A Guide to Merging Structured Reports Using Fusion Rules

Anthony Hunter[1] and Rupert Summerton[2]
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
www.cs.ucl.ac.uk/staff/a.hunter/frt

[1]a.hunter@cs.ucl.ac.uk
[2]r.summerton@cs.ucl.ac.uk

1.9

Contents

Chapter 1

What this guide is about

1.1 The fusion rule approach to merging structured information

This guide is designed to help you build applications to merge potentially inconsistent structured information using fusion rules. Structured information consists of single words or short phrases that are tagged to identify their meaning. Much of the information in XML format, for example, is structured information. Fusion rules are a methodology that applies logic-based techniques to merging potentially inconsistent information. They allow such information to be merged in a context-sensitive way; that is, they enable different components of a structured report to be merged in different ways, as deemed appropriate by a knowledge engineer.

Fusion rules themselves have a conditional or *if-then* form. The antecedents of these rules contain variables that correspond to the tags in the structured input reports, allowing the variables to be ground or instantiated by the textentries in the reports. These ground antecendents, or conditions, are then used to query a knowledgebase containing background knowledge about the application domain in question. If all of the conditions of a rule are found, by the lights of the knowledgebase, to be true, then the consequent of the rule is executed. The consequent of a fusion rule is a number of actions that specify how the output merged report is to be constructed.

1.2 Fusion rule software

There are two pieces of software associated with this guide. The first is a fusion tool that contains a fusion engine whose job is to merge a set of structured input reports using a set of fusion rules and a knowledgebase. The second piece of software is a rule tool that allows you to write or edit a set of fusion rules using a graphical user interface. It also conducts a number of elementary checks on a set of fusion rules to see if they have been correctly engineered.

If you have obtained and unzipped the file `fusion.zip`, we recommend that after quickly consulting Chapter 2, which deals with some preliminary matters, you use the fusion tool to run the demonstration application which concerns weather reports (a domain that is notorious for conflicting information). Details of how to do this can be found in Chapter 3. It is extremely simple to do, since it amounts only to the file management operations of loading the fusion rules, the input reports, and the knowledgebase, and then clicking on the "Merge" option. Doing this will allow you very easily to understand the basic idea of what a fusion rules application does far better than reading any explanation that we could write. (Chapter 6 contains a

user manual for the fusion tool, but it is unlikely that you will need to consult this just to run the demonstration application.) If you are interested in inspecting the background knowledge in the knowledgebase, details of this can be found in Chapter 10.

Having run the demonstration application we hope you might be tempted to develop an application for yourself. If you are, you will need to know much more about the nature of fusion rules and how they exploit background knowledge to merge structured information. The remaining chapters of this guide are intended to provide you with this information.

1.3 A note on the demonstration application

For reasons of ease of use the demonstration knowledgebase is implemented in the form of a number of Java classes; so long as you have the Java Software Development Kit (SDK), or even just the Java Runtime Environment, on your machine, you can run the demonstration using the fusion tool as-is. However, obviously no serious knowledgebase would be implemented in this way. In using the fusion tool in a number of case studies we ourselves have exploited a Prolog engine and program to build more sophisticated knowledgebases. As the Prolog engine we used is proprietary software, we cannot include it with the download package. We have, however, included the Java interface to the Prolog engine so that, should you already have, or choose to get, the Prolog engine, you can easily develop more powerful applications. Please refer to Appendix D for details. (For an explanation of how the knowledgebase fits into the architecture of a fusion rules application please refer to Section 4.3, and to the first two sections in Chapter 10.

Chapter 2

Preliminaries

2.1 Contents of `fusion.zip`

If you have obtained and unzipped the file `fusion.zip` then you should now have the following:

- This guide.
- Javadoc for the classes in package `uk.ac.ucl.cs.fusion.rule`, and for interface `AnswerQuery` and class `Response`. These can be found in the directory `docs/javadoc`, where they can best be browsed via the file `index.html`.
- Three HTML files, `about.html`, `usermanual.html`, and `rulemanual.html`, which can be found in the directory `docs/helpdocs`. These contain information that can be accessed via the "Help" menu in the fusion tool or the rule tool.
- fusion.jar, in directory `lib`. (Contains Java class files for the fusion tool and the rule tool.)
- jdom.jar and xerces.jar, again both in directory `lib`. (Contains Java API for reading and writing XML, and the XML parser)
- Third party licences for JDOM and Xerces.
- Batch files fusion.bat, rule.bat and cpappend.bat (Used to run the fusion tool, the rule tool, and to set the classpath.)
- rulefile.dtd, the DTD for XML files containing fusion rules. This is to be found in the folder `demonstration/rules`.
- weatherreport.dtd, the DTD for XML files containing weather reports for the demonstration application. This is to be found in folder `demonstration/reports`.
- The classes that are dynamically loaded and used by the fusion tool as a knowledgebase to merge the input reports in the demonstration aplication. These are to be found in the folder `dynamicClasses`.
- The rules and the input reports for a simple demonstration application using weather reports. These are to be found in, respectively, the folders `demonstration/rules` and `demonstration/reports`.

2.2 Installation

All the software and documentation comes in a zipped archive. Once you have unzipped the file no further installation is required. The software is written in Java, and uses JDOM (www.jdom.org) as the API for reading and writing XML. JDOM is not itself an XML parser, so in turn it uses Apache's Xerces as a parser. Jar files containing these are included with this release, but if you are running the software on Windows there is no need to add them to your classpath as this is done by the batch files for both the fusion tool and the rule tool.

2.3 System Requirements

The software was developed using Java 1.4.1, but should run using any version of Java 2 (i.e. Java 1.2, or later). If you do not have a version of the Java software development kit (SDK) on your machine, you can download it from the Sun website at www.java.sun.com where it is available for free.

2.4 Running the software

In Windows, simply click on the batch file `fusion` to run the fusion tool, or click on the batch file `rule` to run the rule tool. If you encounter an `out of environment space` error, you can increase the initial environment memory in the following way: (i) click on the MSDOS icon in the top left corner of the command prompt or MSDOS window; (ii) select **Properties** from the drop down menu; (iii) select the **Memory** tab from the **Properties** dialog box; (iv) in the **Initial Environment** combo box scroll down and select **4096**; (v) click **Apply** followed by **ok**.

For other operating systems you can run the software from the command line by typing:

```
java uk.ac.ucl.cs.fusion.fusionSystem.FusionMain
```

for the fusion tool, or by typing:

```
java uk.ac.ucl.cs.fusion.ruleEngineering.RuleTool
```

for the rule tool. You will also need to put fusion.jar, jdom.jar and xerces.jar in your classpath. On some machines you may need to increase the maximum size that Java allots to the heap. You can do this by typing:

```
java −mx80m uk.ac.ucl.cs.fusion.fusionSystem.FusionMain
```

This would increase the size of the heap to 80 megabytes. You can of course experiment with other values.

The fusion tool comes with a simple demonstration application consisting of ten (fictional) weather reports (five each for two different days), and two sets of rules. These can be found in the `demonstration/reports` and the `demonstration/rules` folders. There is also a simple knowledgebase. To load this you must choose the **Select Resource** menu option and in the subsequent dialog box type:

```
DemoWeatherKBase
```

You must get the spelling and the $cases$ of the characters correct, or you will get an error message.

2.5 Disclaimer

This software is provided as-is. Although more than a little effort has been put into anticipating and dealing with possible error conditions, particularly those arising from incorrectly written rules, there are still errors that have not been dealt with (some known and of course many more unknown). This software is supplied for demonstration purposes and to introduce the concept of merging using fusion rules and a knowledgebase. It is not intended for large-scale industrial applications. However, note that if the fusion tool does crash you will not have lost any data, valuable or otherwise. If the problem lies with the rules, edit them and simply run the tool again. If the problem lies with the knowledgebase, find the bug and fix it. If you are convinced the problem lies with the fusion tool or rule tool software itself, then unfortunately we do not have the resources to offer support.

2.6 Third-party acknowledgements

The fusion tool and rule tool software includes software developed by the JDOM Project (http://www.jdom.org/). The fusion tool and rule tool software includes software developed by the Apache Software Foundation (http://www.apache.org/).

Chapter 3

How to run the demonstration weather reports application

The best way to get an overview of what a fusion rules application does is to run the demonstration weather reports aplication using the fusion tool. This is extremely simple to do; just follow these steps:

Step 1 To run the fusion tool simply click on the file `fusion.bat` in the directory in which you unzipped `fusion.zip` (See figure 3.1).

Step 2 On start up, the fusion tool should look as in figure 3.2. No input reports, fusion rules, or knowledgebase have yet been loaded.

Step 3 You may load the fusion rules, input reports, and knowledgebase in any order, but we will start with the reports. Go to the "File" menu and select the first option, "Select Report(s)". As an alternative, you may click on the single folder icon on the toolbar. (See figure 3.3.)

A file chooser will now appear displaying two sets of reports, five for December 10th 2002, and five for June 28th 2004. (All are in fact fictional.) Although you can experiment and see what happens if you load different combinations, on a first run start by loading all five for either one of these two dates. If you are using Windows, you can save time by clicking on the first report, then press the "Shift" key while clicking on the last report. This will select all five reports at once. (See figure 3.4.)

The fusion tool now looks as in figure 3.5. Note that, since we are using the "Select Report(s)" option, each file is displayed separately on an individual tab.

Step 4 To load the fusion rules select the "Select Rules" option from the "File" menu. (Or click on the rules icon–the fifth–on the toolbar.) The demonstration has two sets of rules, one to deal with conflicts by, where appropriate, majority voting; the other deals with conflicts by selecting the textentry from the input report with the most preferred source. Select either set of rules. (See figure 3.6.)

When the rules have been loaded they are displayed on tabs in the right hand window. (See figure 3.7.) For reasons of persistence, fusion rules are marked up in XML, but XML can be hard to read so we have developed another notation which makes it easier to understand the form and purpose of a fusion rule. The fusion rules are displayed in both of these notational forms on seperate tabs. Don't worry for now if they make no sense to you, they will be explained in Chapter 9.

Step 5 To load the knowledgebase select the "Select Resource" option from the "File" menu. (Or click on the resource icon–the sixth–on the toolbar.) A dialog box will now appear. (See figure 3.8.)

As shown in figure 3.8, type `DemoWeatherKBase` in the textfield, making sure you get the spelling and the cases of the letters correct. After clicking "ok", if the resource used as a knowledgebase has

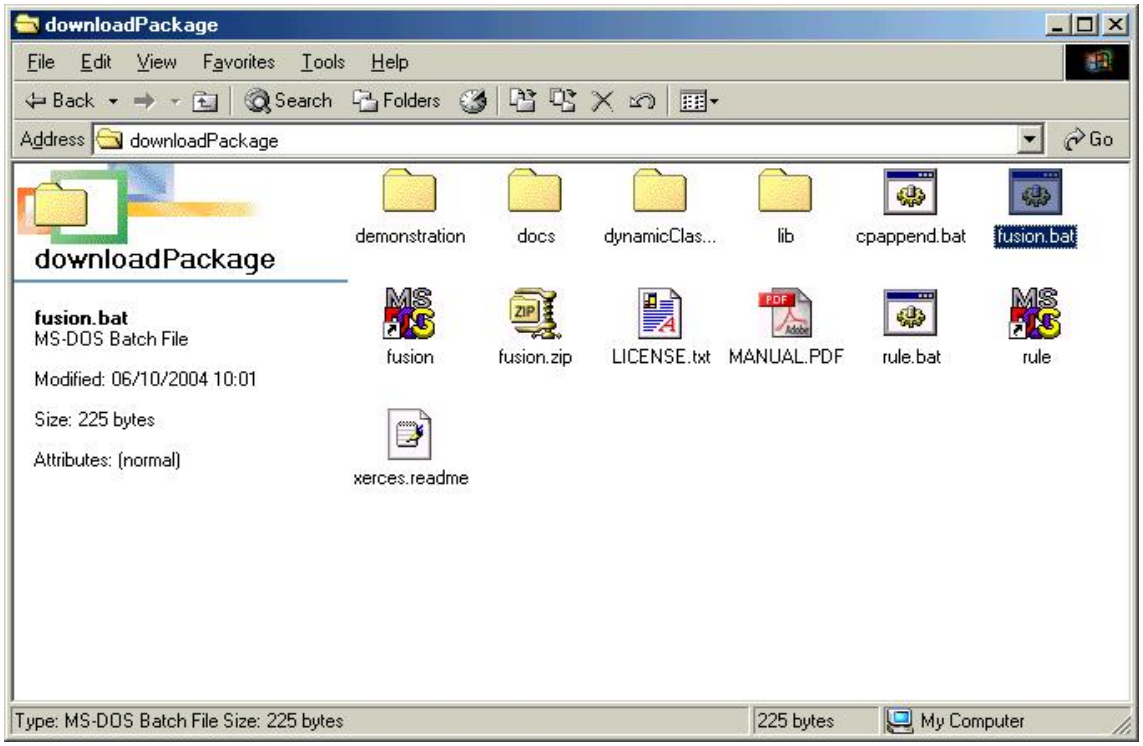


Figure 3.1: The directory in which `fusion.zip` was downloaded and unzipped. To run the fusion tool click on the file `fusion.bat` which, in this case, is highlighted in the top right corner.

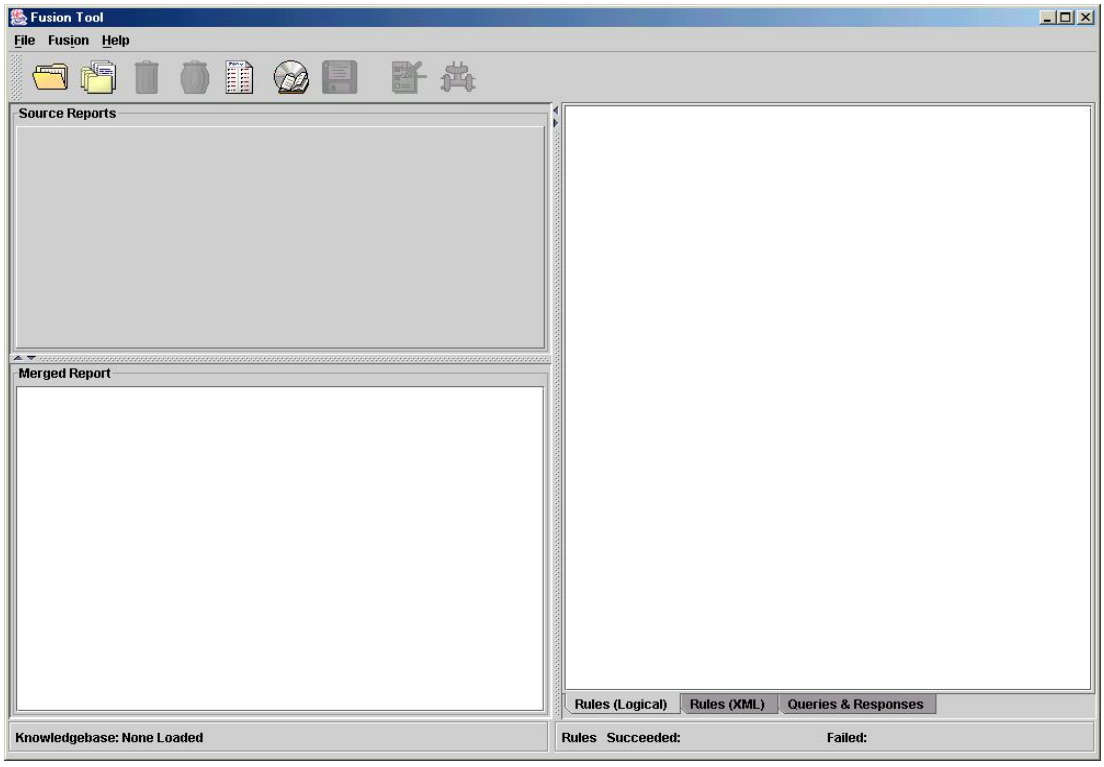


Figure 3.2: The fusion tool at start up. No input reports, rules, or knowledgebase have been loaded.

Figure 3.3: The fusion tool File menu. Choose the first option, “Select Report(s)”, to load the demonstration input reports.

loaded successfully, the “Knowledgebase” label in the lower panel should display the name of the knowledgebase.

Step 6 There is one more thing you must do before you can merge the reports. There are certain conditions that a set of fusion rules must meet if they are to be minimally adequate to merge a set of input reports. (See section 9.6 for an explanation of these conditions.) Accordingly, once a set of fusion rules has been loaded it must be checked prior to being used. You can do this by selecting the “Check Rules” option from the “Fusion” menu. (Or click on the check rules icon–the eighth–on the toolbar.) (See figure 3.9.)

Once you have done this a dialog will open informing you of the result. (See figure 3.10.) After clicking on “OK”, the merge icon on the tool bar and the “Merge” menu option are now enabled.

