel, pp. 47–53.

Galperin, M. Y. (2005), 'The molecular biology database collection: 2005 update', *Nucleic Acids Research* **33**(Database issue), D5–D24.

Harlow, T. J., Gogarten, J. P. & Ragan, M. A. (2004), 'A hybrid clustering approach to recognition of protein families in 114 microbial genomes', *BMC Bioinformatics* **5**.

Lacroix, Z. & Critchlow, T. (2003), *Bioinformatics: Managing Scientific Data*, Morgan Kaufmann Publishers, chapter 1, pp. 1–10.

Liu, X., Fan, K. & Wang, W. (2004), 'The number of protein folds and their distribution over families in nature', *PROTEINS: Structure, Function, and Bioinformatics* **54**, 491–499.

MS Analog Software kb (2003), *ruleCore 1.0 Users Guide v. 0.1.3*, www.rulecore.com.
URL: *http://www.rulecore.com/support/documents/usersguide.pdf*

Murzin, A. G., Brenner, S. E., Hubbard, T. & Chothia, C. (1995), 'Scop: A structural classification of proteins database for the investigation of sequences and structures', *Journal of Molecular Biology* **247**, 536–540.

Murzin, A. G., Chandonia, J.-M., Andreeva, A., Howorth, D., Conte, L. L., Ailey, B. G., Brenner, S. E., Hubbard, T. J. P. & Chothia., C. (2005), 'Scop release notes', Website. Accessed 7 August 2005.
URL: *http://scop.mrc-lmb.cam.ac.uk/scop/release-notes.html*

Papamarkos, G., Poulovassilis, A. & Wood, P. T. (2003), Event-condition-action rule languages for the semantic web, *in* 'First International Workshop on Semantic Web and Databases (SWDB'03)'.

RCSB (2004*a*), Pdb annual report 2004, Technical report, Research Collaboratory for Structural Bioinformatics.

RCSB (2004*b*), 'Tutorial - searching the pdb archive', Website. Accessed 4 May 2005.
URL: *http://www.rcsb.org/pdb/query_tut.html*

Seiriö, M. & Berndtsson, M. (2005), Design and implementation of an eca rule markup language, *in* '4th International Conference on Rules and Rule Markup Languages for the Semantic Web', Galway, Ireland.

Selvam, R. A. & Sasidharan, R. (2004), 'Domins: a web resource for domain insertions in known protein structures', *Nucleic Acids Research* **32**(Database Issue), 193–195.