Step 7 To merge the reports simply click on the “Merge” menu option or the merge icon on the toolbar (the last icon). When the reports have been merged three things happen: (1) the output merged report is displayed in the lower left window. (2) A new tab labelled “Queries and Responses” will appear in the right window. This tab displays the queries generated by grounding the conditions or antecendents of the fusion rules with textentries from the reports. It also displays the responses of the knowledgebase to those queries. Conditions that succeed are highlighted by a green underlined “True”; those that fail are highlighted by a red underlined “False”. Again, don’t worry for now if you don’t understand what’s being displayed in this window. (3) Finally, the lower panel displays the number of rules that have succeeded and the number that have failed. (See figure 3.11.)

That’s all there is to merging the reports! As we mentioned, you may not understand the fusion rules and the queries and responses to and from the knowledgebase, but for now that’s not important. What is important is the merged report. Just by comparing the text content of the various tags in the merged report with the

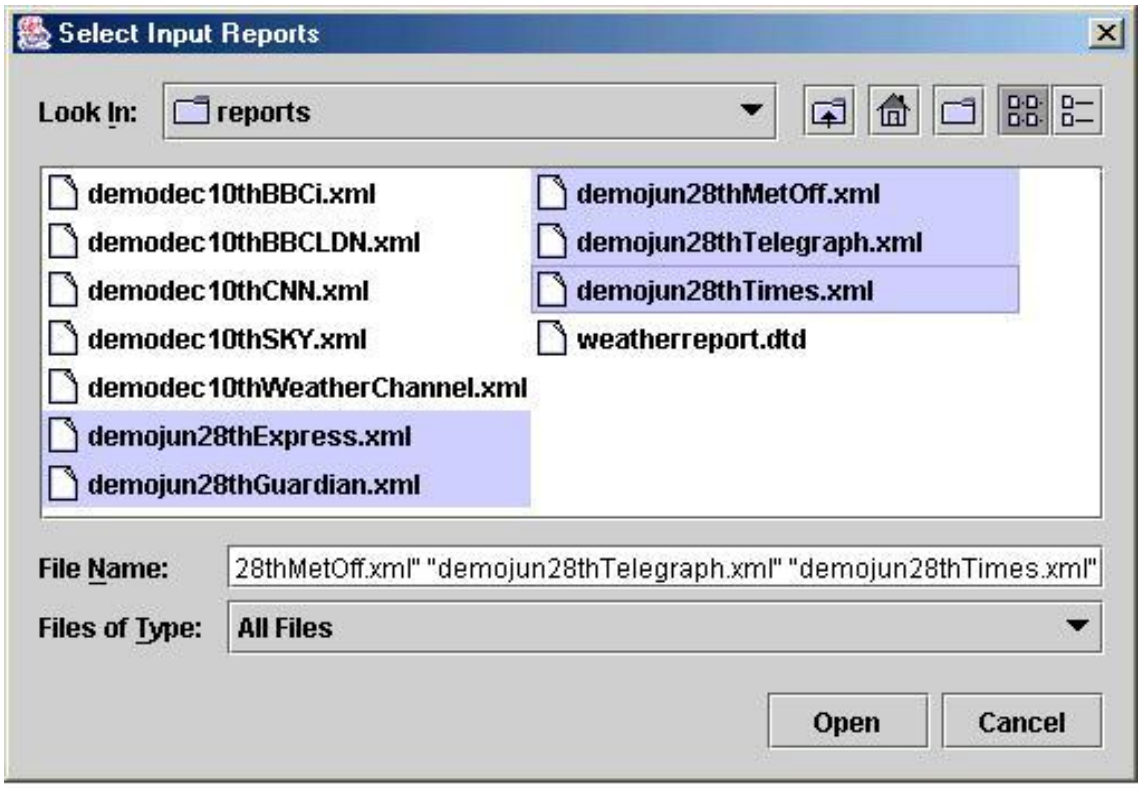


Figure 3.4: Selecting multiple input reports using the filechooser dialog. Here all five demonstration reports for June 28th have been selected.

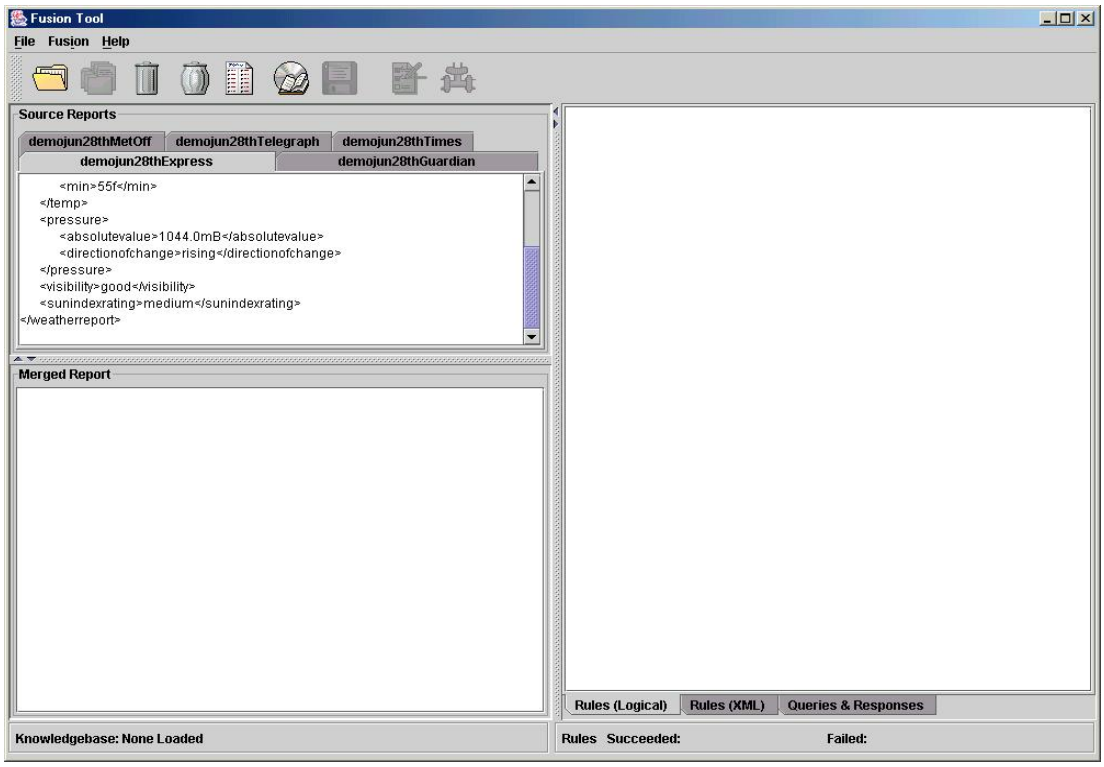


Figure 3.5: The fusion tool with the five demonstration input reports for June 28th loaded. Each report is displayed on a separate tab.

various textentries of the same tags in the input reports it should be fairly clear *how* the fusion rules are merging the reports. For precise details on how the reports in the demonstration are merged please consult Chapter 12. You can now experiment by adding or removing input reports, or by using the other set of fusion rules, and seeing what effect this has on the output merged report.

Finally, if you encounter any error messages in trying to run the demonstration please consult appendix A for an explanation.

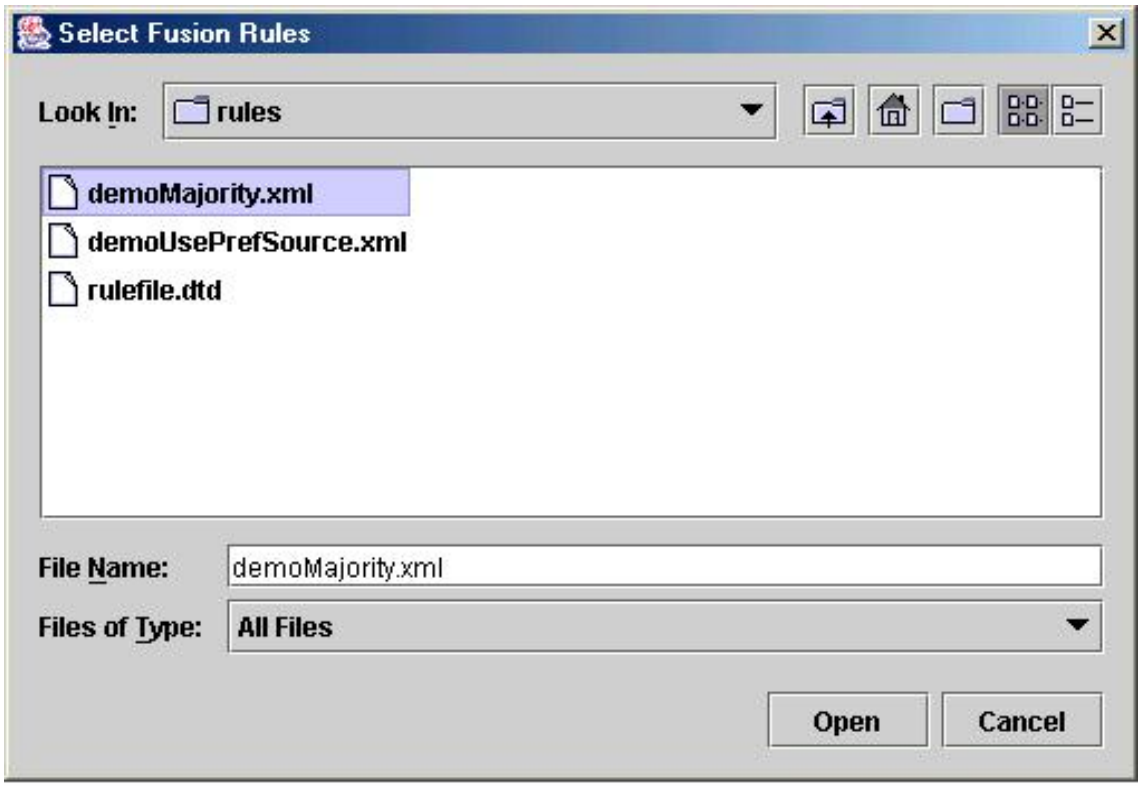


Figure 3.6: Selecting a set of fusion rules for the demonstration using the filechooser dialog. Here the set "demoMajority" has been chosen.

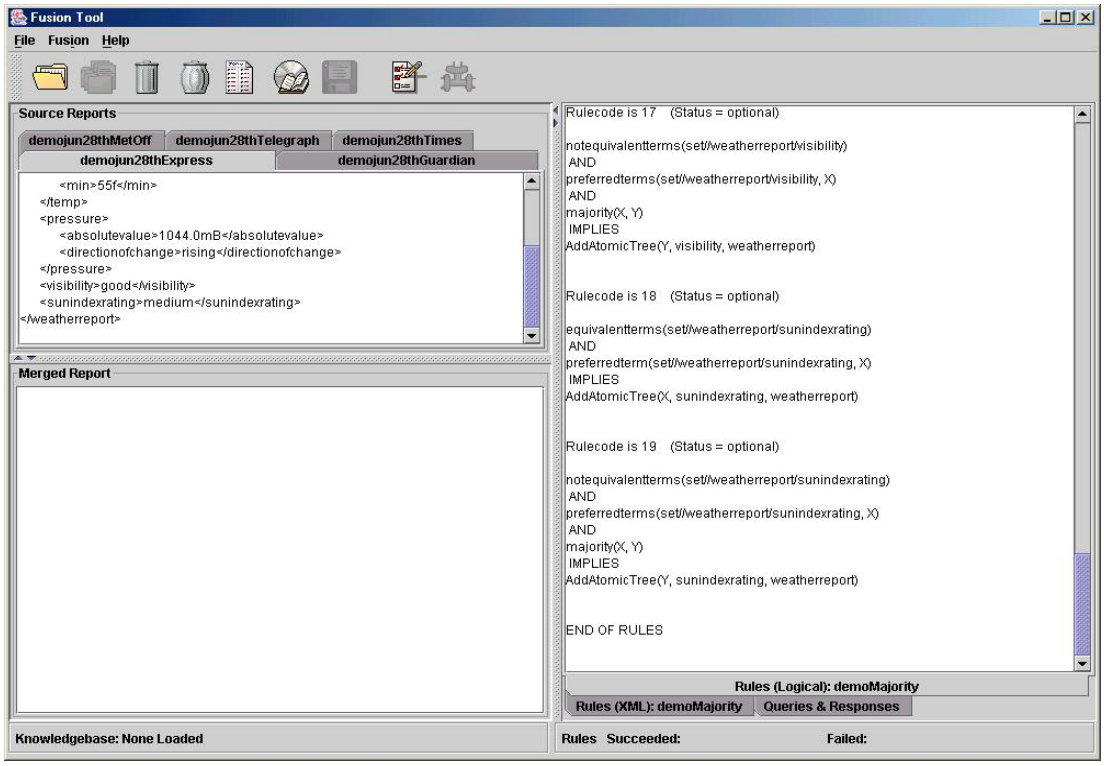


Figure 3.7: The fusion tool with the fusion rules loaded, as well as the five input reports. The name of the rules, “demoMajority”, is displayed on the tabs in the right hand window. The rules are displayed using both a logical notation (making them easier to read) and in XML.

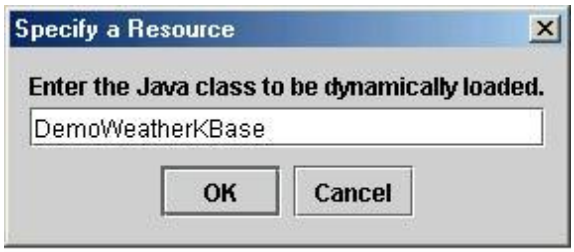


Figure 3.8: Selecting a knowledgebase for the demonstration application using the resource dialog.

Figure 3.9: The “Check Rules” option on the fusion tool “Fusion” menu. Before the reports can be merged, a set of rules must pass this check. (See section 9.6 for an explanation of the conditions being tested.)

Figure 3.10: Dialog confirming that the rules loaded have passed checks concerning foundational rules and Initialize actions.

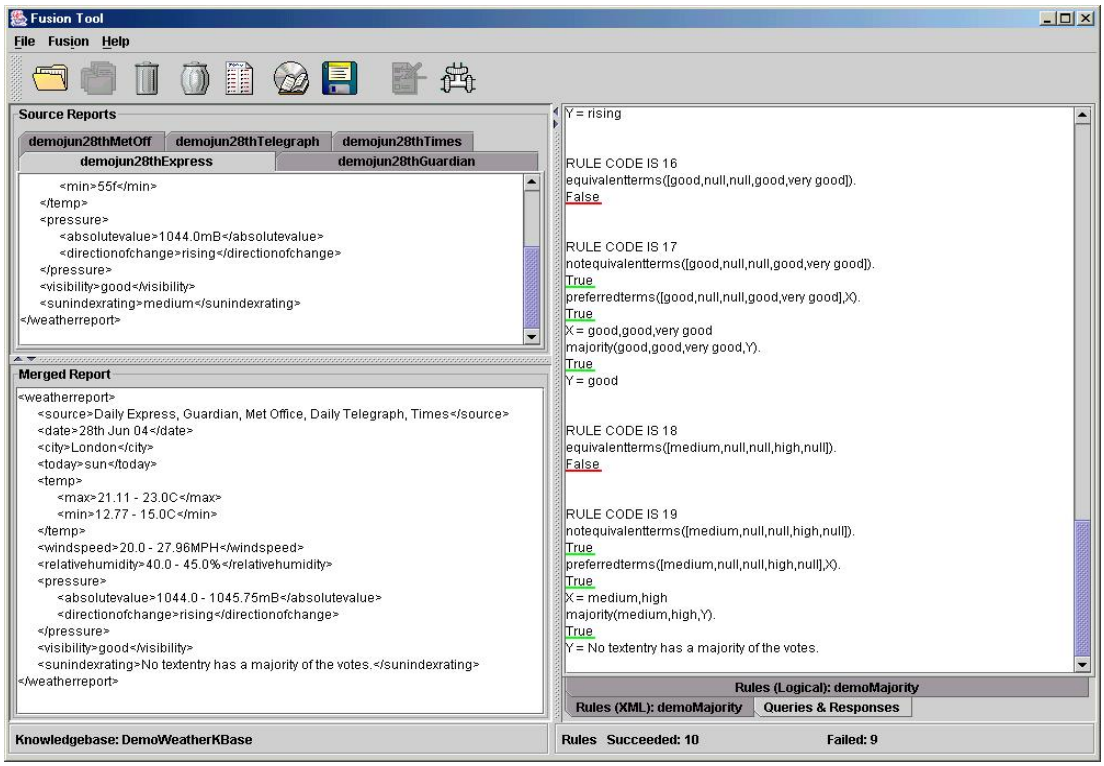


Figure 3.11: The fusion tool after the reports have been merged. The merged report is displayed in the lower left window; the right window displays the queries to, and the responses from, the knowledgebase (conditions that succeeded being underlined in green, those that failed, in red); and the lower panel records the number of rules that succeeded (10), and the number that failed (9).