A PDB Event Connector

```
#!/usr/bin/perl

# A simple Perl script for downloading the files
# from any given Protein Data Bank update date.
#
# Extended to generate events and send them to
# ruleCore(TM)'s Socket Event Input Server

#################
# main program #
#################

# The URL of the Protein Data Bank FTP archive
$pdbFtpUrl = "ftp://ftp.rcsb.org/pub/pdb/";

# Optional feature: set ftp url with second command line
   argument
$pdbFtpUrl = $ARGV[1] if $ARGV[1];

# Print usage info and quit if no command line argument was
   given
if ( ! $ARGV[0] ) {
        &usage;
        exit(0);
}

# set result variables to 0
$obsolete = 0;
$modified = 0;
$notfound = "REVDAT not found ";
$files = 0;
$sendevent = 0;
$obswarning = 0;
$data = 0;

# Read command line argument
$updateDate = $ARGV[0];

# Print usage info and quit if command line argument was not
# 'dates', 'latest', or an 8-digit date or 'since'<8-digit-
   date>
if ( $updateDate !~ /^\d{8}$/
        and $updateDate !~ /^latest$/i
        and $updateDate !~ /^dates$/i
    and $updateDate !~ /^since\d{8}$/) {
        &usage;
        exit(0);
}
```

```
# Test whether we have wget
eval{ system("wget -q -O wgettest.temp http://www.rcsb.org/
   pdb/index.html"); };
if ( ! $@ and -e "wgettest.temp" ) {
        # We have a working wget
        print STDERR "Using wget for file downloads\n";
        $haveLWP = 0;
} else {
        # We do not have wget
        print STDERR "This program requires wget\n";
        exit(0);
}

# Get the ls-lR file (list of FTP archive content)
&get_lslR;
# Parse the ls-lF file
&parse_lslR;

# Choose between the four valid program modes
if ( $updateDate =~ /^latest$/i ) {
        $updateDate = $latest;
        print STDERR "Latest update date is $latest\n";
    # Get the PDB files
    &getFiles;

    #Generate and send Events
    &makeEvents;
}
elsif ( $updateDate =~ /^dates$/i ) {
        &printUpdateDates;
        exit(0);
}
elsif ($updateDate =~ /^since(\d{8})/){
    # call since to process all update since the given date
    &since($1);
}
elsif ( ! exists $dates{$updateDate} ) {
        print STDERR "$updateDate is not a valid update date
           \n";
        &printUpdateDates;
        exit(0);
} else {
    # Get the PDB files
    &getFiles;

    #Generate and send Events
    &makeEvents;
}

print "finished, obsolete events sent: $obsolete\n
    modified events sent: $modified\n
```

```
    number of files downloaded: $files\n
    sendevent called $sendevent times\n
    obsolete warning: $obswarning \n
    $notfound\n
    amount of data downloaded: $data\n";


exit(0);


###############
# subroutines #
###############

# To sort numerically, instead of alphabetically
sub numerically { $a <=> $b }

# Get the ls-lR file (list of FTP archive content)
# and save it to local disk
sub get_lslR {
        print STDERR "Getting ls-lR file (content list of
           FTP archive)\n";
        $lslR_Url = $pdbFtpUrl . "ls-lR";
        eval{ &download($lslR_Url, "ls-lR"); };
    if ( $@ or ! -e "ls-lR" ) {
                print STDERR "Could not download or save ls-
                   lR file\n";
                print STDERR "Trying one more time: \n";
                eval{ &download($lslR_Url, "ls-lR"); };
                if ( $@ or ! -e "$saveAs" ) {
                        print STDERR "Sorry, still could not
                            get it. Please try
again later.\n";
                        exit(0);
                }
                else {
                        print STDERR "Ok, got it this time!\
                           n";
            $data += -s "$saveAs";
                }
        } else {
        $data += -s "$saveAs";
    }
}

# Parse ls-lR file
# Survey content of the following FTP directories:
#   data/status/<yyyymmdd>
#   structures/{divided|obsolete}/{pdb|structure_factors|
   nmr_restraints}
#   structures/models/{current|obsolete}/pdb
```

```
# Determine all update dates and the latest update date
#
sub parse_lslR {
        print STDERR "Parsing ls-lR file\n";
        open (IN, "<ls-lR");
        while (<IN>) {
        $line = $_;
                chomp $line;
                # update dates
                if ( $line =~ /data\/status\/(\d{8})/ ) {
                        $date = $1;
                        $latest = $date if ( $date > $latest
                            );
                        $dates{$date} = 1;
                }
                # contents of each update
                elsif ( $date and $line =~ /^-r.*\s(\S+)$/ )
                    {
                        $status{$date}{$1} = 1;
                }
                # experimental structures
                elsif ( $line =~ /structures\/(divided|
                   obsolete)\/(pdb|structure_factors|
                   nmr_restraints)\/(..):/ ) {
                        $level1 = $1;
                        $level2 = $2;
                        $level3 = $3;
                }
                # experimental structures
                elsif ( $level1 and $line =~ /^-r.*\s(\S+)$
                   / ) {
                        $ftpContent{$level1}{$level2}{
                           $level3}{$1} = 1;
                }
                # theoretical models
                elsif ( $line =~ /structures\/models\/(
                   current|obsolete)\/(pdb)\/(..):/ ) {
                        $mlevel1 = $1;
                        $mlevel2 = $2;
                        $mlevel3 = $3;
                }
                # theoretical models
                elsif ( $mlevel1 and $line =~ /^-r.*\s(\S+)$
                   / ) {
                        $modelContent{$mlevel1}{$mlevel2}{
                           $mlevel3}{$1} = 1;
                }
                # reset
                elsif ( $line !~ /\S/ ) {
                        $level1 = $level2 = $level3 = 0;
                        $mlevel1 = $mlevel2 = $mlevel3 = 0;
```

```
                        $date = 0;
                }
        }
        close IN;
}

# Print all update dates
sub printUpdateDates {
        print STDERR "Valid update dates are:\n";
        foreach ( reverse sort numerically keys %dates ) {
                $n++;
                print STDOUT "$_";
                $delimeter = ( $n%8 ? " " : "\n" );
                print STDOUT $delimeter;
        }
}

# Get the PDB file lists
# (added.pdb etc.)
sub getFiles {

        # Create the update date directory if necessary
        if ( ! -e "$updateDate" ) {
                eval{ mkdir "$updateDate", 0777; };
                if ( $@ ) {
                        print STDERR "Could not make
                           directory $updateDate\n";
                        return;
                }

        }

        # The following are the files in each data/status/<
           yyyymmdd> directory
        # which in turn list all the PDB files associated
           with that update
    # (here only modified and obsolete)
        @fileLists = ("modified.pdb", "obsolete.pdb" );

        print STDERR "Starting file downloads\n";

        foreach $fileList ( @fileLists ) {

                # Was the fileList (e.g. models_added.pdb)
                   included
                # in the status/<yyyymmdd> directory for the
                    update date?
                if ( exists $status{$updateDate}{$fileList
                   } ) {
                        print STDERR "Getting $fileList\n";
                }
```

```
                else {
                        print STDERR "No $fileList file for
                           update of $updateDate\n";
                        next;
                }


                # Get the file list from FTP and save to
                   local disk
                $fileListUrl = $pdbFtpUrl . "data/status/
                   $updateDate/$fileList";
                $saveAs = "$updateDate" . "/" . "$fileList";
        if (! -e $saveAs) {
            eval{ &download($fileListUrl, $saveAs); };
        }

        # Provided the file list was successfully downloaded
            and saved,
        # get each file listed and save to local disk
        if ( $@ or ! -e $saveAs ) {
            print STDERR "Could not download or save file
               list $updateDate/$fileList\n";
            print STDERR "Trying one more time: ";
            eval{ &download($fileListUrl, $saveAs); };
            if ( $@ or ! -e "$saveAs" ) {
                print STDERR "Sorry, still could not get it
                   . ";
                print STDERR "The download will be missing
                   the files from this list.\n";
            }
            else {
                print STDERR "Ok, got it this time!\n";
                $data += -s "$saveAs";
            }
        }
        else {
            $data += -s "$saveAs";
            open (IN, "$saveAs");
            while (<IN>) {
                chomp;
                if ( /^-r.*\s(\S+)$/ ) {
                    $fileName = $1;
                    &getFile($fileList,$fileName);
                }
            }
            close IN;
        }
    }
}

# Attempt to get an individual PDB file
```

```
# (coordinates, structure factors, or nmr restraints)
sub getFile($$) {
    $file++;
    my $fileList = shift;
    my $fileName = shift;

    # Some files are listed without the '.Z' extension
    # (e.g. in added.pdb), even though all files in the
    # FTP archive have the '.Z' extention
    if ( $fileName !~ /\.Z$/ ) {
        $fileName .= ".Z";
    }


    $fileType = pdb;

    # Determine the hash directory
    $fileName =~ /^pdb.(..).\.ent/;
    $hashDir = $1;

    # Set the directory under which the file will be saved
       on local disk
    $dirName = $updateDate;
    $fileList =~ /^(\S+)\./;
    $subDirName = $1;

    # Assemble FTP download URL
    #
    # Was the file listed under the corresponding divided (
       current) directory?
    if ( exists $ftpContent{'divided'}{$fileType}{$hashDir}{
       $fileName} ) {
        $fileUrl = $pdbFtpUrl . "data/structures/divided/
           $fileType/$hashDir/$fileName";
    }
    # Was the file listed under the corresponding obsolete
       directory?
    elsif ( exists $ftpContent{'obsolete'}{$fileType}{
       $hashDir}{$fileName} ) {
        $fileUrl = $pdbFtpUrl . "data/structures/obsolete/
           $fileType/$hashDir/$fileName";
        $subDirName = "obsolete";
        if ( $fileList !~ /obsolete/ ) {
            print STDERR "WARNING: file $fileName from list
               $fileList";
            print STDERR " has been obsoleted since
               $updateDate\n";
            $obswarning++;
        }
    }
    #~ # Current model?
```

```
#~ elsif ( exists $modelContent{'current'}{$fileType}{
   $hashDir}{$fileName} ) {
    #~ $fileUrl = $pdbFtpUrl . "data/structures/models/
       current/$fileType/$hashDir/$fileName";
    #~ $subDirName = "models_" . $subDirName unless (
       $subDirName =~ /models/ );
    #~ if ( $fileList !~ /models/ ) {
        #~ print STDERR "WARNING: file $fileName from
           list $fileList";
        #~ print STDERR " is a theoretical model\n";
    #~ }
#~ }
#~ # Obsolete model?
#~ elsif ( exists $modelContent{'obsolete'}{$fileType}{
   $hashDir}{$fileName} ) {
    #~ $fileUrl = $pdbFtpUrl . "data/structures/models/
       obsolete/$fileType/$hashDir/$fileName";
    #~ $subDirName = "models_obsolete";
    #~ if ( $fileList !~ /models/ ) {
        #~ print STDERR "WARNING: file $fileName from
           list $fileList";
        #~ print STDERR " is a theoretical model\n";
    #~ }
    #~ if ( $fileList !~ /obsolete/ ) {
        #~ print STDERR "WARNING: file $fileName from
           list $fileList";
        #~ print STDERR " has been obsoleted since
           $updateDate\n";
    #~ }
#~ }
else {
    print STDERR "ERROR: Could not find file $fileName
       from list $fileList";
    return;
}

# Set the name under which the file will be saved on
   local disk
$saveAs = "$dirName/$subDirName/$fileName";

# Create sub directory (modified, obsolete) if necessary
if ( ! -e "$dirName/$subDirName" ) {
    eval{ mkdir "$dirName/$subDirName", 0777; };
    if ( $@ ) {
        print STDERR "Could not make directory $dirName/
           $subDirName\n";
        return;
    }
}

# Get file from FTP and save to local disk
```

```
    print STDERR "$fileUrl -> $saveAs";
    $unzipped = $saveAs;
    chop $unzipped;
    chop $unzipped;

    if (! -e $unzipped) {
        # download only if neither unzipped nor zipped is
           already there
        if(! -e $saveAs) {
            print STDERR " - downloading";
            eval{ &download($fileUrl, $saveAs); };
            if ( $@ or ! -e $saveAs ) {
                print STDERR "Could not download or save
                   file $fileName from list $updateDate/
                   $fileList\n";
                print STDERR "Trying one more time: ";
                eval{ &download($fileUrl, $saveAs);};
                if ( $@ or ! -e "$saveAs" ) {
                    print STDERR "Sorry, still could not get