Chapter 4

Overview

4.1 Structured input reports

An application to merge potentially inconsistent structured reports using fusion rules consists of the following three primary components: (1) The input reports to be merged; (2) A knowledgebase containing the background knowledge about the application domain in question; and (3) A set of fusion rules.

The input reports must be in XML. Although they may contain any XML features (such as attributes, namespaces, and processing instructions), the only XML features exploited by the fusion tool are elements and their textual content. Thus the information to be merged must appear as the textual content of the elements in the input reports. Moreover, to qualify as structured information, this content should not be free text, but rather single words or short phrases (or numerical values, with or without units) of the sort that can, for example, be defined in equivalence classes in a knowledgebase (say, "rain" or "heavy showers", etc.).

It is worth pointing out that there is a large amount of information readily available that is already in structured form. This includes information in XML repositories, some information on the Web, and much information in databases. All of these sources do of course also contain a lot of free text, but where the text occurs as words or short phrases then it qualifies as structured information. This kind of information can, of course, also be produced by hand, as was the case for the simple weather reports demonstration application.

4.2 Merging structured information using fusion rules

In our approach to merging structured reports, we draw on domain or background knowledge in a knowledgebase to help produce merged reports. The approach is based on fusion rules. These rules are of the form $\alpha \Rightarrow \beta$, where if α holds in the knowledgebase (i.e. it is a conjunction of queries that succeed in the knowledgebase), then β is an instruction that needs to be undertaken in the process of building a merged ouput report.

Fusion rules are thus comprised of condition literals (the antecedent of the rule) and action literals (the consequent). A single rule may have zero or more conditions, and one or more actions. Both conditions and actions are themselves comprised of variables, logical variables, and constants. Variables specify both what information is to be extracted from an input report, and from which report it is to be extracted. This

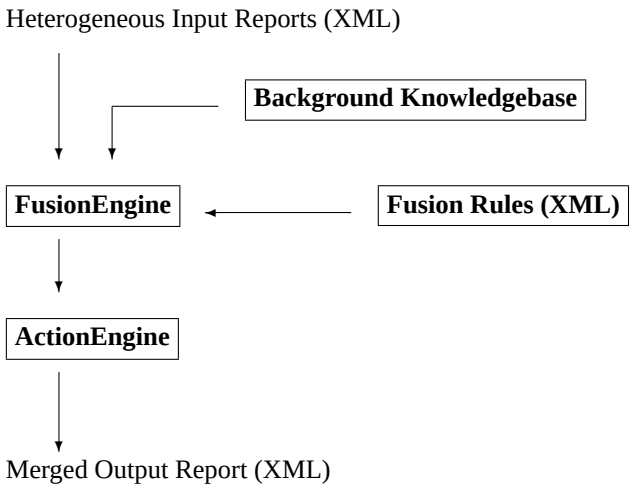


Figure 4.1: The basic architecture for a fusion system that merges information in the heterogeneous input reports using a knowledgebase and fusion rules. The FusionEngine and ActionEngine modules are implemented in Java. The fusion engine executes a set of fusion rules (an XML file containing the fusion rules marked up in FusionRuleML), by grounding the fusion rules with textentries from the structured reports, and then checks whether all the conditions of each ground fusion rule are implied by the knowledgebase and, if they are, then the ground actions of the rule are added to the actionlist (a list of actions that specify how the merged report should be constructed). The ActionEngine executes the actionlist to build a merged report.

information is then used to instantiate the fusion rules prior to querying the knowledgebase. Because either individual reports or the whole set of input reports may be specified as the source, variables are of two types: set variables and singleton or "source" variables. Constants take constant values which either denote targets in the merged report to which text entries and subtrees are to be added, or numerical values specifying, for example, voting thresholds or the number of nodes to be added to the output report, etc. Finally, logical variables indicate where the knowledgebase is expected to provide a binding that may be used either as input to a subsequent condition, or as output for the merged report.

To merge a set of structured reports, we start with the knowledgebase and the information in the input reports to be merged, and apply the fusion rules to this information. For a set of structured reports and a set of fusion rules, we attempt to ground each fusion rule with textentries or subtrees from the structured reports, and then check whether all the conditions of each ground fusion rule are implied by the background knowledge, and if they are, then the ground actions of the rule are added to the actionlist (a list of actions that specify how the merged report should be constructed).

The FusionEngine is the software that is responsible for taking each fusion rule in turn, grounding the variables with the required information from the input reports, and posting the resulting queries to the knowledgebase, and if all the conditions succeed, then posting the resulting actions to the ActionEngine. The ActionEngine is the software responsible for taking the actions and building the merged reports. The ActionEngine is implemented to handle a range of actions such as adding text entries and nodes to the merged report. (The basic architecture for a fusion system is given in Figure 4.1.)

Please note that the function of the fusion tool provided with this package is simply to merge the reports using the rules and the knowledgebase. It is not a knowledgebase editor or builder. The knowledgebase must be built independently of, and preferably prior to (for testing purposes), the fusion rules. There is a

rule tool supplied to make it easier to write and edit the rules.

4.3 System architecture and the knowledgebase

The fusion tool is designed so that the knowledgebase can be implemented in the way chosen by the application developer. For example, it could be a Prolog program (i.e. a .pl file), some XML files, or an SQL database. Because queries to these different resources must be formatted in different ways, and because they return results in different formats, there must be an interface between the knowledgebase and the fusion tool. This interface is in Java and must be written by the application developer after the choice of knowledgebase resource has been made, because, as just mentioned, the function of this interface is to parse the queries produced by the fusion tool into the particular format required by the knowledgebase, and to reparse the proprietary responses of the knowledgebase back into the format required by the fusion tool.

Clearly this makes it harder to develop an application than it would otherwise be. But this architectural decision was taken as the solution to a familiar trade-off between simplicity and ease of use, on the one hand, and versatility, on the other. Freeing the developer of a fusion application from the burden of writing the Java interface would have tied the developer to exploiting a particular kind of resource as a knowledgebase.

4.4 What you need to know to develop an application

Here is a checklist of what the developer needs to know to develop a fusion application.

- You must understand and be able to write fusion rules. It helps to understand the fusion rule ML, but you can avoid writing and editing raw XML if you use the rule tool.
- You must be able to get the information you want to merge into XML, though it only needs to be in very simple XML using elements and their textual content.
- If you use the rule tool to write or edit fusion rules you must be able to write a DTD for the reports that those rules are intended to merge. (This helps the tool check that the rules are correct.)
- You must be able to code the Java interface to the knowledgebase.
- If you do not already have one, you must be able to build the knowledgebase proper, for example the Prolog program or SQL database, etc.

If this sounds complicated, that's because it is! Developing a merging application using fusion rules is not a simple process, like using a browser to access the internet, or a word processor to write a letter. Just like building an ontology which embodies knowledge about a particular domain, or just like building an application using an information mediator to integrate information on the Web, building an application to merge information using fusion rules and a knowledgebase is a time consuming process. There are no short cuts to developing this kind of application. On the other hand, the software supplied with this package to edit rules and merge reports is extremely simple to use. Merging reports using the fusion tool is simply a matter of loading the reports, loading the rules (and running a quick check on them), loading the knowledgebase, and then clicking on the "Merge" button! In other words, it's simply a matter of some file management operations. What is hard work is building the application.

Chapter 5

Background

5.1 Merging conflicting information

Terms such as “merging”, “fusion”, and “information integration” can be used in a variety of ways. It is useful, therefore, to start by explaining what we mean by those terms, and to illustrate thereby the kinds of problems that applications built using fusion rules and a knowledgebase might be used to solve. After doing this we will briefly survey some existing approaches to merging or integrating information in order to see how they compare with our approach. It will be our contention that they are all in fact aiming to do something rather different from what we wish to accomplish. The purpose of this chapter, then, is to explain when it would be appropriate to build a fusion system application, and why someone might want to build one.

The kind of merging (or fusion, or integration) of information with which we are primarily concerned in a fusion rules application is to take potentially $conflicting$ information from different sources and, by reasoning about those conflicts, to create a distinct, novel output report that summarizes, or in other ways combines, that information. The upshot of merging in this sense is that a new source of information is created, something that did not previously exist. For information to conflict, however, it must both be about the $same$ thing, and it must be about the $same$ properties or attributes of that thing. If the information is about different things, or about different properties of the same thing, then it cannot be conflicting information; it would, rather, be $complementary$ information. We can of course speak of integrating complementary information, as for example in cases where we bring together information about different properties of the same thing, and this kind of integration can be carried out using fusion rules. But, as we discuss below, there are other technologies that deal with integrating this kind of information. It is with conflicting information, however, that fusion rules come into their own as a technology for information integration.

A couple of examples should make this clearer. In example 5.1.1, the two reports to be merged both contain information about the same things—the major political parties—and they both contain information about the same properties or attributes of those things—their poll ratings. But the properties ascribed by the two reports to the same things $conflict$. Merging or integrating these two reports, therefore, requires some way of dealing with this conflict.

Example 5.1.1 *Consider the following two conflicting (and imaginary) reports concerning opinion polls for the political parties in Great Britain.*

```
⟨opinionpoll⟩
        ⟨source⟩ Mori ⟨/source⟩
        ⟨client⟩ Guardian ⟨/client⟩
        ⟨date⟩ 19/3/04 ⟨/date⟩
        ⟨parties⟩
                ⟨party⟩ Labour ⟨/party⟩
                ⟨poll⟩ 41% ⟨/poll⟩
                ⟨party⟩ Conservative ⟨/party⟩
                ⟨poll⟩ 39% ⟨/poll⟩
                ⟨party⟩ Liberal Democrat ⟨/party⟩
                ⟨poll⟩ 21% ⟨/poll⟩
        ⟨/parties⟩
⟨/opinionpoll⟩
```

```
⟨opinionpoll⟩
        ⟨source⟩ Gallup ⟨/source⟩
        ⟨client⟩ Daily Telegraph ⟨/client⟩
        ⟨date⟩ 20 March 2004 ⟨/date⟩
        ⟨parties⟩
                ⟨party⟩ Lab ⟨/party⟩
                ⟨poll⟩ 38% ⟨/poll⟩
                ⟨party⟩ Con ⟨/party⟩
                ⟨poll⟩ 42% ⟨/poll⟩
                ⟨party⟩ Lib Dem ⟨/party⟩
                ⟨poll⟩ 19% ⟨/poll⟩
        ⟨/parties⟩
⟨/opinionpoll⟩
```

Merging or integrating these two reports in the way that we have in mind might result in the following merged report. In this case the entry for each `poll` *result is determined by finding the mean of the* `poll` *results in the two reports.*

```
⟨pollofpolls⟩
        ⟨sources⟩ Mori and Gallup ⟨/sources⟩
        ⟨clients⟩ Guardian and Daily Telegraph ⟨/clients⟩
        ⟨date⟩ 19/3/04 ⟨/date⟩
        ⟨parties⟩
                ⟨party⟩ Labour ⟨/party⟩
                ⟨poll⟩ 39.5% ⟨/poll⟩
                ⟨party⟩ Conservative ⟨/party⟩
                ⟨poll⟩ 40.5% ⟨/poll⟩
                ⟨party⟩ Liberal Democrat ⟨/party⟩
                ⟨poll⟩ 20% ⟨/poll⟩
        ⟨/parties⟩
        ⟨summary⟩ Conservative lead of 1% ⟨/summary⟩
⟨/pollofpolls⟩
```

Similarly, in example 5.1.2 all four input reports are about the same *thing*, the Royal Bank of Scotland, and they all contain the same *kind of information* about it–information about its stock price and a recommendation to buy or sell. But that information *conflicts*. So once again, if this information is to be merged, something must be done about this conflict (even if it is only to conjoin all the conflicting values, perhaps with their sources, so as to convey the fact that there *is* a conflict.).

Example 5.1.2 *Consider the following four conflicting (and imaginary) reports concerning stock in the Royal Bank of Scotland.*

```
⟨stockreport⟩
        ⟨stock⟩ RBS ⟨/stock⟩
        ⟨sector⟩ financial services ⟨/sector⟩
        ⟨date⟩ 19/3/04 ⟨/date⟩
        ⟨source⟩ Financial Times ⟨/source⟩
        ⟨price⟩ 209.5 ⟨/price⟩
        ⟨recommendation⟩ buy ⟨/recommendation⟩
⟨/stockreport⟩
```

```
⟨stockreport⟩
        ⟨stock⟩ Royal Bank of Scotland ⟨/stock⟩
        ⟨sector⟩ finance ⟨/sector⟩
        ⟨date⟩ 21 March 2004 ⟨/date⟩
        ⟨source⟩ HSBC ⟨/source⟩
        ⟨price⟩ 209 ⟨/price⟩
        ⟨recommendation⟩ sell ⟨/recommendation⟩
⟨/stockreport⟩
```

```
⟨stockreport⟩
        ⟨stock⟩ RBS ⟨/stock⟩
        ⟨sector⟩ financial services ⟨/sector⟩
        ⟨date⟩ 20 − 03 − 04 ⟨/date⟩
        ⟨source⟩ Waterhouse ⟨/source⟩
        ⟨price⟩ 209.5 ⟨/price⟩
        ⟨recommendation⟩ buy ⟨/recommendation⟩
⟨/stockreport⟩
```

```
⟨stockreport⟩
        ⟨stock⟩ RBS ⟨/stock⟩
        ⟨sector⟩ banking ⟨/sector⟩
        ⟨date⟩ 18th March 2004 ⟨/date⟩
        ⟨source⟩ Morgan Stanley ⟨/source⟩
        ⟨price⟩ 209.25 ⟨/price⟩
        ⟨recommendation⟩ buy ⟨/recommendation⟩
⟨/stockreport⟩
```

Integrating or fusing these input reports in the way that we mean might produce the following merged report. In this case the entry for `overallrecommendation` *is determined by a preference for HSBC over the others as a source, and so is* `sell`.

```
⟨stockreport⟩
        ⟨stock⟩Royal Bank of Scotland ⟨/stock⟩
        ⟨sector⟩ financialservices ⟨/sector⟩
        ⟨date⟩ 18 − 21/3/04 ⟨/date⟩
        ⟨price⟩ 209 − 209.5 ⟨/price⟩
        ⟨sources⟩
                ⟨source⟩ Financial Times ⟨/source⟩
                ⟨recommendation⟩ buy ⟨/recommendation⟩
                ⟨source⟩ HSBC⟨/source⟩
                ⟨recommendation⟩ sell ⟨/recommendation⟩
                ⟨source⟩ Waterhouse ⟨/source⟩
                ⟨recommendation⟩ buy ⟨/recommendation⟩
                ⟨source⟩ Morgan Stanley ⟨/source⟩
                ⟨recommendation⟩ buy ⟨/recommendation⟩
        ⟨/sources⟩
        ⟨overallrecommendation⟩ sell ⟨/overallrecommendation⟩
⟨/stockreport⟩
```

An alternative way of merging these reports, with the entry for `overallrecommendation` *being* `buy` *instead, would be a simple first past the post, or a majority, voting function.*

It is in situations such as these that we envisage that an application built using fusion rules and a knowledgebase would be useful.

5.2 The Semantic Web

By contrast, several other technologies already exist that claim to integrate or merge information, but they all strike us as attempting to do something somewhat different.

Chief amongst technologies whose aim could be described as integrating information are those associated with the Semantic Web[1], such as the Resource Description Framework (RDF) and RDF schema (RDFS), the Ontology Inference Layer (OIL, also used as an acronym for "Ontology Interchange Language"), and the DARPA Agent Mark-up Language (DAML)[2]. The combination of these technologies is intended to make the information on the Web not just machine *readable*, as at present, but machine *understandable*, thus increasing the "semantic interoperability" of the Web. If this is to be done, what we need is semantic metadata–data that tells machines what the content on the Web means–rather than metadata that simply tells machines how to display Web content (HTML tags), or what syntactic structure it has (XML).

The consensus is that such semantic metadata is best provided via ontologies, which offer a source of shared and defined terms standing in clear relationships to one another. These ontologies will provide us with the vocabulary to mark up or annotate the semantics of information on the Web. DAML+OIL is an ontology representation language allowing for the specification and exchange of ontologies that is designed to provide this intelligent access for machines to the heterogenous and distributed information found on the Web. In particular, three features suit it to this purpose: (1) The use of a frame or class based modelling framework makes it easier for humans to construct ontologies; (2) The choice of XML and RDF Web

[1] Tim Berners-Lee, *Weaving the Web*, Orion Business Books, 1999. T Berners-Lee, J Hendler and O Lassila, "The Semantic Web", *Scientific American*, 2001, volume 284, pp. 34–43.

[2] D Fensel, "The Semantic Web and its languages", *IEEE Intelligent Systems*, volume 14, p. 67, 2000.

languages as a syntax makes DAML+OIL interoperable with existing Web software; (3) The decision to design the language so that its semantics can be mapped to a description logic allows for reasoning in the construction and maintenance of ontologies, something that is very useful when checking the consistency of large ontologies.

Given this account of the purpose of these technologies, they can indeed be said to integrate information in the sense that they enable machines to determine the content or meaning of that information, and thus determine, for example, whether two sources contain information about the same thing. But it is important to note that they do this by acting on or exploiting this new semantic metadata; they do not concern themselves with the data–the actual Web content–itself. The significance of this is that although they can be said to integrate diverse, heterogeneous data, perhaps it would be more accurate to say that they put us in a position of being able to integrate such data. For, knowing what information on a Web site is about is one thing, knowing what to do with it is another. And, as the developers of DAML+OIL have acknowledged, "Defining languages for the Semantic Web is just the first step. Developing new tools, architectures, and applications is the real challenge that will follow." (see Fensel, $op.\ cit.$, p. 67.)

With respect to our interest in merging information, the relevance of these technologies is that they enable us to establish that the information to be merged is (or is not) about the same subject matter. But having done that, in themselves they say nothing about what to do with that data itself–what we should do if it does not conflict, what we should do if it does. Thanks to the Semantic Web, software agents will be able to tell us that one website on Claret contains information about the same thing as another website containing information on the wines of Bordeaux. But suppose these sites also contain differing opinions on the merits of the 2003 vintage. How should we deal with this conflict? In themselves, the languages developed to enable the Semantic Web say nothing about what to do here, but this is exactly the kind of task that we have in mind when we speak of merging information. Merging or integrating information, as we are using those terms, is one of those further applications that need to be developed on top of the Semantic Web, along with applications such as intelligent search engines and other envisaged applications in e-commerce and knowledge management. Our view of the Semantic Web, then, is that it is an enabling technology for information integration. The fusion system that we advocate could be considered an information integration agent that could make use of Semantic Web technology.

5.3 Information mediators and information integration on the Web

The aim of merging, as we choose to understand it, is to take potentially conflicting information from different sources and, by reasoning about those conflicts, to create a distinct, novel output report that summarizes, or in other ways combines, that information. The upshot of merging in this sense is that a new source of information is created, something that did not previously exist.

This aim seems to be substantially different from most extant approaches to merging or integrating information, where the emphasis often seems to be more on extracting information from different sources and presenting it in such a way as to make it easier for the user to find the information she wants. The upshot of merging or integration in this sense is the user finds a pre-existing source of information; no new information is created (although, of course, the information will be new to the user–that's why she's looking for it), rather, existing information is made more useable.

The difference between these two projects can perhaps best be brought out in terms of the familiar entity-attribute model. Our interest in merging is to take information from different sources about the same entity, where that information concerns the *same* attributes, and then combine it. If the values of those attributes are the same (or similar), not much reasoning will be involved. But where the values of those attributes conflict, a variety of kinds of reasoning using a knowledgebase can be used to resolve the conflict. By contrast, most other projects of integrating information on the web seem concerned with a different task: they start from the same place, with different sources of information about the same entity, but those sources

typically contain information about *different* attributes. Thus these different data sources are conceived of as containing *complementary* information, not *conflicting* information. As a result, merging is conceived of, not as the process of resolving a conflict, but as the process of combining the information about the different attributes of a given entity, thus making it easier for the user to find the information she wants.

A couple of examples should make this clearer. One example is the TheatreLoc application[3]: here the user chooses to search for restaurants and/or cinemas in a specified city, and is presented with a list of them, together with their locations displayed on an interactive map. The user can then navigate via the list or the map to detailed information on individual theatres or restaurants, including watching previews of the films. In this application, information about a theatre's films is brought together with video previews, and information about the street address of the theatre is used to find the theatre's grid reference, allowing it to be placed on a street map. The end product is a user interface making it easier for the user to find what she wants; a single Website now allows the user both to find what's showing, and how to get there. In that sense we can speak of information integration.

A second application[4] seeks to integrate the information contained in restaurant review websites such as Fodor's and Zagat's, with information on the health or sanitation status of those restaurants contained on a local government Website. Combining this information would allow users to answer such queries as "Find all the Japanese retaurants in Santa Monica with a grade A health rating." Once again, the purpose of integration here is not to reason about conflicting data, but to make it easier for the user to find information which already exists. The end result is that the user is looking at the same Web pages that she could have found manually, but she got their a lot more easily.

Both applications work by using the Ariadne information mediator forming an intermediate layer between the user and the various heteregeneous information sources. The purpose of the mediator is to provide a uniform query interface, abstracting away from the heteregeneous formats of the different sources. The mediator also plans how the sources should be queried and how the data that is retrieved is to be integrated. The extraction itself is done by wrappers. It should be acknowledged that this sort of information integration does involve some reasoning concerning conflicts, but this is confined to determining if information from two sources is in fact about the same entity, say the same film or restaurant. For example, we need to be able to tell that information from a cinema Website about a film called "A bug's Life" is information about the same film that is listed as "Bug's Life, A" on a trailer Website, so that links to the two can be combined. Or, we need to know whether a restaurant listed as "Art's Delicatessen" on one site is the same as one listed as "Art's Deli" on another. The role of reasoning about conflicts thus seems to be restricted to providing robustness to the semantic heterogeneity of different information sources in their description of the entities they are about. These applications do not seem to provide a role for reasoning about conflicts between the values of the attributes of those entities.

There is no question that the sort of issues in integration addressed by these applications are pressing, but the end result of this approach is that it is easier for the user to find one or more of the pre-existing input sources that she's interested in. Resolving conflicts between sources may be required in order to do this, but it remains a subsidiary goal. By contrast, the objective of our approach is that the user gets to see a wholly different, merged or summarized, output report. The focus of our approach is in developing ways of merging or integration that can deal with a whole array of problems that arise with conflicting information of which the examples just mentioned in the case of these web integration applications are only one.

[3] Greg Barish, Craig A Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot and Cyrus Shahabi, "TheaterLoc: A Case Study in Information Integration", *Information Integration Workshop, Stockholm, Sweden (IJCAI '99)*, 1999.

[4] Craig A Knoblock and Steven Minton, "The Ariadne approach to Web-based information integration", *IEEE Intelligent Systems*, volume 13, number 5, 1998, pp. 17–20.

5.4 Database integration

The goal of database integration is to provide uniform access to multiple, heterogeneous databases that each have their own associated local schema. Logic-based techniques in data integration, such as global-as-view and local-as-view, offer some ability to relate sources using restricted forms of firstorder logic, and so can be considered as special cases of knowledgebased merging. However, the format of the clauses used are largely limited to defining virtual tables directly in terms of existing tables. We could describe this process of integration as providing a mapping of one schema onto another, so that, for example, data in a column headed location in one table is mapped onto a column headed address in another table. So, "fusion" or "integration" in this context refers to the "combining" of several database tables into one. These "mappings" can be regarded as a form of ontological knowledge.

If this is how the goal of database integration should be understood, then we can see, once again, that this goal is substantially different from the primary goal of merging that we are advocating. Data integration in the former sense is the combining of information previously available in different applications, and making that same information available to a single application so as to enable easier access and querying. But having brought the information together, this approach in itself has nothing to say about what should be done with it. By contrast, "integration" as we mean to use the term stands for the use of logical inference to create a new piece of information that was not necessarily previously available. As was the case with the semantic web, far from being a competitor, we can view database integration as a useful enabling technology that provides the kind of input that could subsequently be merged in the ways that we envisage.

5.5 Conclusion

In one way or another the various technologies we have briefly examined in this chapter are concerned with giving the user access to an *existing* piece of information–a piece of information that is already there on the Web, or in a database, but that is difficult to access, perhaps because of the vast amount of information available, or because it's spread accross more than one database. By contrast, what's different about the kind of merging of information that we have highlighted is that it is designed to create a *new* piece of information–the merged output report–from these existing sources.

None of the comments in this chapter should be taken as criticisms of these technologies. We believe, rather, that they are all aiming to do something somewhat different from the task that we envisage for an application built using fusion rules. Because these technologies are dealing with different issues they should all be considered, not as in competition with fusion rules applications, but as complementary to them. In particular, the problems of information extraction that some of these technologies deal with are problems that are ignored by fusion rules themselves, which simply assume that the information to be merged already exists in XML. Moreover, the problem of deciding whether input reports that appear to be about the same entity really are about the same entity is likewise ignored. Hence, any full-fledged commercial application built using fusion rules would likely require support from some of these other technologies.

Chapter 6

The fusion tool manual

6.1 Overview

The fusion tool is a piece of software designed to do only one thing—merge the input reports using a set of fusion rules and a knowledgebase. It is, therefore, very simple to use, amounting to litle more than some simple file management operations. The basic workflow is this:

1. Load the input reports.
2. Load the knowledgebase.
3. Load the rules.
4. Run a check on the rules.
5. If the rules pass the check, merge the reports.
6. Save the merged report (obviously this is optional).

The reports, rules and knowledgebase can be loaded in any order, though you can of course check the rules only after having loaded them.

As mentioned, the fusion tool allows you only to merge the reports. You cannot use it to edit the reports, or edit the rules, or edit the knowledgebase. If you want to edit the rules, a rule tool is provided (see Chapter 7), or you can simply edit the raw XML using a suitable editor.

6.2 Layout of the fusion tool GUI

The fusion tool contains three main panels. (See figure 6.1) These are:

Source Reports This panel displays the reports to be merged. If the “Select Report(s)” menu option is chosen, each report is displayed on a separate tab in a tab pane.

Merged Report This panel displays the merged report. If the reports cannot be merged because of some error condition, a message detailing that condition is displayed instead.

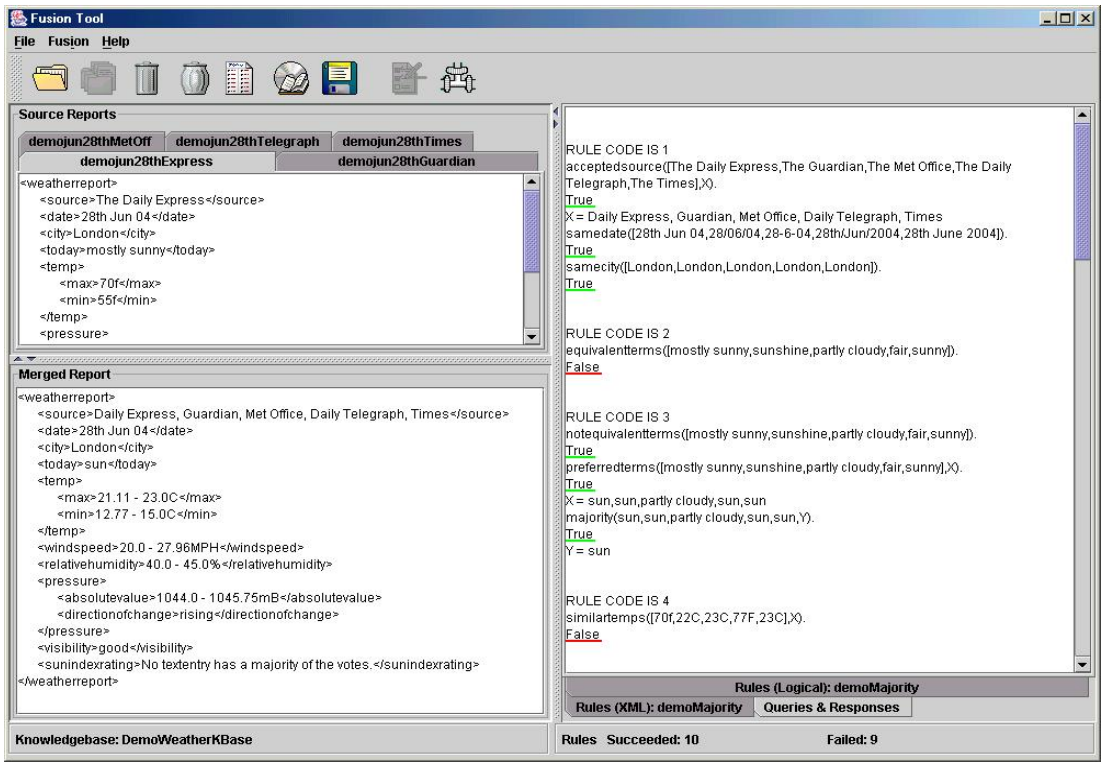


Figure 6.1: The fusion tool for merging input reports using a set of fusion rules and a knowledgebase. The input reports are displayed on tabs in the upper left window; currently displayed in the right window are the queries to and the responses from the knowledgebase. The lower panel shows the knowledgebase being used (*DemoWeatherKBase*), and the number of rules that have succeeded and failed.

Rules and Responses The right hand panel contains a tab pane with three tabs. Two of those tabs display the rules in both logical notation and in their XML form. The other tab displays the queries to, and responses from the knowledgebase.

In addition to these three main panels a subsidiary panel running beneath them displays the name of any resource loaded to serve as a knowledgebase, and, once the reports have been merged, the number of rules that succeeded and failed.

6.3 Fusion tool operations

We now describe briefly the functions of the various menu and toolbar options.

FILE MENU

Select Report(s) This option allows you to select one or more input reports. The resulting file chooser dialog enables multiple selection in the standard ways: either select multiple reports by holding down the `control` key, or if you wish to select multiple reports that are listed consecutively, select the first, depress the `shift` key, and then select the last report; all the intervening reports should be highlighted. Any reports selected using this option are displayed individually on separate tabs in the window labelled “Source Reports”. Because displaying each report on a separate tab takes up room

Figure 6.2: The fusion tool File menu.

in the source reports window, if you wish to load more than around a dozen input reports you should use the next option, "Load Batch of Reports". Note that if any report selected is not an XML file then an error (Error 1, see Appendix A) will result. Also note that once a report has been loaded using this option, the "Load Batch of Reports" option is unavailable until all reports have been deleted.

Load Batch of Reports The underlying functionality of this menu option is the same as that of the "Select Report(s)" option — it loads and displays one or more input reports.The difference between them is cosmetic; to avoid overcrowding the source reports window with tabs, any reports selected using this option are not individually displayed on separate tabs. Instead, a list of their file names is displayed on a single tab. Thus you should use this option to select a larger number of reports, say, more than a dozen or so. Again, if any selected report is not an XML file then an error will result. Note that once a report has been loaded using this option, both this menu option, and the "Select Report(s)" option are unavailable until all reports have been deleted.

Delete Selected Report This option deletes the report on the tab that is currently selected. Note that this both removes the report from the display, and removes the report from the set of reports to be merged. Thus, if you merge the reports, delete a report, and re-merge, you may well get a different merged report. Note also that if you delete a tab that is displaying the names of a batch of files (see "Load Batch of Reports" above) then *all* the input reports are deleted. If the deletion of a report results in less than two reports remaining, the "Merge" option will be disabled. If you delete all the reports after previously merging them, the Merged Report window and the Queries and Responses tab will clear.

Delete All Reports Use this option if you wish to delete all the input reports. As with the "Delete Selected Report" option, this not only removes all the reports from the display, it removes all reports from the set of reports to be merged. It also results in the "Merge" option being disabled. Again, if you delete all the reports after previously merging them, the Merged Report window and the Queries and Responses tab will clear.

Figure 6.3: The fusion tool Fusion menu.

Select Rules This option selects a set of fusion rules. If the file selected is not an XML file, or if it is an XML file but it does not conform to the rulefile DTD, then an error (Error 2, see Appendix A) will result. If no error results, then the rules are loaded and they are also displayed on tabs, both in their logical form and in XML. Note that once a set of rules has been selected, even if some reports and a knowledgebase have already been loaded, the "Merge" option will not yet be enabled. First you must check the rules to see if they conform to some constraints regarding foundational rules and Initialize actions. (See section 9.6.2) Once this check has been passed the reports may be merged. If you load a new set of rules after having previously merged some reports, then the Merged Report window and the Queries and Responses tab will clear.

Select Resource This option loads the resource that is to be used as a knowledgebase. If a resource is loaded, the name of the resource is displayed on a label in the lower panel of the fusion tool. The architecture of the fusion tool is designed so that the user may exploit any kind of resource she chooses, for example a Prolog program, some XML files, or an SQL database. To decouple the resource in this way, the fusion tool exploits Java's dynamic class loading facility. Instead of hard coding the link to a particular kind of resource into the fusion tool, the application developer writes her own Java code for this link, and the class containing that code is then dynamically loaded. It is the name of this Java class that must be typed in the dialog box that appears when this option is selected. For example, if the Java class is named "MyClass.class", you should type `MyClass` in the dialog textfield. If you misspell the class name (as, say, `MyGlass`), an error of type 3 will result. If you get the spelling correct, but the *case* of one or more characters wrong, an error of type 4 will result. If you load a Java class that does not conform to the `AnswerQuery` interface then an error of type 14 will occur. (See Appendix A) If you load a resource after having previously merged some reports, then the Merged Report window and the Queries and Responses tab will clear.