                        it.\n";
                    print STDERR "The download will be
                       missing this file.\n";
                }
                else {
                    print STDERR "Ok, got it this time!\n";
                    $data += -s "$saveAs";
                }
            } else {
                $data += -s "$saveAs";
            }
        } else {
            print STDERR " - zipped already there, unzipping
               \n";
            sleep 1;
        }
        `gzip -d "$saveAs" `;
    } else {
        print STDERR " - unzipped already there\n";
        sleep 1;
    }
}

# Download a file and save to local disk
# Uses  wget.
# Program would have exited earlier if not available.
sub download($$) {
        my $url = shift;
        my $saveAs = shift;

    system("wget -O $saveAs $url");
```

```
}

# Print usage information
sub usage {
        print STDERR "Usage: pdb_download.pl latest\n";
        print STDERR "   or: pdb_download.pl dates\n";
        print STDERR "   or: pdb_download.pl <yyyymmdd>\n";
    print STDERR "   or: pdb_download.pl since<yyyymmdd>\n";
        print STDERR "   eg: getPdbUpdate.pl 20020603\n";
}

# Generate events to send to rulecore
sub makeEvents{
    @fileLists = ("modified.pdb", "obsolete.pdb");

    # Generating events from file lists
    foreach $fileList ( @fileLists ) {

        print STDERR "\nGenerating events from $updateDate/
           $fileList \n";

        # set event paramenter status
        if ($fileList =~ /modified/) {
            $status = "modified";
        }elsif ($fileList =~ /obsolete/) {
            $status = "obsolete";
        }else{
            print STDERR "unknown status in $fileList";
        }

        open(PDB,"$updateDate/$fileList");
        @lines = <PDB>;
        close(PDB);
        $i = 0;
        foreach (@lines) {
            $i++;
            if (/pdb(\w{4}).ent/) {
                # set event parameter pdbid
                $pdbID = "$1";
                $repdbID = uc "$pdbID";

                open(ENTRIES,"$updateDate/$status/pdb${pdbID
                   }.ent");
                @entries = <ENTRIES>;
                close(ENTRIES);

                if ($status eq "modified"){
                    $modID = 10;
                    foreach $entry  ( @entries ) {
                        if ($entry =~ /^REVDAT (.{3})(.{2})
                            (.{9}) .{5} {3}(\d) {7}((\w[\w
```

```
                        | ?]{6})*) *(($repdbID).{0,6})? *
                        $/ && $modID == 10){
                         # set event parameter modID
                         $modID = $4;

                         # set event parameter modCom
                         $modCom = $5;
                    }
                }

                if ($modID == 10) {
                    print STDERR "REVDAT not found in
                       $file File for pdb entry $pdbID\n
                       ";
                    $notfound .= "$pdbID ";
                } else {
                    &sendEvent($status,$pdbID,$modID,
                       $modCom,"na");
                }

            }elsif ($status eq "obsolete"){
                foreach $entry  (@entries) {
                    if ($entry =~ /^OBSLTE {2}(.{2})
                        .{9} .{4} {6}((\w{4} ?)+) *((
                       $repdbID).{0,6})? *$/){
                         # set event parameter subst
                         $subst = $2;
                    }
                }
                if (!$subst) {
                    print STDERR "OBSLTE not found in
                       $file File for pdb entry $pdbID\n
                       ";
                    $subst = "unknown";
                }
                &sendEvent($status,$pdbID, "na","na" ,
                   $subst);

            }
        } else {
            print STDERR "no pdb id found in line $i \n"
               ;
        }
    }
    if ($i == 0) {
        print STDERR "no entries in $fileList \n";
    }
  }
}

#send event through socket on port 8532
```

```
sub sendEvent{
    $sendevent++;
    my $status = shift;
    $status .= "PDB";
    my $pdbid = shift;
    my $modID = shift;
    my $modCom = shift;
    my $subst = shift;

    #invoke socket
    use Socket;
    my $host = 'localhost';
    my $port = 8532;
    my $proto = getprotobyname('tcp');
    my $iaddr = inet_aton($host);
    my $paddr = sockaddr_in($port, $iaddr);
    socket (SOCKET, PF_INET, SOCK_STREAM, $proto) or die "
       Error: $!\n";
    connect(SOCKET, $paddr) or die "Socket connect error: $
       ! \n";

    #generate and send event string
    my $updatedate_length = 8;
    my $status_length = length ($status);
    my $pdbid_length = 4;
    my $modID_length = 1;
    my $modCom_length = length ($modCom);
    my $subst_length = 4;

    # sending actual event string
    if ($status eq "modifiedPDB") {
        print STDERR "Sending event string: $status_length,
           type:$status$updatedate_length,param,DATE:
           $updateDate$pdbid_length,param,PDBID:
           $pdbid$modID_length,param,MODID:
           $modID$modCom_length,param,MODCOM:$modCom\n";
        print SOCKET "$status_length,type:
           $status$updatedate_length,param,DATE:
           $updateDate$pdbid_length,param,PDBID:
           $pdbid$modID_length,param,MODID:
           $modID$modCom_length,param,MODCOM:$modCom";
        $modified++;
    }
    elsif ($status eq "obsoletePDB") {
        print STDERR "Sending event string: $status_length,
           type:$status$updatedate_length,param,DATE:
           $updateDate$pdbid_length,param,PDBID:
           $pdbid$subst_length,param,SUBST:$subst\n";
        print SOCKET "$status_length,type:
           $status$updatedate_length,param,DATE:
           $updateDate$pdbid_length,param,PDBID:
```

```
            $pdbid$subst_length,param,SUBST:$subst";
        $obsolete++;
    }
    else {
        print STDERR "Status not valid, no event string send
            .\n";
    }
    close SOCKET or die "close: $! \n";
}

# process all updates somce given date
sub since ( $ ) {
    my $since = shift;
    print STDERR "processing all updates since $since";
    foreach ( sort numerically keys %dates ) {
        if ( $_ > $since) {
            $updateDate = $_;
            print STDERR "\nFOR UPDATE DATE $_\n";
            # Get the PDB files
            &getFiles;

            #Generate and send Events
            &makeEvents;
        }
    }
}
```