Save Merged Report Select this option to save the merged report as an XML file.

Exit Quit the fusion tool.

All of these options, except the last, are also available via the tool bar.

FUSION MENU

Check Rules Once a set of rules has been loaded it must be checked to ensure that it conforms to some simple conditions regarding foundational rules and Initialize actions. (See section 9.6.2) The reason for this is that if we attempted to merge a set of rules that, say, had no Initialize action, then whatever the results of the queries to the knowledgebase, no merged report would be constructed to which they could be added. Again, if a set of rules contained a rule with two Initialize actions then, if that rule succeeded, the fusion tool would attempt to build two merged reports. In order to avoid such situations, then, the rules must first be checked using this option. Their passing this check is a necessary condition for merging the reports. If the rules fail in one or more respects, details of the failures will be given in an error message. (Error 12)

Merge This option merges the reports using the rules and the knowledgebase. The conditions which are jointly necessary and sufficient for this option to be enabled are: at least two source reports have been loaded; a resource has been loaded; and a set of rules has been loaded and checked. If these conditions obtain, this option is enabled. For the reports to be merged, however, other conditions must also obtain, but these can only be checked at "merge-time". For example, no foundational rule that does not contain an Initialize action may fail (Error 7). Other errors, too, may result in the reports not being merged. For example a "source variable" has been incorrectly numbered (Error 6); an action is unknown (Error 8); an action has an illegal argument (Error 9); or a rule contains an invalid merged report branch (Error 10). Hence, it does not follow from the fact that the "Merge" option is enabled that the reports will in fact be merged. However, if the reports fail to be merged then the Queries and Responses tab will display the results of querying the knowledgebase. Obviously these results can be very useful for diagnostics purposes. The exception to this is if an error of type 6 occurs. In this case what has gone wrong is that one of the "source" variables in the rules is incorrectly numbered and so cannot be ground. When this happens obviously the condition in question cannot be used to query the knowledgebase and so no queries and responses can be displayed. If, on the other hand, the reports are merged, several things happen: (i) The merged report will be displayed in the Merged Report Window. (ii) The Save Merged Report option will be enabled. (iii) The number of rules that have succeeded and the number of rules that have failed will be displayed in the lower panel. (iv) The results of querying the knowledgebase are of course displayed in the Queries and Responses tab.

Both of these options are also available on the toolbar.

HELP MENU

User Manual Displays this user manual, and explanations of the error codes. If you require information about how to read or write fusion rules, information about the knowledgebase, or information about any other aspect of a fusion rules application you should consult the other chapters of this guide.

About Displays authorship information.

Chapter 7

The rule tool manual

7.1 Overview

The rule tool has two basic functions: to make it easier to write fusion rules, and to provide some elementary automated support for engineering a set of rules.

Fusion rules are written in a very simple form of XML, exploiting only the features of elements, their text content, and attributes. Nevertheless, a set of rules marked up in XML is not easy to read, and therefore not easy to edit. So in order to ease the burden of writing or editing a set of rules, a tool with the usual graphical user interface features has been provided. Using this tool you can write new rules and new sets of rules; edit an existing set of rules; insert new rules into existing rule sets; delete rules; and renumber the rules in a rule set. All these can be done simply by selecting items from combo boxes, typing in textfields, or dragging and dropping pieces of text. This means that you can in fact write fusion rules even if you do not know any XML. But note that in writing a set of rules you are attempting to construct a merged report, something which is an XML document; so in order to construct a coherent, meaningfull set of rules you must understand the basic mathematical idea of a tree, since XML documents must strictly observe that structure.

There is no algorithm for writing a correct set of rules. It is possible, however, to include some automated tests that check the "health" of a set of rules, including the fact that, if executed, they would construct a merged report with a coherent tree structure. The rule tool also allows you to conduct some other tests: for example, it duplicates the checks conducted by the fusion tool that the constraints on foundational rules and Initialize actions have been observed. (See section 9.6.2)

Details of these and other tests are described below (7.3).

The basic workflow of the rule tool is as follows: if you are editing an existing set of rules, load the rules. Before either creating a new rule or editing an existing rule you must load the DTD for the output merged report. This generates a list of all the complete branches in the merged report (for example `weathereport/temp/max`) which can then be used via drag and drop functionality to create or edit rules in the editing dialogs. Once you have done this, you can then choose to create or edit a rule. This involves navigating a sequence of three dialogs: the first allows you to select an individual condition or action to edit; the second allows you to select an individual argument to edit from this condition or action; the third allows you to edit the selected argument. By navigating back and forth between these dialogs as required, you can edit the selected rule in any way you wish.

If, on the other hand, you only wish to check a set of rules, then simply run the health check. You can also

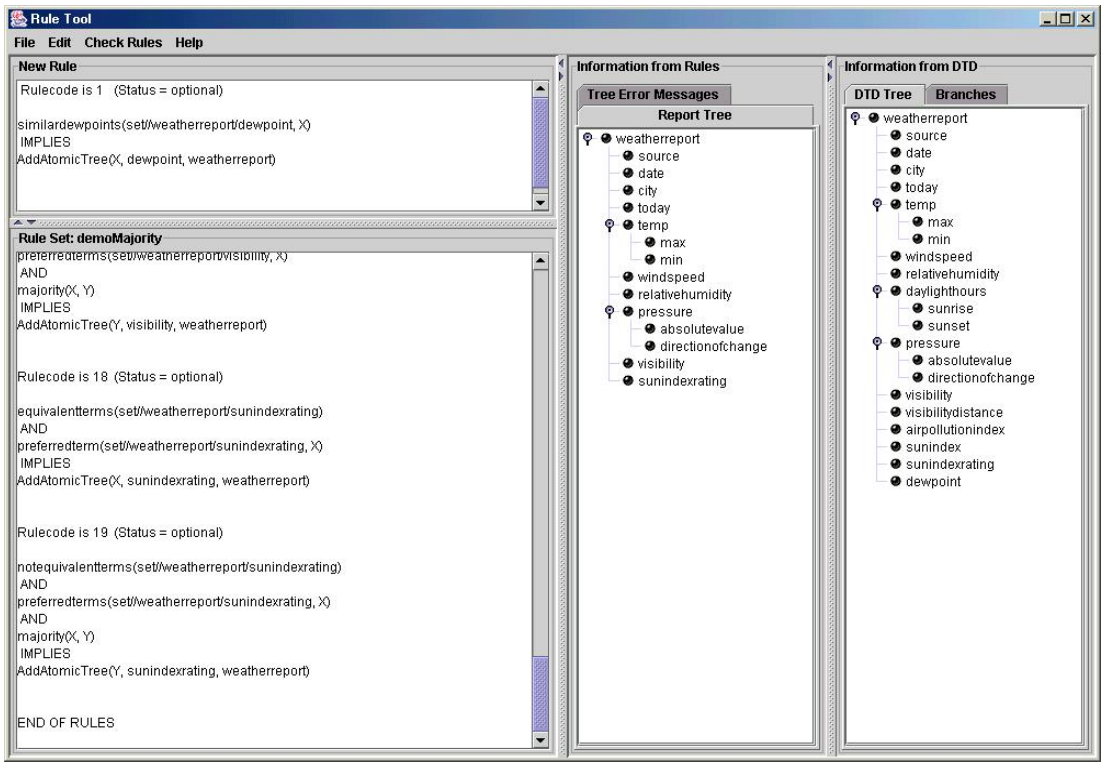


Figure 7.1: The rule tool for editing fusion rules. A new rule for merging textentries for *dewpoint* has been created and displayed in the upper left window. The rule set being edited (a set of weather rules) is displayed in the lower left window. The middle right window displays the structure of the output report that would be created by the rules if they were executed, while the far right window displays the structure of the output report as defined by the output report DTD. In this case the envisaged output report has tags for *visibilitydistance*, *airpollutionndex*, *sunindex*, and *dewpoint* that so far no rules have been written to deal with.

load the output report DTD and use the Display Merged Report Tree option to check that the structure of the output report is as intended.

7.2 Layout of the rule tool GUI

The rule tool has four main panels (see figure 7.1) and three main dialog boxes. The dialogs are used to edit or to write rules.

New Rule panel This panel displays a new rule once it has been written. The rule can subsequently be added to an existing set of rules or used to start a new rule set.

Rule Set If an existing set of rules is loaded for editing, it will be displayed here. Similarly, if the user chooses to create a new set of rules with a newly written rule, the set will be displayed here.

Information from Rules This panel displays the results of the various "health checks" that can be carried out on the rules. It also displays the output report tree, if any, that the rules would in fact construct.

Information from DTD This panel displays the tree that, according to the DTD for the merged report, the rules *ought* to construct.

Figure 7.2: The rule tool File menu.

Edit Rule Dialog This is the main dialog that allows the user to edit or create a rule.

Add Argument Dialog This is a subsidiary dialog that allows the user to add, remove, or alter the order of the arguments of an action or condition.

Edit Argument Dialog This is a subsidiary dialog that allows the user to edit an argument.

Further details of what these panels and dialogs do are given in the following section.

7.3 Rule tool operations

We now describe briefly the functions of the various menu options.

FILE MENU

Open Rules Opens a set of rules and displays them in the Rule Set panel. If the file selected is not an XML file, or if it is an XML file but does not conform to the rulefile DTD, then an error will result and the rules will not be loaded.

Save Rules Select this option to save the rules as XML. Note that when a set of rules is saved as an XML document the rule tool adds a document type declaration to an external DTD (`rulefile.dtd`). The path used will be the *absolute* path to the directory `demonstration/rules` where the DTD is located. This means that should you alter the position of this directory in your filestore, or should you alter the location of the file `rulefile.dtd`, then the rules saved using the rule tool will not load into the fusion tool. In this case, the simplest solution is to edit the XML file directly, so that the path in the document type declaration is correct.

Exit Quits the application.

EDIT MENU

Create New Rule This option launches the Edit Rule Dialog. For details of this dialog see below.

Edit an Existing Rule This option also launches the Edit Rule Dialog. However, before doing this the user is prompted to enter the rule code of the rule she wishes to edit.

Delete Rule This option allows the user to delete one or more rules by specifying their rule codes. Multiple rules can be specified using hyphens and/or commas, as for example $3, 6, 8, 15-21, 34$. But note that the tool contains no undo facility. Once you have deleted a rule it is lost. However, if you do delete a rule and then decide you want to keep it, simply omit to save the rules and either quit the rule tool or load another set of rules. In this case the rules in the XML file will be unchanged and you will have lost nothing.

Figure 7.3: The rule tool Edit menu.

Figure 7.4: The rule tool Check Rules menu.

Reposition Rule Rules are executed in the order in which they occur, not according to the order of their rule codes. If you wish to change the position of a rule this option provides you with a dialog box in which you enter the rule code of the rule to be repositioned, and the rule code that would place the rule in the position desired. The rule tool will automatically renumber any rules that need to be renumbered as a result of this. For example, if you wish to move rule 8 to the position occupied by rule 12, then simply type in 8 and 12. The existing rule 12 will then be renumbered as 13, subsequent rules being renumbered if required.

Create NEW RuleSet with Rule If a new rule has been created the user can decide to create an entirely new set of rules with it, as opposed to adding it to an existing set. Selecting this option will result in the new rule being displayed in the Rule Set panel, and further new rules can then be added to it.

Insert Rule into EXISTING RuleSet If a set of rules has already been loaded into the rule tool, and a new rule has been created, this option allows the user to insert the new rule into the rule set at a position specified by the user.

Renumber Rules This option will renumber a set of rules consecutively, starting from 1.

CHECK RULES MENU

Load Output Report DTD This option allows the user to select the DTD for the output merged report. This is then used to construct the tree structure that the merged report ought to have. This tree is displayed in the Information from DTD panel. In addition, on another tab in this panel, the set of complete branches in the merged report are displayed. This list will also be displayed in the dialogs to create or edit rules, where the branches can be dragged and dropped to create the branches for variables and constants. Because the structure of the *input* reports may in fact be different from the structure of the *merged* report, after the DTD for the merged report has been selected the user

is prompted to select, if required, the distinct DTD for the input reports. If this is done, the output report tree remains unchanged, but the additional branches from the input reports will be added to the list of branches.

Health Check Selecting this option results in a number of checks being conducted on the rules. If the rules fail any of the tests then appropriate warning messages will be displayed. The tests are:

- The rules are checked to see that they conform to the following conditions concerning foundational rules and Initialize actions.
  1. All foundational rules must *preceed* all non-foundational rules. This is because the success of these rules is in one way or another essential to the construction of the merged report.
  2. Every set of rules must contain at least one rule with an Initialize action.
  3. Any Initialize action must occur in a foundational rule.
  4. A single rule may contain at most only one Initialize action.
  5. An Initialize action must be the *first* action in a rule. Quite often, for example, a single rule contains an Initialize action followed by several actions adding text or nodes to the the skeleton thus created. Clearly, if the order of these actions was reversed, then some actions would be seeking to add text and nodes to a skeleton that doesn't yet exist.
  6. For a similar reason, rules with Initialize actions should preceed all other rules (including, of course, foundational rules without Initialize actions). (Further details on the rationale for these tests can be found in section 9.6).
- The rules are checked to see that, for each leaf element in the output report, there is at least one rule that has that element as the target node of one of its actions. This offers a minimal adequacy condition that the rules in fact cover the output report. (See section 9.8 for more on this.)
- Often, though not always, rules are grouped into subgroups such that all the rules in the subgroup share the same target node in the output report. In these cases, though again not always, it is often the intention of the application developer that, for any given set of input reports, at most one rule in each subgroup should fire. (See section 9.7 for more on this.) Accordingly, this test groups rules into subgroups, and for each subgroup, conducts a check on the pairwise inconsistency of the rules. If any pairs are found not to be inconsistent, a warning message results.

It should be noted that in the case of all these tests, the results are purely for advisory purposes. If a set of rules fails any of them you are not prevented from using them in the fusion tool. Of course, if you ignore the checks on foundational rules and Initialize actions then the fusion tool will prevent you from merging any reports. On the other hand, if your rules fail, say, the "inconsistency" test (and their passing it is important to their correct behaviour), then that will not prevent the reports from being merged, but you may well get erroneous results.

Display Merged Report Tree The first option on this menu (Load Output Report DTD) displays the structure of the merged report that the developer $intends$ her rules to construct (as indicated by the DTD for the merged report). By contrast, this option reads the actions in the rules to try to construct the structure of the merged report that the rules $would$ construct if executed. This structure is then displayed in the Information from Rules panel. Comparison of the two trees side by side allows the user of the rule tool to determine very easily if the rules would in fact construct a merged report with the intended structure. If, on the other hand, the rules would not construct a structure that is a coherent tree, then no tree is displayed; instead, another tab will display a warning message.

HELP MENU

User Manual Displays this user manual. If you require information about how to read or write fusion rules, information about the knowledgebase, or information about any other aspect of a fusion rules application you should consult the other chapters of this guide.

About Displays authorship information.

CREATE/EDIT RULE DIALOG

This dialog box allows the user to add and delete conditions and actions, reorder conditions and actions, and edit conditions and actions. We explain here the functions of the available operations.

New (button) The New button occurs on both the conditions and the actions tab. It creates a new condition or action, adding that new condition or action as the last condition or action of the rule. Note that when you create a new condition or action in this way, it is given a dummy name, such as "condition2" or "action4" according to its position. Assuming you do not wish to include this name in the finished rule, you should select a new name using the Condition Name or Action Name combo boxes.

Delete (button) The Delete button deletes the selected condition or action. (i.e. the condition or action selected in the list on the appropriate tab.)

Up (button) The Up button moves the selected condition or action one position up, or earlier, in the rule's list of conditions or actions, unless the selected action is already in the highest position. In terms of the implementation, this means that the index of the selected condition or action is decremented by one. Note that the list selection stays with the condition or action that has been moved, so you can continue to move it to an earlier position in the rule by continuing to click on this button.

Down (button) The Down button moves the selected condition or action one position down, or later, in the rule's list of conditions or actions, unless the selected action is already in the lowest position. In terms of the implementation, this means that the index of the selected condition or action is incremented by one. Note that the list selection stays with the condition or action that has been moved, so you can continue to move it to a later position in the rule by continuing to click on this button.

Condition/Action Name (combo box) These two combo boxes allow you to select the name of a condition or action. In the case of conditions, although a list of useful names is included, you can of course enter your own name, which you will usually need to do. In the case of actions, the combo box is not editable, and you must select one of the twelve pre-defined names.

Condition Sign (combo box) This combo box allows you to set the sign of the condition, either as positive or negative. If the knowledgebase implements this feature, negative conditions are intended to fail whenever the corresponding positive condition succeeds, and vice versa. See section 9.4.2 for a further explanation of this feature.

Arguments (textfield) If the selected condition or action has arguments then they are displayed in this textfield. If the selected condition or action has no arguments then the textfield is empty.

Add/Edit (button) This button allows you to add arguments to the selected condition or action, or to edit any existing arguments of the selected action or condition. In either case, arguments can be added or edited using the Add Argument dialog that is displayed.

Rule (panel) This panel contains the Optional/Foundational radio buttons allowing you to select the status of the rule. It also contains a Cancel button, cancelling any changes you have made to the rule you have been editing. If, on the other hand, you are happy with those changes, then you should click the Submit button. If the rule you were editing was an existing rule, the ammended rule is simply included in it original position in the rule set. If it was a new rule, then it will be displayed in the New Rule panel.

ADD ARGUMENT DIALOG

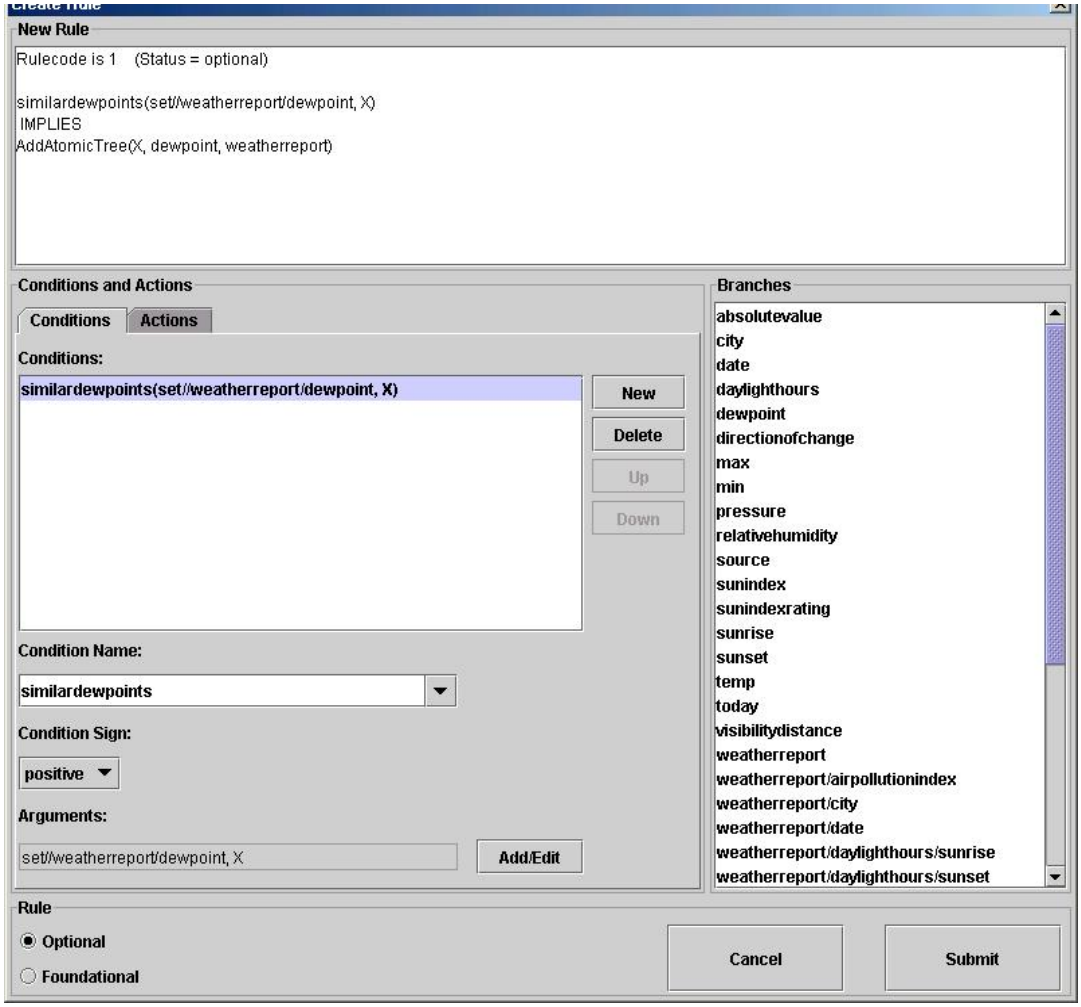


Figure 7.5: The highest level dialog for creating or editing a rule. The upper window displays the rule being edited or created. In this case the conditions tab has been selected, and it displays the rule's single condition. Buttons allow conditions to be added or deleted, and, if there is more than one, their order can be changed. Combo boxes allow the condition name and sign to be edited. The Add/Edit button allows new arguments to be added or existing arguments to be edited via further dialogs. The lower panel allows the rule's status to be selected, and the user can save or abandon the changes to the rule.

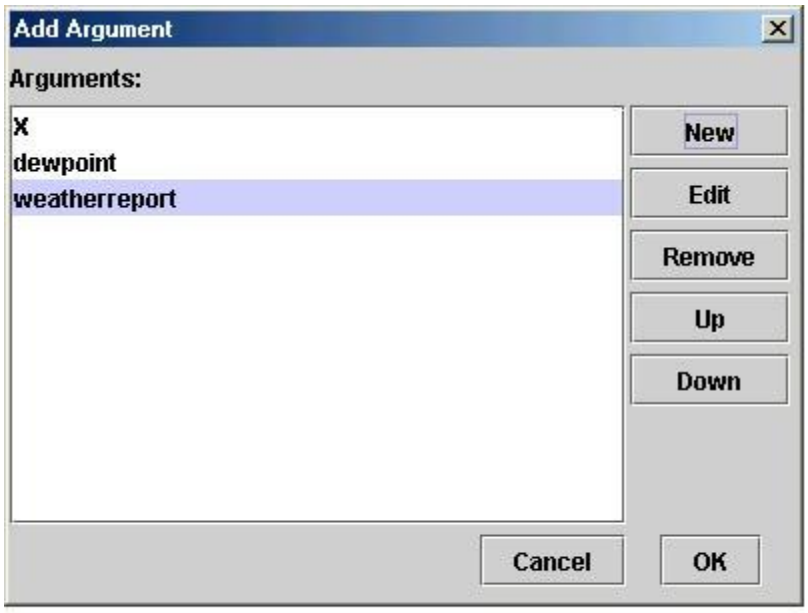


Figure 7.6: The second level dialog for creating or editing a rule. This dialog allows the user to add or remove the arguments of a condition or action, or to change the order of a condition's or an action's arguments. In this case, the selected condition has three arguments.

New (button) Clicking this button allows you to add a new argument to the selected condition or action. The Create Argument dialog is displayed. (See below.)

Edit (button) Clicking this button allows you to edit the selected argument. Again, this results in another dialog, Edit Argument, being displayed with the selected argument.

Remove (button) This button deletes the selected argument. Although there is no undo option, if you were editing an already existing argument then you can retrieve it by clicking the Cancel button and returning to the selected condition or action in its unammended form. However if the deleted argument was a new argument then you will not be able to retrieve it.

Up (button) This moves the selected argument one position up, or earlier, in the selected condition or action, unless the selected argument is already in the highest position.

Down (button) This moves the selected argument one position down, or later, in the selected condition or action, unless the selected argument is already in the lowest position.

Cancel (button) Cancels any changes to the selected condition's or action's arguments. Returns you to the main rule editing dialog.

OK (button) Submits any changes made to the selected condition's or action's arguments, and updates the main displays accordingly. Returns you to the main rule editing dialog.

EDIT/CREATE ARGUMENT DIALOG

Argument (textfield) The argument being edited or created should be entered here. For a variable this means specifying the branch. For a logical variable this means specifying the name of the variable, `X`, `Y`, etc., *not* the binding you might expect for the logical variable. For a constant this will be the

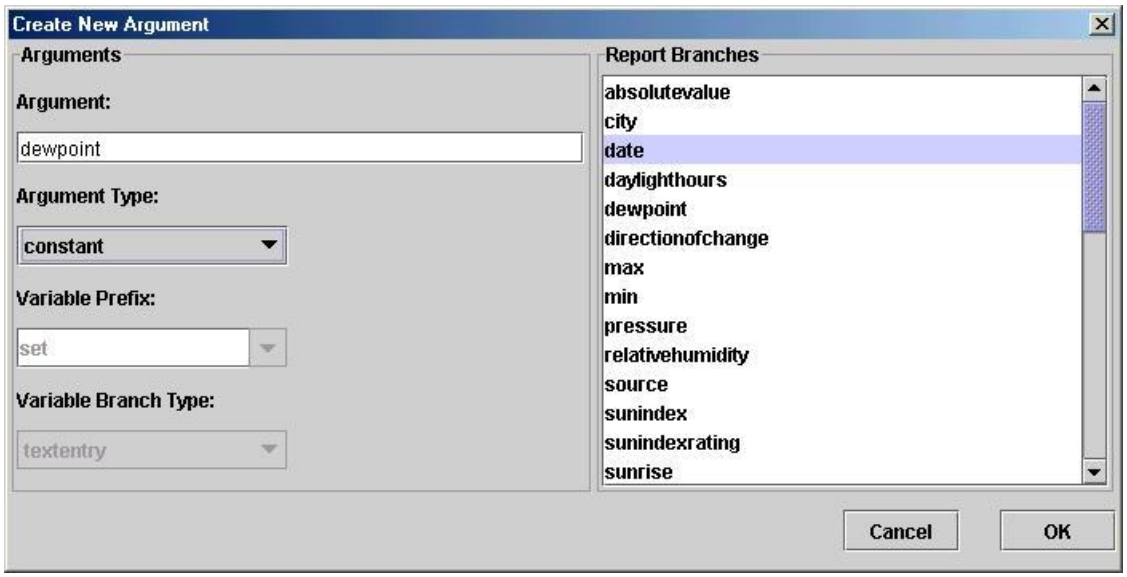


Figure 7.7: The third level dialog for creating or editing a rule. This dialog allows the user to create or edit arguments. The argument name is entered in the text field. If this name, as in this example, is a tag or a branch from the input or output reports, then it can be dragged and dropped from the list in the right hand window. The argument type, variable, logical variable, or constant, is selected using the first combo box. If the argument is a variable, the two lower combo boxes are enabled, allowing the user to specify if the variable is to be ground by an individual report, or by the set of all reports, and also if it is to be ground by a textentry or a subtree.

value of the constant, usually a branch, but it might be the numerical value of a voting threshold, or an accepted range within which values are deemed similar, etc. If a branch is to be entered, then you can drag and drop branches from the list displayed in the right panel of this dialog, thus saving effort but, more importantly, avoiding typos that would lead to later errors.

Argument Type (combo box) Select the type of the argument — variable, logical variable, or constant. It is important that you check that you have made the correct selection since, because the rule being edited or created is displayed in the logical notation, the display will not in fact indicate the argument type. (See section 9.3 for more on this.) In other words, make sure you have created the type of argument that you intended to create! Note that if you are editing an existing argument, then this combo box will automatically be set with the argument type. So if you want to check on the type of any argument, one way to do this (other than to view the rules using a browser or an XML editor) is to select the argument for display with this dialog.

Variable Prefix (combo box) If the argument type is set to "variable" then this editable combo box is enabled. You may either select the set variable prefix `set`, or enter the prefix for a source variable (1, 2, etc.).

Variable Branch Type (combo box) Again, if the argument type selected is "variable" then this combo box is enabled. Use it to specify what the variable is to be ground with; either a textentry or a subtree.

Cancel (button) Cancels any changes to the selected argument. Returns you to the previous Add Argument editing dialog.

OK (button) Submits any changes made to the selected argument. Returns you to the previous Add Argument editing dialog.

Chapter 8

The input reports

8.1 The form of the input reports

The salient feature about the input reports for a fusion application is that they must be in XML. This is because the fusion rules exploit the tree structure of the XML elements of the input reports in order to locate the textentries or subtrees used to ground the variables prior to querying the knowledgebase and constructing the merged report. It is important to note, however, that it is only the elements and their textual content that are exploited by the fusion rules. Input reports may of course include any legal XML features, such as attributes or processing instructions. But the informational content that is to be merged must appear as the textual content of XML elements and not, for example, as the values of attributes. This latter kind of content is simply ignored by the fusion rules and the fusion tool.

8.2 The content of the input reports

It is also important to note that the textual content itself must consist of single words (including numerical values, with or without unit suffixes) or short phrases, and not free text. This is because the textentries will be used to query the knowledgebase and so they must be suitably formulated. Unless your knowledgebase includes sophisticated text processing capabilities, words and recognized short phrases are likely to be the only suitable input. For example, a knowledgebase for a weather reports application could easily be built to handle textentries such as “rain”, “heavy rain”, or “light showers”. But it would be much harder to build a knowledgebase that could handle entries such as “a cold front will move in at 00.30 hours bringing with it cool, moist air. This will mix with tropical air from a warm front over southern Britain leading to torrential downpours and high winds.”

See example 8.2.1 for an input report from a weather reports application. (Further examples of the kind of textentries included in a weather reports application can be found in Chapter 12.)

Example 8.2.1 *An example of an individual weather report from a weather reports case study. This report*

conforms to the DTD in figure 8.1.

```
⟨weatherreport⟩
        ⟨source⟩ BBCi ⟨/source⟩
        ⟨date⟩ 10/12/02 ⟨/date⟩
        ⟨city⟩ London ⟨/city⟩
        ⟨today⟩ cloudy ⟨/today⟩
        ⟨temp⟩
                ⟨max⟩ 3C ⟨/max⟩
                ⟨min⟩ − 1C ⟨/min⟩
        ⟨/temp⟩
        ⟨windspeed⟩ 11mph ⟨/windspeed⟩
        ⟨relativehumidity⟩ 63% ⟨/relativehumidity⟩
        ⟨daylighthours⟩
                ⟨sunrise⟩ 7.13 ⟨/sunrise⟩
                ⟨sunset⟩ 15.51 ⟨/sunset⟩
        ⟨/daylighthours⟩
        ⟨pressure⟩
                ⟨absolutevalue⟩ 1021mb ⟨/absolutevalue⟩
                ⟨directionofchange⟩ rising ⟨/directionochange⟩
        ⟨/pressure⟩
        ⟨visibility⟩ good ⟨/visibility⟩
        ⟨airpollutionindex⟩ 3 ⟨/airpollutionindex⟩
        ⟨sunindex⟩ 1 ⟨/sunindex⟩
        ⟨dewpoint⟩ 2C ⟨/dewpoint⟩
⟨/weatherreport⟩
```

Input reports can, of course, contain more information (in the sense of more elements and their textentries) than is exploited by the rules. Any extra elements and their textentries will simply be ignored. There is, then, no reason why a set of rules must deal with every element in the input reports. Indeed, if an application deals independently with the elements (and their textentries) of the input reports, then it's usually easier to write a rule and develop the knowledgebase for each element in turn; test it, and then move on to the next element. There is no need to write all the rules first before testing any of them.

We should point out the none of the software included with this package includes any functionality to help with information extraction—for example, the fusion tool does not include any "wrapper" technology to extract wanted content from HTML Web pages and transform it into XML. It is up to you to get the content that you wish to merge into XML form. The fusion tool is, rather, intended to use the output of such technology, and in that sense complements it.

8.3 DTDs for input reports

Finally, if you want to use the rule tool, you will need a DTD for these reports. This is for two reasons: First, it allows the tool to check that you have written rules dealing with all of the elements with textual content in the input reports. As mentioned above, this may not always be the developer's intention, but it is helpful to know. Second, it allows the tool to construct strings corresponding to all the branches in the input reports. These can then be dragged and dropped as required for the variables and constants being created in editing or writing rules (thus helping to avoid typos and the errors they cause!). A DTD for the weather report in example 8.2.1 is given in figure 8.1.

```
⟨!ELEMENT weatherreport(source, date, city, today, temp,
                        windspeed, relativehumidity, daylighthours,
                        pressure, visibility, visibilitydistance,
                        airpollutionindex, sunindex, sunindexrating,
                        dewpoint)⟩
⟨!ELEMENT temp(max, min)⟩
⟨!ELEMENT daylighthours(sunrise, sunset)⟩
⟨!ELEMENT pressure(absolutevalue, directionofchange)⟩
⟨!ELEMENT source(#PCDATA)⟩
⟨!ELEMENT date(#PCDATA)⟩
⟨!ELEMENT city(#PCDATA)⟩
⟨!ELEMENT today(#PCDATA)⟩
⟨!ELEMENT max(#PCDATA)⟩
⟨!ELEMENT min(#PCDATA)⟩
⟨!ELEMENT windspeed(#PCDATA)⟩
⟨!ELEMENT relativehumidity(#PCDATA)⟩
⟨!ELEMENT sunrise(#PCDATA)⟩
⟨!ELEMENT sunset(#PCDATA)⟩
⟨!ELEMENT absolutevalue(#PCDATA)⟩
⟨!ELEMENT directionofchange(#PCDATA)⟩
⟨!ELEMENT visibility(#PCDATA)⟩
⟨!ELEMENT visibilitydistance(#PCDATA)⟩
⟨!ELEMENT airpollutionindex(#PCDATA)⟩
⟨!ELEMENT sunindex(#PCDATA)⟩
⟨!ELEMENT sunindexrating(#PCDATA)⟩
⟨!ELEMENT dewpoint(#PCDATA)⟩
```

Figure 8.1: A DTD for the weather reports used in a weather reports case study. Note that these reports contain slightly more information than those used in the demonstration application.