B ECA Rules in `rcml` Format

```
<?xml version='1.0' encoding='UTF-8'?>
<root>
  <config>
    <engine-cfg>
      <logging>
        <directory>C:/Program Files/ruleCore/logfiles/</
           directory>
        <size>500000</size>
        <level>Debug 1</level>
      </logging>
      <monitor>
        <port>4455</port>
      </monitor>
      <debug-mode>on</debug-mode>
    </engine-cfg>
    <event-inputs/>
    <rules>
      <rule create='init' delivery='all' limit='None'
         parameter='append' name='obsPDB'>
        <description>Handles obsolete entries in PDB and
           calls update script for SCOP.</description>
        <event-ref enabled='yes'>obsPDB</event-ref>
```

```
    <condition-ref enabled='no'/>
    <action-ref enabled='yes'>SCOPobsPDB</action-ref>
    <minus-action-ref enabled='no'/>
    <instance-limit/>
    <filter/>
  </rule>
  <rule create='init' delivery='all' limit='None'
     parameter='append' name='modPDB'>
    <description>Handles modified entries in PDB and
       calls update script for SCOP.</description>
    <event-ref enabled='yes'>modPDB</event-ref>
    <condition-ref enabled='yes'>MODID</condition-ref>
    <action-ref enabled='yes'>SCOPmodPDB</action-ref>
    <minus-action-ref enabled='no'/>
    <instance-limit/>
    <filter/>
  </rule>
  <rule create='init' delivery='all' limit='None'
     parameter='append' name='obsSCOPLeaf'>
    <description>Handles obsolete leaf nodes in SCOP and
        calls update script for SCOP.</description>
    <event-ref enabled='yes'>obsLeaf</event-ref>
    <condition-ref enabled='no'/>
    <action-ref enabled='yes'>SCOPobsNode</action-ref>
    <minus-action-ref enabled='no'/>
    <instance-limit/>
    <filter/>
  </rule>
  <rule create='init' delivery='all' limit='None'
     parameter='append' name='obsSCOPNode'>
    <description>Handles obsolete non-leaf nodes in SCOP
        and calls update script for SCOP.</description>
    <event-ref enabled='yes'>obsNode</event-ref>
    <condition-ref enabled='yes'>CHILDNODES</condition-
       ref>
    <action-ref enabled='yes'>SCOPobsNode</action-ref>
    <minus-action-ref enabled='no'/>
    <instance-limit/>
    <filter/>
  </rule>
</rules>
<event-defs>
  <event-def type='basic' name='obsoletePDB'>
    <parameters>
      <parameter type='string' name='DATE'/>
      <parameter type='string' name='PDBID'/>
      <parameter type='string' name='SUBST'/>
    </parameters>
  </event-def>
  <event-def type='basic' name='modifiedPDB'>
    <parameters>
```

```
    <parameter type='string' name='DATE'/>
    <parameter type='string' name='PDBID'/>
    <parameter type='string' name='MODID'/>
    <parameter type='string' name='MODCOM'/>
  </parameters>
</event-def>
<event-def type='basic' name='obsoleteNode'>
  <parameters>
    <parameter type='string' name='DATE'/>
    <parameter type='string' name='NODE'/>
    <parameter type='string' name='PARENT'/>
    <parameter type='string' name='CHILDNODES'/>
    <parameter type='string' name='OBSCHILDNODES'/>
  </parameters>
</event-def>
<event-def type='basic' name='obsoleteLeaf'>
  <parameters>
    <parameter type='string' name='DATE'/>
    <parameter type='string' name='NODE'/>
    <parameter type='string' name='PARENT'/>
  </parameters>
</event-def>
<event-def type='composite' name='obsPDB'>
  <detect-event/>
  <no-detect-event/>
  <event-selector name=''>
    <condition/>
    <scope>global</scope>
    <events/>
  </event-selector>
  <detector>
    <event-ref alias='obsoletePDB' type='alias'>
       obsoletePDB</event-ref>
  </detector>
</event-def>
<event-def type='composite' name='modPDB'>
  <detect-event/>
  <no-detect-event/>
  <event-selector name=''>
    <condition/>
    <scope>global</scope>
    <events/>
  </event-selector>
  <detector>
    <event-ref alias='modifiedPDB' type='alias'>
       modifiedPDB</event-ref>
  </detector>
</event-def>
<event-def type='composite' name='obsNode'>
  <detect-event/>
  <no-detect-event/>
```

```
    <event-selector name=''>
      <condition/>
      <scope>global</scope>
      <events/>
    </event-selector>
    <detector>
      <event-ref type='event'>obsoleteNode</event-ref>
    </detector>
  </event-def>
  <event-def type='composite' name='obsLeaf'>
    <detect-event/>
    <no-detect-event/>
    <event-selector name=''>
      <condition/>
      <scope>global</scope>
      <events/>
    </event-selector>
    <detector>
      <event-ref type='event'>obsoleteLeaf</event-ref>
    </detector>
  </event-def>
</event-defs>
<condition-defs>
  <condition-def always-true='no' composite='yes' name='
     MODID'>
    <parameters>
      <parameter name='modtype'>
        <item-ref type='event'>
          <event-ref alias='modifiedPDB' type='alias'>
             modifiedPDB</event-ref>
        </item-ref>
        <param-ref>MODID</param-ref>
        <instance>all</instance>
        <function/>
        <key>no</key>
      </parameter>
    </parameters>
    <expressions>
      <expression name='modtype1'>
        <lhs>
          <param-ref>modtype</param-ref>
        </lhs>
        <operator>not equal</operator>
        <rhs>
          <value>1</value>
        </rhs>
      </expression>
      <expression name='modtype0'>
        <lhs>
          <param-ref>modtype</param-ref>
        </lhs>
```

```
        <operator>not equal</operator>
        <rhs>
          <value>0</value>
        </rhs>
      </expression>
    </expressions>
    <composite-condition>
      <and>
        <condition-ref>modtype1</condition-ref>
        <condition-ref>modtype0</condition-ref>
      </and>
    </composite-condition>
  </condition-def>
  <condition-def always-true='no' composite='no' name='
     CHILDNODES'>
    <parameters>
      <parameter name='CHILDNODES'>
        <item-ref type='event'>
          <event-ref type='event'>obsoleteNode</event-
             ref>
        </item-ref>
        <param-ref>CHILDNODES</param-ref>
        <instance>all</instance>
        <function/>
        <key>no</key>
      </parameter>
      <parameter name='OBSCHILDNODES'>
        <item-ref type='event'>
          <event-ref type='event'>obsoleteNode</event-
             ref>
        </item-ref>
        <param-ref>OBSCHILDNODES</param-ref>
        <instance>all</instance>
        <function/>
        <key>no</key>
      </parameter>
    </parameters>
    <expressions>
      <expression name='CHILDNODES'>
        <lhs>
          <param-ref>CHILDNODES</param-ref>
        </lhs>
        <operator>equal</operator>
        <rhs>
          <param-ref>OBSCHILDNODES</param-ref>
        </rhs>
      </expression>
    </expressions>
    <composite-condition/>
  </condition-def>
</condition-defs>
```

```
    <action-defs>
      <action-def name='SCOPobsNode'>
        <action-item type='script' name='SCOPobsNode'>perl
           SCOP_obsoleteNode.pl</action-item>
      </action-def>
      <action-def name='SCOPmodPDB'>
        <action-item type='script' name='SCOPmodPDB'>perl
           SCOP_modifiedPDB.pl</action-item>
      </action-def>
      <action-def name='SCOPobsPDB'>
        <action-item type='script' name='SCOPobsPDB'>perl
           SCOP_obsoletePDB.pl</action-item>
      </action-def>
    </action-defs>
    <global-filter-defs/>
    <states/>
  </config>
</root>
```