Chapter 9

Fusion rules

9.1 Overview

Fusion rules have a dual purpose: to query a knowledgebase and to use the results of those queries to construct a merged report. Their two major components reflect those two functions: conditions are used to query the knowledgebase and actions are used to construct the merged report. Both conditions and actions are in turn constructed out of variables, logical variables, and constants. Variables specify both what information is to be extracted from an input report, and from which report it is to be extracted. This information is then used to instantiate the fusion rules prior to querying the knowledgebase. Because either individual reports or the whole set of input reports may be specified as the source, variables are of two types: set variables and singleton, or source, variables. Constants take constant values which either denote targets in the merged report to which text entries and subtrees are to be added, or numerical values specifying, for example, voting thresholds or the number of nodes to be added to the output report, etc. Finally, logical variables indicate where the knowledgebase is expected to provide a binding that may be used either as input to a subsequent condition, or as output for the merged report.

To merge a set of input reports, we start with the background knowledge and the information in the reports to be merged, and apply the fusion rules to this information. For a set of input reports and a set of fusion rules, we attempt to ground each fusion rule with textentries from the reports, and then check whether all the conditions of each ground fusion rule are implied by the background knowledge, and if they are, then the ground actions of the rule are added to the actionlist (a list of actions that specify how the merged report should be constructed).

9.2 FusionRuleML

For reasons of persistence, fusion rules are marked up in XML. XML fusion rule files can then be loaded into the fusion tool where they are transformed into Java objects before being grounded with textentries or subtrees extracted from the input news reports, and then being used to query a knowledgebase and construct a merged report. The DTD for FusionRuleML is given in figure 9.1.

The root node of an XML document containing a set of rules is ⟨`rulefile`⟩. A rule file must contain at least one rule. There are of course elements for rules and their basic components: conditions, actions, variables, logical variables, and constants. The role of the components corresponding to these elements will be explained in the relevant sections of this chapter. Note that rules must have at least one action (there's

```
⟨!ELEMENT rulefile (rule+)⟩
⟨!ELEMENT rule (condition*, action+)⟩
⟨!ELEMENT condition (conditionname, (constant | variable | logicalvariable)+)⟩
⟨!ELEMENT action (actionname, (variable | constant | logicalvariable)*, constant+)⟩
⟨!ELEMENT constant (#PCDATA)⟩
⟨!ELEMENT variable ((source | set), branch)⟩
⟨!ELEMENT logicalvariable (#PCDATA)⟩
⟨!ELEMENT source (#PCDATA)⟩
⟨!ELEMENT set (#PCDATA)⟩
⟨!ELEMENT branch (#PCDATA)⟩
⟨!ELEMENT conditionname (#PCDATA)⟩
⟨!ELEMENT actionname (#PCDATA)⟩
⟨!ATTLIST conditionname sign (positive | negative) #REQUIRED⟩
⟨!ATTLIST branch extract (textentry | subtree) #REQUIRED⟩
⟨!ATTLIST rule code CDATA #REQUIRED status (foundational | optional) #REQUIRED⟩
```

Figure 9.1: The DTD for the fusion rules.

no point having a rule that does nothing), and zero or more conditions. Conditions must have at least one constant, variable, or logical variable. Actions must have at least one constant—every action seeks to attach something to some node in the merged report, and this *target* node is specified by this constant. The exception to this are those actions that build the initial structure of the merged report. In that case, the single constant specifies that structure. All other actions require a further variable, logical variable, or constant specifying what is to be added to the target node.

In addition to these elements, there are elements for condition names and action names. Queries are individuated using the ⟨`conditionname`⟩ textentry, allowing them to be dealt with correctly by the knowledgebase. The ⟨`actionname`⟩ textentry tells the action engine which of the twelve predefined actions is to be executed. There are also ⟨`source`⟩ elements which are used to specify if a variable refers to an individual report by containing a textentry with a natural number. This number results in the corresponding report, as loaded into the fusion tool, being used to ground the variable. For example, if the textentry for the source element was '3' then the third report loaded would be used to ground this variable. (Note that to take advantage of this feature you will need to select the input reports *individually* rather than collectively as a group; this is because if multiple files are selected the file chooser dialog will simply load the files in the order in which they occur in the directory.) Alternatively, if a variable is to be ground using all of the reports loaded, the ⟨`set`⟩ element is given the textentry '1'. ⟨`variable`⟩ elements have another child element, ⟨`branch`⟩, specifying the input report branch from which the textentry or subtree is taken and used to ground the variable.

The fusion rule mark-up also contains a number of attributes. Rule elements have attributes `code` and `status` specifying the rule code of the rule (a natural number), and the rule's status—either foundational or optional. ⟨`branch`⟩ elements have an `extract` attribute whose value specifies if the variable is to be ground with a textentry, or a subtree of the input report. Finally, ⟨`conditionname`⟩ elements have a `sign` attribute specifying if a rule is positive or negative. If the value of this attribute is negative the query involving this condition should fail if the query involving the same condition with a positive sign attribute succeeds, and vice versa. Whether this is so depends, of course, on the knowledgebase being correctly engineered.

A simple rule, marked up using these tags, is given in example 9.2.1.

Example 9.2.1 *An example of a rule from the weather reports application provided as a demonstration to run with the fusion tool. The rule is used to merge the textentries for today's weather. It contains two conditions and an action. The first condition checks if the textentries all belong to the same equivalence class (e.g. "sun", "sunny", "bright", "sunshine", etc.). If they do, the second condition selects the preferred term from amongst the terms in that equivalence class and uses it to bind the logical variable,* `X`*. The action then attaches this binding as a leaf to the branch* `weatherreport/today` *specified by the*

constant in the action `AddText`.

```
⟨rule code = "2" status = "optional"⟩
    ⟨condition⟩
        ⟨conditionname sign = "positive"⟩equivalentterms⟨/conditionname⟩
        ⟨variable⟩
            ⟨set⟩1⟨/set⟩
            ⟨branch extract = "textentry"⟩weatherreport/today⟨/branch⟩
        ⟨/variable⟩
    ⟨/condition⟩
    ⟨condition⟩
        ⟨conditionname sign = "positive"⟩preferredterm⟨/conditionname⟩
        ⟨variable⟩
            ⟨set⟩1⟨/set⟩
            ⟨branch extract = "textentry"⟩weatherreport/today⟨/branch⟩
        ⟨/variable⟩
        ⟨logicalvariable⟩X⟨/logicalvariable⟩
    ⟨/condition⟩
    ⟨action⟩
        ⟨actionname⟩AddText⟨/actionname⟩
        ⟨logicalvariable⟩X⟨/logicalvariable⟩
        ⟨constant⟩weatherreport/today⟨/constant⟩
    ⟨/action⟩
⟨/rule⟩
```

Given that fusion rules are marked up in XML then one way to write or edit a set of rules is to write some XML using your favourite XML editor (or any plain text editor). However, if you would prefer not to do this the rule tool provides a graphical user interface with a number of combo boxes, textfields and radio buttons allowing you to create or edit rules while avoiding any direct contact with XML. Refer to Chapter 7 for instructions on how to use this tool.

Even if you are fluent in XML it's not easy to discern the content or purpose of this rule. For this reason, we have also developed an alternative notation which better displays the logical form of a fusion rule.

9.3 A Logical Notation for Fusion Rules

We look, first, at the components of a fusion rule, and how they are represented; then we will see how they can be combined to represent a rule. Variables indicate which textentry (or textentries, or subtrees) is to be extracted to ground or instantiate a condition so that it in turn can be evaluated by the knowledgebase. Variables have two components: (1) a number prefix indicating which individual input report is to be used to ground or instantiate the variable. Alternatively, if (as is more likely) we wish to collect textentries from *all* of the input information sources, we indicate this by using the prefix `set`. In either case, this prefix is followed by a double slash "//". (2) a sequence of tag or element names separated by a forward slash, "/", indicating the branch in the report from which the textentry or subtree is to be taken. So for example, suppose we wished to merge some weather reports that had a root node ⟨`weatherreport`⟩ and a child element ⟨`today`⟩ that tagged the textentry for today's weather. If we wished to use the textentry for today's weather from the second report to be loaded we would use the variable:

```
2//weatherreport/today
```

Alternatively, if we wished to extract the textentries for today's weather from all of the input information sources, then we would use the variable:

```
set//weatherreport/today
```

Note that this logical notation does not indicate if a variable is to be ground by a textentry or a subtree.

Constants are not ground or instantiated by textentries; rather, as their name suggests, they are used to represent constant values. These may be numerical values representing, say, voting thresholds or acceptable ranges for values such as temperatures. However they are most ofen used to refer to the branches or to the individual tags or elements of the output merged report. In this case they are used to indicate to which part of the merged output report a particular piece of merged information is to be attached, and so typically (though not exclusively) they occur in actions rather than conditions. A constant, then, usually specifies either a branch or an individual tag in a report. Thus the constant

```
weatherreport/temp/max
```

specifies the branch in a weather report whose leaf node is the element for the maximum temperature.

The knowledgebase not only evaluates whether a condition is true, it also provides information to be included in the merged output. So for example, we might wish to know which of all of the textentries for today's weather is the one that occurs most commonly, and then use that textentry in the merged output. To do this the fusion rule must contain a *logical variable* which is provided with a *binding* by the knowledgebase. This binding will then be used either as input to a further condition, or as input to an action so that it can be included in the merged output. We have adopted a convention, influenced by Prolog, that logical variables are indicated by uppercase letters, "W", "X", "Y", "Z", etc., though in fact nothing in the fusion tool implementation requires this.

Finally, we can combine variables, constants, and logical variables, into conditions and actions. The name of a condition or action occurs immediately to the left of a pair of round brackets "()", and the variables, constants, and logical variables, within the scope of a condition or an action are placed within those brackets, separated by commas. Multiple conditions or actions within a rule are conjoined by "AND", while the conditions (if there are any) are prepended to the actions by "IMPLIES". In the following rule, all of the textentries for today's weather are checked to see if they belong to the same equivalence class (as determined by a knowledge engineer), e.g."cloud", "cloudy", "mostly cloudy", etc. If they do all belong to one class, the knowledgebase selects the preferred term, and uses it to bind the logical variable `X`. This in turn is used as the textentry to be added to the `weatherreport/today` branch in the merged output.

Example 9.3.1 *An example of a fusion rule displayed in logical form.*

```
EquivalentTerms(set//weatherreport/today)
AND PreferredTerm(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

This, of course, is the same rule that was displayed above using FusionRuleML in Example 9.2.1, but it's much easier to read using this notation.

It is important to note, however, that this logical notation for fusion rules has severe limitations. It is designed to make fusion rules easier to read by making their logical form and purpose more perspicuous. But the notation is in many respects not sensitive to the underlying components of the rules. We have already noted one limitation: the logical notation does not indicate if a variable is to be ground by a textentry or a subtree. A more serious limitation is that the notation is in fact invariant over the types of the arguments taken by conditions and actions. For example, the action `AddText` requires as its arguments a variable or a logical variable, and a constant. But this is not revealed by the logical notation, which will be the same if, for example, the action has instead two constants for arguments. (See example 9.3.2) In this case, the error would not be revealed until the rule was used to merge some reports, and an error of type 9 would result (see Appendix A).

Example 9.3.2 *Consider the following two actions:*

```
⟨action⟩
       ⟨actionname⟩AddText⟨/actionname⟩
       ⟨logicalvariable⟩Y⟨/logicalvariable⟩
       ⟨constant⟩weatherreport/today⟨/constant⟩
⟨/action⟩


⟨action⟩
       ⟨actionname⟩AddText⟨/actionname⟩
       ⟨constant⟩Y⟨/constant⟩
       ⟨constant⟩weatherreport/today⟨/constant⟩
⟨/action⟩
```

The first action has as its arguments a logical variable and a constant. This is correct. The second action has as its arguments two constants. This is incorrect. However, using the logical notation both of these actions would be represented as:

```
AddText(Y, weatherreport/today)
```

This illustrates the limitations of using the logical notation as a guide to the underlying nature of the rules.

This is not meant to be a criticism of the logical notation. Its intended purpose, to repeat, is simply to provide an aid to make a set of rules easier to understand. It is not meant to be an authoritative guide to the underlying nature of the rules. For this reason the fusion tool also displays the rules in their XML form, and this should be consulted whenever you wish to verify that nature.

9.4 Conditions

The function of conditions is to query the knowledgebase (and to return the results of those queries). In writing the conditions for fusion rules a number of issues arise. We deal with them in the following subsections.

9.4.1 How conditions are ground

Before querying the knowledgebase the variables in a condition must be ground with textentries or subtrees from the input reports. The variable branch specifies from *where* in the reports the information is to be taken. *What* information is to be extracted is specified by the value of the variable's `extract` attribute: either "textentry" or "subtree". From *which* reports that information is to be taken is, as we have already seen (section 9.2), specified by the ⟨`set`⟩ or ⟨`source`⟩ element of the variable. If it's a "set" variable, then a comma-separated list of the textentries from all the reports is constructed. If it's a "source" variable, then only the textentry from the individual report specified by the textual content (1, 2, 3, etc.) of the ⟨`source`⟩ element is used.

Example 9.4.1 *Some examples of conditions from the weather reports demonstration case study:*

```
SameCity(set//weatherreport/city)
SameDate(set//weatherreport/date)
SimilarHumidities(set//weatherreport/relativehumidity, X)
EquivalentTerms(set//weatherreport/today)
PreferredTerm(set//weatherreport/today, X)
UsePreferredSource(set//weatherreport/source, set1//weatherreport/visibility, X)
```

Given a suitable set of input reports, these would be ground as:

SameCity([Birmingham, Birmingham, B′ham, Birmingham])
SameDate([10/12/02, 10thDec2002, 10.12.02, 10thDecember2002])
SimilarHumidities([67%, 67%, 72%, 65%], 65 − 72%)
EquivalentTerms([rain, wet, heavyrain, rain])
PreferredTerm([sun, sunshine, sunny, bright], sun)
UsePreferredSource([BBCi, BBCLDN, Sky, CNN], [good, good, moderate, good], good)

All of the conditions in example 9.4.1 used "set" variables. This is typical since usually we want to merge the information from all of the input reports. But it is important to note what happens if individual "$source$" report variables are used instead of a "set" variable. Given that there are only four reports in the demonstration case study, we could merge all of them by using four individual "source" variables instead of one "set" variable. Of course, as this is more cumbersome we would never do this, but the point is that from a logical point of view we are asking exactly the same query, it's just that we are asking it in a slightly different way. This means that if we are to get the same answer (as clearly we should), then the ground condition must be parsed in the same way, to produce the same query.

But this creates a problem. If a number of "source" variables are used instead of a single "set" variable then there will be a discrepancy between the arity of a condition as defined in a particular fusion rule, and the arity of the same condition as defined in the knowledgebase. (Example 9.4.2 should clarify this point.) But, if the knowledgebase is to function correctly, a query must be constructed from the ground condition that has the *same* arity as the condition defined in the knowledgebase. The important point to note here, though, is that the fusion tool will not do this. This is because the fusion tool is designed so that any kind of external resource may be used as a knowledgebase; hence, the correct format for queries to that knowledgebase cannot be known in advance of that choice. Thus, parsing the ground conditions so as to get a query with the correct arity, as defined by the knowledgebase, is the responsibility of the interface to the knowledgebase. The moral, to repeat, is: if you anticipate using individual report ("source") variables, then you must make sure that the interface to the knowledgebase correctly deals with them when parsing queries. A fuller explanation of these issues can be found in Chapter 10.

Example 9.4.2 *Suppose we wish to merge the textentries for today's weather from exactly four input weather reports. We could do this in two different ways, either using a rule with four "source" variables or, as is normal, by using a rule with just one "set" variable. The resulting two conditions are as follows:*

EquivalentTerms(1//weatherreport/today, 2//weatherreport/today,
3//weatherreport/today, 4//weatherreport/today)

EquivalentTerms(set//weatherreport/today)

Note that as specified by the fusion rules, these two conditions have different arities: the arity of the first is four, and the arity of the second is one. But, intuitively, these two conditions are asking the same question. So, despite their different arities, they must be parsed by the interface to the knowledgebase to yield the same query:

EquivalentTerms([rain, wet, heavy rain, rain])

This will, of course, enable them to get the same answer.

We have discussed, so far, what happens if a condition contains a variable that is to be ground by a textentry (either from a single report, or from the set of all input reports). But what happens if it contains a variable

that is to be ground by a $subtree$? The answer is that the fusion tool takes the subtree whose root element is the tag specified by the last segment of the variable's branch, and parses that subtree into a special string representation that is used to ground the variable. For an illustration of this see example 9.4.3.

Example 9.4.3 *Suppose the following input report is the first to be loaded into the fusion tool.*