C SCOP Wrappers

C.1 Modified Entry in PDB

```
#!/usr/bin/perl
#
# a script to be executed when a pdb entry is modified
#

#~ # Parse xml input
while (<>) {
    if (/<event type='(.+)' name/) {
        $TYPE = $1;
    }
    if (/'DATE'>(.+)</) {
        $DATE = $1;
    }
    if (/'FILE'>(.+)</) {
        $FILE = $1;
    }
    if (/'PDBID'>(.+)</) {
        $PDBID = $1;
    }
    if (/'MODID'>(.+)</) {
        $MODID = $1;
    }
    if (/'MODCOM'>(.+)</) {
        $MODCOM= $1;
    }
}
```

```
#set modification type
if ($MODID == 2)  {
    $modtype = "CONECT record";
} elsif ($MODID == 3) {
        $modtype = "coordinates or transformations";
} else {
    $modtype = "unknown";
}

# find PDB IDs in dir.cla.scop.txt_1.69 and set
   corresponding sundids
open(CLA,"SCOP/dir.cla.scop.txt_1.69")|| die ("$PDBID Could
   not open file. $!");
@cla_scop = <CLA>;
close(CLA);
$i = 1;
foreach (@cla_scop) {
    if (/^\w{7}\t($PDBID).+px=(\d{5,6}).*$/) {
        $sunid{$i} = $2;
        #print STDERR "sunid$i zu PDB ID $PDBID: $sunid{$i}\
           n";
        $i +=1;
    }
}

if ($i > 1) {
    # find sundids in dir.com.scop.txt_1.69

    use Fcntl qw(:DEFAULT :flock);
    sysopen(COM, "SCOP/dir.com.scop.txt_1.69", O_RDWR |
       O_CREAT) or die "$PDBID can't open dir.com.scop.txt_1
       .69: $!";

    #locking dir.com.scop.txt_1.69 to read and write
    flock(COM, LOCK_EX) or die "$PDBID can't write-lock dir.
       com.scop.txt_1.69: $!";
    $prPDBID = uc "$PDBID";
    print STDERR "$prPDBID-LOCKED \n";
    @com_scop = <COM>;

    foreach $com_scop (@com_scop) {
        for ($j = 1; $j <  $i; ++$j) {
            # add comment to respetive line
            if ($com_scop=~ /^($sunid{$j})( ! .*)*\n/) {
                chop $com_scop;
                $com_scop .= " ! PDB ID $PDBID modified in
                   PDB update $DATE (modification type
                   $MODID - $modtype) $MODCOM \n";
                print STDERR "$PDBID added: $com_scop";
                $sunid{$j}=done;
            }
```

```
        }
        push (@com_scop_new, $com_scop);
    }

    #add new comment line if no comment for sunid present
    for ($j = 1; $j <  $i; ++$j) {
        if ($sunid{$j} ne "done" ) {
            push (@com_scop_new, "$sunid{$j} ! PDB ID $PDBID
                modified in PDB update $DATE (modification
               type $MODID - $modtype) $MODCOM \n");
            print STDERR "$PDBID new: $sunid{$j} ! PDB ID
               $PDBID modified in PDB update $DATE (
               modification type $MODID - $modtype) $MODCOM
                \n";
        }
    }

    #write new dir.com.scop.txt_1.69
    seek (COM,0,0);
    truncate(COM, (tell COM)) or die "$PDBID can't truncate
       dir.com.scop.txt_1.69: $!";
    print COM @com_scop_new;
    print STDERR "$prPDBID-WRITTEN NOW UNLOCKING\n";
    # closing and unlocking dir.com.scop.txt_1.69
    close(COM) or die "$PDBID can't close dir.com.scop.txt_1
       .69: $!";

    #writing to logfile
    open (LIST, ">>SCOP/modPDB.log") || die ("$PDBID Could
       not open file. $!");
    print LIST "$TYPE\t$DATE\t$PDBID\t$SUBST\n";
    close(LIST);
} else {
    print STDERR "$PDBID not found in dir.cla.scop.txt_1
       .69.\n";
    #writing to logfile
    open (LIST, ">>SCOP/modPDB.log") || die ("$PDBID Could
       not open file. $!");
    print LIST "$TYPE\t$DATE\t$PDBID\t$SUBST - not found in
       dir.cla.scop.txt_1.69\n";
    close(LIST);
}
exit(0);
```

C.2 Obsolete Entry in PDB

```
#!/usr/bin/perl
#
# a script to be executed when a pdb entry is obsolete
#

#~ ################
```

```
#~ # main program #
#~ ################

# Parse xml input
while (<>) {
    if (/<event type='(.+)' name/) {
        $TYPE = $1;
    }
    if (/'DATE'>(.+)</) {
        $DATE = $1;
    }
    if (/'PDBID'>(.+)</) {
        $PDBID = $1;
    }
    if (/'SUBST'>(.+)</) {
        $SUBST = $1;
    }
}

# find PDB ID in dir.cla.scop.txt_1.69 and set corresponding
    sunids
open(CLA,"SCOP/dir.cla.scop.txt_1.69")|| die ("$PDBID Could
   not open file. $!");
@cla_scop = <CLA>;
close(CLA);
$i = 1;
foreach (@cla_scop) {
    if (/^\w{7}\t($PDBID).+sp=(\d{5,6}).*px=(\d{5,6}).*$/) {
        $sunid{$i} = $3;
        $sp{$i} =$2;
        $i += 1;
    }
}

if ($i > 1) {
    # find sunids in dir.com.scop.txt_1.69

    use Fcntl qw(:DEFAULT :flock);
    sysopen(COM, "SCOP/dir.com.scop.txt_1.69", O_RDWR |
       O_CREAT) or die "$PDBID can't open dir.com.scop.txt_1
       .69: $!";

    #locking dir.com.scop.txt_1.69 to read and write
    flock(COM, LOCK_EX) or die "$PDBID can't write-lock dir.
       com.scop.txt_1.69: $!";
    $prPDBID = uc "$PDBID";
    print STDERR "$prPDBID-LOCKED \n";
    @com_scop = <COM>;
    foreach $com_scop (@com_scop) {
        for ($j = 1; $j <  $i; ++$j) {
            # add comment to respective line
```

```
        if ($com_scop=~ /^($sunid{$j})( ! .*)*\n/) {
            chop $com_scop;
            $com_scop .= " ! PDB ID $PDBID is obsolete
               with PDB update $DATE, replaced by PDB
               entry $SUBST\n";
            print STDERR "$PDBID added: $com_scop";
            $check{$j}=done;

        }
    }
    push (@com_scop_new, $com_scop);
}
#add new comment line if no comment for sunid present
for ($j = 1; $j <  $i; ++$j) {
    if ($check{$j} ne "done" ) {
        push (@com_scop_new, "$sunid{$j} ! PDB ID $PDBID
            is obsolete with PDB update $DATE, replaced
           by PDB entry $SUBST\n");
        print STDERR "$PDBID new: $sunid{$j} ! PDB ID
           $PDBID is obsolete with PDB update $DATE,
           replaced by PDB entry $SUBST\n";
    }
}

#write new dir.com.scop.txt_1.69
seek (COM,0,0);
truncate(COM, (tell COM)) or die "$PDBID can't truncate
   dir.com.scop.txt_1.69: $!";
print COM @com_scop_new;
print STDERR "$prPDBID-WRITTEN NOW UNLOCKING\n";
# closing and unlocking dir.com.scop.txt_1.69
close(COM) or die "$PDBID can't close dir.com.scop.txt_1
   .69: $!";

# obtain hierarchy information for sunids
open(HIE,"SCOP/dir.hie.scop.txt_1.69")|| die ("$PDBID
   Could not open file. $!");
@hie_scop = <HIE>;
close(HIE);
for ($j = 1; $j <  $i; ++$j) {
    foreach $hie_scop (@hie_scop) {
        # find line where $sunid{$j} is child
        if ($hie_scop=~ /^(\d{5,6})\t\d{5,6}\t((\d
           {5,6},)*$sunid{$j},?(\d{5,6},?)*)\n/ ){
            $sunid_par{$j} = $1;
        }
    }
    &sendEvent($sunid{$j},$sunid_par{$j});
}

# write to logfile
```

```
    open (LIST, ">>SCOP/obsPDB.log") || die ("$PDBID Could
       not open file. $!");
    print LIST "$TYPE\t$DATE\t$PDBID\t$SUBST\n";
    close(LIST);
} else {
    print STDERR "$PDBID not found in SCOP/dir.cla.scop.
       txt_1.69.\n";
    # write to logfile
    open (LIST, ">>SCOP/obsPDB.log") || die ("$PDBID Could
       not open file. $!");
    print LIST "$TYPE\t$DATE\t$PDBID\t$SUBST - not found in
       dir.cla.scop.txt_1.69\n";
    close(LIST);
}

exit(0);

###############
# subroutines #
###############

#send new event through socket on port 8532
#to check cascading obsolete
sub sendEvent($){
    my $node = shift;
    my $parent = shift;

    #invoke socket
    use Socket;
    my $host = 'localhost';
    my $port = 8532;
    my $proto = getprotobyname('tcp');
    my $iaddr = inet_aton($host);
    my $paddr = sockaddr_in($port, $iaddr);
    socket (SOCKET, PF_INET, SOCK_STREAM, $proto) or die "
       $PDBID Error: $!\n";
    connect(SOCKET, $paddr) or die "$PDBID Socket connect
       error: $! \n";

    #generate and send event string
    my $date_length = length($DATE);
    my $node_length = length($node);
    my $parent_length = length($parent);

    #sending actual event string
    print STDERR "$PDBID Sending event string: 12,type:
       obsoleteLeaf$date_length,param,DATE:$DATE$node_length
       ,param,NODE:$node$parent_length,param,PARENT:$parent\
       n";
    print SOCKET "12,type:obsoleteLeaf$date_length,param,
       DATE:$DATE$node_length,param,NODE:$node$parent_length
```

```
       ,param,PARENT:$parent";
    sleep 1;

    close SOCKET or die "$PDBID close: $! \n";
}
```

C.3 Obsolete Entry in SCOP

```
#!/usr/bin/perl
#
# a script to be executed when a scop entry is possibly
   obsolete
#

#~ #################
#~ # main program #
#~ #################

# Parse xml input
while (<>) {
    if (/<event type='(.+)' name/) {
        $TYPE = $1;
    }
    if (/'DATE'>(.+)</) {
        $DATE = $1;
    }
    if (/'NODE'>(.+)</) {
        $NODE = $1;
    }
    if (/'PARENT'>(.+)</) {
        $PARENT = $1;
    }
    if (/'CHILDNODES'>(.+)</) {
        $CHILDNODES = $1;
    }
    if (/'OBSCHILDNODES'>(.+)</) {
        $OBSCHILDNODES = $1;
    }
}


# print to logfile
open (LIST, ">>SCOP/obsSCOP.log")|| die ("$PARENT Could not
   open file. $!");
print LIST "$TYPE\t$DATE\t$NODE\t$PARENT\t$CHILDNODES\
   t$OBSCHILDNODES\n";
close(LIST);


# find sunid in dir.com.scop.txt_1.69

use Fcntl qw(:DEFAULT :flock);
```

```
sysopen(COM, "SCOP/dir.com.scop.txt_1.69", O_RDWR | O_CREAT)
    or die "$PARENT can't open dir.com.scop.txt_1.69: $!";

#locking dir.com.scop.txt_1.69 to read and write
flock(COM, LOCK_EX) or die "$PARENT can't write-lock dir.com
   .scop.txt_1.69: $!";
$prPARENT = uc "$PARENT";
print STDERR "$prPARENT - LOCKED \n";
@com_scop = <COM>;

# add comment to respective line
foreach $com_scop (@com_scop) {

    # for a non-leaf node
    if ($CHILDNODES) {
        if ($com_scop=~ /^$NODE.*\n/) {
            chop $com_scop;
            $com_scop .= " ! all $CHILDNODES child nodes
               possibly obsolete\n";
            print STDERR "$PARENT added: $com_scop";
        }
    }

    # for all nodes
    if ($com_scop=~ /^$PARENT.*\n/) {
        chop $com_scop;
        $com_scop .= " ! possibly obsolete child node: $NODE
            (due to PDB update $DATE)\n";
        $obschildnodes = 0;
        while ($com_scop =~ /obsolete child node/g) {
            $obschildnodes++;
        }
        print STDERR "$PARENT added: $com_scop";
        $check=done;
    }
    push (@com_scop_new, $com_scop);
}
#add new comment line if no comment for node present
#print STDERR "added line for $PARENT : ";
if ($check ne "done" ) {
    push (@com_scop_new, "$PARENT ! possibly obsolete child
       node: $NODE (due to PDB update $DATE)\n");
    $obschildnodes = 1;
    print STDERR "$PARENT new: $PARENT ! possibly obsolete
       child node: $NODE (due to PDB update $DATE)\n";
}

#write new dir.com.scop.txt_1.69
seek (COM,0,0);
truncate(COM, (tell COM)) or die "$PARENT can't truncate dir
   .com.scop.txt_1.69: $!";
```

```
print COM @com_scop_new;
print STDERR "$prPARENT - WRITTEN NOW UNLOCKING\n";
# closing and unlocking dir.com.scop.txt_1.69
close(COM) or die "$PARENT can't close dir.com.scop.txt_1
   .69: $!";

# obtain hierarchy information for $PARENT
open(HIE,"SCOP/dir.hie.scop.txt_1.69")|| die ("$PARENT Could
    not open file. $!");
@hie_scop = <HIE>;
close(HIE);

foreach $hie_scop (@hie_scop) {
    # find number of child nodes of $PARENT
    if ($hie_scop =~/^$PARENT.*\n/) {
        $childnodes = 1;
        while ($hie_scop =~ /,/g) {
            $childnodes++;
        }
    }
    # find line where $PARENT is child
    if  ($hie_scop=~ /^(\d{1,6})\t\d{1,6}\t((\d{5,6},)*
       $PARENT,?(\d{5,6},?)*)\n/) {
        $node_par = $1;
    }
}
&sendEvent($PARENT,$node_par,$childnodes,$obschildnodes);

exit(0);

###############
# subroutines #
###############

#send new event through socket on port 8532
#to check cascading obsolete
sub sendEvent{
    my $node = shift;
    my $parent = shift;
    my $childnodes = shift;
    my $obschildnodes = shift;

    #invoke socket
    use Socket;
    my $host = 'localhost';
    my $port = 8532;
    my $proto = getprotobyname('tcp');
    my $iaddr = inet_aton($host);
    my $paddr = sockaddr_in($port, $iaddr);
    socket (SOCKET, PF_INET, SOCK_STREAM, $proto) or die "
       $PARENT Error: $!\n";
```

```
    connect(SOCKET, $paddr) or die "$PARENT Socket connect
       error: $! \n";

    #generate event string
    my $date_length = length($DATE);
    my $node_length = length($node);
    my $parent_length = length($parent);
    my $childnodes_length = length($childnodes);
    my $obschildnodes_length = length($obschildnodes);

    # send actual event string
    print STDERR "$PARENT Sending event string: 12,type:
       obsoleteNode$date_length,param,DATE:$DATE$node_length
       ,param,NODE:$node$parent_length,param,PARENT:
       $parent$childnodes_length,param,CHILDNODES:
       $childnodes$obschildnodes_length,param,OBSCHILDNODES:
       $obschildnodes\n";
    print SOCKET "12,type:obsoleteNode$date_length,param,
       DATE:$DATE$node_length,param,NODE:$node$parent_length
       ,param,PARENT:$parent$childnodes_length,param,
       CHILDNODES:$childnodes$obschildnodes_length,param,
       OBSCHILDNODES:$obschildnodes";

    close SOCKET or die "close: $! \n";
}
```

C.4 Lookup Comment

```
#!/usr/bin/perl
#
# a script to retrieve the comment of
# a SCOP entry by its sunid
#

#read command line argument
if ($ARGV[0]) {
    $sunid = $ARGV[0];
}else {
    print STDERR "Usage information\n
        findcomment.pl <sunid>\n";
    exit(0);
}

# find sundid in dir.com.scop.txt_1.69
open (COM, "SCOP/dir.com.scop.txt_1.69");
@com_scop = <COM>;
close(COM);
foreach $com_scop (@com_scop) {
    if ($com_scop=~ /^$sunid.*$/) {
        print STDERR "$com_scop\n"
    }
}
```

```
exit(0);
```