```
⟨weatherreport⟩
        ⟨source⟩The Weather Channel⟨/source⟩
        ⟨date⟩4/5/04⟨/date⟩
        ⟨city⟩Glasgow⟨/city⟩
        ⟨temp⟩
                ⟨maxtemp⟩14C⟨/maxtemp⟩
                ⟨mintemp⟩4C⟨/mintemp⟩
        ⟨/temp⟩
⟨/weatherreport⟩
```

Then the variable

```
⟨variable⟩
        ⟨source⟩1⟨/source⟩
        ⟨branch extract = "subtree"⟩weatherreport/temp⟨/branch⟩
⟨/variable⟩
```

would be ground by the string `temp(maxtemp(14C), mintemp(4C))`.

In the case where the variable that is to be ground by a subtree is also a "set" variable, then the grounding takes the form of a comma-separated list of these string representations.

It must be admitted that it would be rather cumbersome for the knowledgebase to deal with such strings, at least if the subtrees were large, and especially if there were long lists of them. For this reason it is not anticipated that variables occurring in $conditions$ will often be ground with subtrees. Rather, the intention of the "subtree" attribute value is that it should be exploited by $actions$ whose purpose it is simply to extract parts of the input reports and add them directly to the output report. Some examples of this are given in Sections 9.5.5, 9.5.6, 9.5.11, and 9.5.12.

There is a final, but important, point to note about how the fusion tool grounds conditions. We have seen what happens if a report has a textentry with which to ground a variable; but what happens if it does not? That is, what happens if a condition variable specifies a branch, say `weatherreport/visibility`, but there is no such branch in one of the input reports?[1] The answer is that the fusion tool grounds the variable with a special textentry, `null`. This denotes that it is unknown whether a value exists, and should not be confused with values such as "zero" or "none".

This has important implications for the knowledgebase designer for at least two reasons: (1) because a `null` should not be understood as a special kind of value the knowledgebase should not treat `null`s in the same way as it treats other textentries. For example, `null`s should be discounted from lists of textentries if those textentries are being counted for votes. (2) If *all* of the textentries are ground to `null` this can lead to unwanted consequences when rules exploit the idea of negative conditions. The possible problems that can arise will be discussed in the next section (9.4.2).

[1] A common reason why this happens is that a branch has been incorrectly specified in the rule. If a `null` occurs unexpectedly then check that the appropriate variable branch is correct.

9.4.2 Negative conditions

As we saw when discussing the fusion rule ML (section 9.2, above), ⟨conditionname⟩ elements have a sign attribute whose values are "positive" and "negative". If this attribute has the value "negative", the intended effect is that the condition should succeed if and only if the same condition with the value "positive" should fail. However, for that to be the case depends on the knowledgebase being implemented accordingly. There is no need to do this, of course, if you do not intend exploit this feature of the fusion rules. In that event, you should simply assign the value "positive" to all conditions because negative values result in a "Not" being prepended to the condition as it is displayed in logical form in the fusion tool.

If you do decide to implement this feature of the fusion rule ML think carefully before using it, for it may have unintended consequences. In particular, it may result in a rule succeeding when the intention is that it fail. This situation may arise when *all* of the textentries for a particular input report element are ground with null. Typically, a developer will design a knowledgebase so that in this situation the condition, and so the rule, will fail. But if the rule in question is one of a pair such that, if the first rule fails, the second rule is designed to succeed by default, as it would do if it contained a negative condition, then we may well end up with the situation where the second rule adds a null as textentry to the output report. Presumably this is not the intended outcome!

To avoid this, the conditions of a subgroup of rules that belong together should be formulated so that that subgroup of rules can all *fail*, and no action be taken. If that is the intended behaviour then negative values for the sign attribute should not be used. Rather, a distinct predicate, with the appropriate behaviour, should be defined.

As an instance of this consider Example 9.4.4: if the set variable that is the argument to EquivalentTerms in rule 12 is ground with a list of textentries that are all null, then the predicate will fail. This means, of course, that were Not EquivalentTerms to be used with the same set variable as an argument in rule 13, it would succeed, leading to the risk that if other conditions in the rule were also to succeed, a null textentry would be added to the merged report. If this is deemed undesirable, a new predicate, NotEquivalentTerms should be defined that also fails if the set variable is ground with a list of textentries that are all null. Hence, if no input report contains a tag for today neither will the output report.

Example 9.4.4 *A pair of rules for merging weather reports dealing with the entry for today's weather. Rule 12 says that if all the textentries for today's weather are from the same equivalence class, the preferred term from that class is selected and that textentry is added to the* today *tag in the merged report. However, if none of the reports has a textentry for today's weather, and so the variable* set//weatherreport/today *is ground with a list of* nulls*, then rule 12 will fail, and rule 13, which uses a condition whose sign attribute is "negative", will succeed. The upshot will be that a* null *is added as the text entry in the merged report for today's weather.*

```
Rulecode is 12 (Status = optional)

EquivalentTerms(set//weatherreport/today)
AND PreferredTerm(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)

Rulecode is 13 (Status = optional)

Not EquivalentTerms(set//weatherreport/today)
AND Disjunction(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

If this is not the desired behaviour then rule 13 should be altered so that it no longer uses a negative value for the condition's sign attribute, together with the same predicate as rule 12. Instead, a new predicate,

`NotEquivalentTerms`, *is defined. This predicate succeeds if not all of the textentries are from the same equivalence class and not all of the textentries are* `null`.

```
Rulecode is 13 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND Disjunction(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

9.4.3 Conditions and their arguments

Unlike actions, which are built into the fusion tool, conditions are user-defined, and so it is not possible here to specify what kinds of arguments they should take. We can, however, make a couple of general remarks. First, what arguments are in fact required by a condition is $context\ dependent$. That is, a condition that requires, say, two logical variables when it occurs in one rule may require a variable and a logical variable when it occurs in a different rule. (See example 9.4.5)

Example 9.4.5 *One of a pair of rules for merging weather reports dealing with the entry for today's weather. Rule 13 says that if not all the textentries for today's weather are from the same equivalence class, the preferred form from the equivalence class of each textentry is selected, and the majority term, if any, is found from amongst those preferred forms and is added to the* `today` *tag in the merged report.*

```
Rulecode is 13 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND PreferredTerms(set//weatherreport/today, X)
AND Majority(X, Y)
IMPLIES AddText(Y, weatherreport/today)
```

In this first case, because we wish to conduct the vote amongst the preferred forms of all the textentries in the input reports, the condition `Majority` *takes two logical variables as its arguments. If, however, we wished to conduct the vote simply from amongst the textentries as they occur in the input reports, then the same condition,* `Majority`*, would instead require as its arguments a variable and a logical variable.*

```
Rulecode is 13 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND Majority(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

Second, conditions can be *overloaded*. In the first instance this is a matter of the same condition name being defined in the knowledgebase with a different arity. (See example 9.4.6) The knowledgebase will of course have to be developed to allow for this. How much work that involves depends on the kind of resource being exploited. A Prolog knowledgebase, for example, automatically seeks to unify the query with the matching goal, so no extra work is required in coding the interface beyond writing the appropriate Prolog code. Other resources, by contrast, may require the developer to add functionality specifically to deal with this in the Java interface.

Example 9.4.6 *An example of an overloaded condition from the weather reports demonstration application. The first rule occurs in the set of rules that resolves conflicts by using preferences over sources. The*

condition, `NotSimilarWindspeeds/1`*, simply succeeds if not all the windspeeds that ground the set variable are within some specified range (here defined in the knowledgebase); otherwise it fails. If it succeeds then a subsequent condition,* `UsePreferredSource`*, selects the textentry from the report with the most preferred source from amongst those reports that do have textentries.*

```
Rulecode is 9 (Status = optional)

NotSimilarWindspeeds(set//weatherreport/windspeed)
AND UsePreferredSource(set//weatherreport/source, set1//weatherreport/windspeed, X)
IMPLIES AddAtomicTree(X, windspeed, weatherreport)
```

The second rule comes from the rule set that resolves some conflicts by using majority voting. However, because it is not appropriate to resolve conflicts between numerical values by voting, condition `NotSimilarWindspeeds/2` *resolves the conflict by rejecting outlying values. So, although it too succeeds if not all the windspeeds are within some specified range, it also provides a binding for the logical variable* `X`*, consisting of the minimum and maximum values of the largest group of windspeeds that lie within the specified range. Thus in the context of this rule, the condition* `NotSimilarWindspeeds` *requires an extra argument, the logical variable, and so is overloaded.*

```
Rulecode is 9 (Status = optional)

NotSimilarWindspeeds(set//weatherreport/windspeed, X)
IMPLIES AddAtomicTree(X, windspeed, weatherreport)
```

In theory it should also be possible to overload conditions by defining conditions that are distinct, not in terms of their arity, but in terms of the kinds of arguments they expect. Presumably this would require some work in the Java interface to the knowledgebase. However, it is hard to envisage a need for this.

9.4.4 The relationship between conditions and a knowledgebase

There is a close relationship between the conditions of a set of fusion rules and a knowledgebase. From a syntactical point of view it is obvious that the names of the conditions should match the names of the top-level predicates defined in the knowledgebase; and the arity of the conditions should also match the arity of the corresponding predicates. But the relationship is in fact closer than this. This is because fusion rules can embody or contain knowledge about a domain of application, and in such cases they are thus themselves also a *part* of the knowledge required for merging. For example, a rule merging reports for today's weather may deal with conflict by using a voting function with a threshold, so that only if, say, at least 75% of input reports agree on a value will it be included in the output report. This threshold may be specified in the resource queried by the the fusion rule, but it may instead be explicitly specified (as a constant) in the fusion rule itself. Or consider another example: a fusion rule merging textentries for temperature may deem them to be not conflicting if they are all within a certain acceptable range, say 3C. Again, in this case the knowledge as to what constitutes an acceptable range of temperatures could be specified in the resource being queried, but it could equally be explicitly specified in the fusion rule itself. (see example 9.4.7)

Example 9.4.7 *Rule 8 merges the textentries for the maximum temperature as specified by a set of weather reports. If the temperatures are deemed to be within an acceptable range then the logical variable,* `X`*, is bound by a string specifying the interval from the smallest to the largest values (or just a single value if they are all the same), and this interval is added as the textentry for the maximum temperature in the merged report. However, given this way of writing the rule, the user of this application would have to inspect the*

resource being queried in order to find out what that acceptable range is.

```
Rulecode is 8 (Status = optional)

SimilarTemps(set//weatherreport/temp/max, X)
IMPLIES AddText(X, weatherreport/temp/max)
```

We can rewrite the rule so as explicitly to display the accepted range within which values for the maximum temperature are deemed not to conflict. This range, 3C, is specified in the rule by the addition of a constant. By checking the values in the input reports, the user of this rule can now easily see why the rule succeeds or fails when the rules are executed.

```
Rulecode is 8 (Status = optional)

SimilarTemps(set//weatherreport/temp/max, 3C, X)
IMPLIES AddText(X, weatherreport/temp/max)
```

For a given application domain there is thus some leeway as to what knowledge is included in the resource queried and what is included in the fusion rules, and for this reason it is better to consider both as constituting the knowledgebase for that application.[2] Moreover, because one of the virtues of fusion rules is that they make it clearer why the reports are being merged as they are, in general it is to be preferred that simple knowledge, such as acceptable ranges or voting thresholds, should be explicitly included in the fusion rules rather than buried in the resource being queried.

9.4.5 How many conditions should a rule have?

Consider a pair of rules whose function is to deal with the textentries for today's weather. The first rule deals with the case where all the textentries belong to the same equivalence class, as defined in the knowledgebase, by adding the preferred term from that equivalence class to the output report. The second rule deals with the situation where not all of the textentries belong to the same equivalence class by simply forming their disjunction, and adding that to the merged report.

Given that job description, there is at least two ways in which the conditions for the pair of rules could be implemented. The first way is the simplest: all of the functionality, as just described, for each rule could be combined into a *single* condition, as in example 9.4.8. Here, in the first rule the logical variable, X, gets bound by the preferred term, and then added to the merged report. While in the second rule the logical variable gets bound by the disjunction, and then that is added to the output report.

Example 9.4.8 *A simple version of a pair of rules for merging the textentry for today's weather. In rule* 2*, if all the textentries belong to the same equivalence class (as defined in the knowledgebase) then the logical variable* X *is bound by the preferred term from that class. In rule* 3*, if they do not all belong to that class, then their disjunction is constructed and used to bind* X*.*

```
Rulecode is 2 (Status = optional)

EquivalentTerms(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)

Rulecode is 3 (Status = optional)

NotEquivalentTerms(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

However, although this pair of rules would satisfactorily carry out their intended function, it is not easy to discern what that function is. The first rule makes no mention of selecting the preferred term from

[2] We shall, however, continue to use the term "knowledgebase" to denote only the external resource queried by the fusion rules. This should not be taken to imply that the rules are not themselves part of the knowledgebase any more than the fact that typically the term "human" is used to contrast human beings with animals implies that human beings are not animals.

amongst those in an equivalence class, and the second makes no mention of the fact that a disjunction of the nonequivalent textentries is to be added to the merged report. The user of these rules would thus be left scratching her head trying to work out *how* they are supposed to merge the textentries for today's weather. To find that out she would have to delve into the knowledgebase itself.

Instead, it would be much better if these rules were rewritten so as explicitly to display their purpose. This can easily be done by adding a condition to each, as in example 9.4.9

Example 9.4.9 *The pair of rules from example 9.4.8 rewritten so as to display more clearly how they will merge the textentries for today's weather.*

```
Rulecode is 2 (Status = optional)

EquivalentTerms(set//weatherreport/today)
AND PreferredTerm(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)

Rulecode is 3 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND Disjunction(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

As far as a rule's conditions are concerned, then, the rule engineer faces a choice between, on the one hand, making them as simple as possible and, on the other hand, making them more perspicuous or revealing of their function. In the case of Examples 9.4.8 and 9.4.9, although the former is simpler (and perhaps makes clearer the logical relationship between the rules), the latter surely makes it clearer how the information is to be merged and is thus to be preferred.

9.4.6 Ordering of conditions

The reason why the order of conditions within a single rule is important is because often the *output* (i.e. the bindings of logical variables) of one condition is required as the *input* for another condition. In cases such as these it is obvious that the former condition must *preceed* the latter condition. If the order is incorrect, so that a condition does not get the binding that it expects for a logical variable, then a knowledgebase error (Error 5, see Appendix A) results. Example 9.4.10 is a case in point.

Example 9.4.10 *A rule from the demonstration case study for merging weather reports dealing with the entry for today's weather. The first condition succeeds if the textentries for today's weather are not all from the same equivalence class. If that is so, the second condition attempts to resolve the conflict by selecting the textentry, if any, that has a majority of the occurrences in the reports. This condition expects a binding for the logical variable* `X` *which should be a list of the textentries for today's weather. However, that binding is not supplied. If it had a chance to execute, the third condition,* `PreferredTerms`*, would provide the required list (in the form of the preferred term from the equivalence class of each of the textentries in the list binding* `X`*). But* `PreferredTerms` *will not execute because an error will have already occurred when the second condition queried the knowledgebase.*

```
Rulecode is 3 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND Majority(X, Y)
AND PreferredTerms(set//weatherreport/today, X)
IMPLIES AddText(Y, weatherreport/today)
```

Here is the same rule with the conditions in the correct order.

```
Rulecode is 3 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND PreferredTerms(set//weatherreport/today, X)
AND Majority(X, Y)
IMPLIES AddText(Y, weatherreport/today)
```

In cases where the conditions of a rule do not share a logical variable it nearly always makes more sense for the conditions to observe a particular order, even if that is not strictly required for the execution of the rules. Consider a pair of rules designed to merge textentries for `relativehumidity`. The first rule deals with the case where there is no conflict; if all the values are within the accepted range, it simply finds the interval of the minimum and maximum values and attaches that to the meged report. If, on the other hand, not all the textentries are within the accepted range, the second rule selects the value from the input report with the most preferred source and includes that in the output report. (See example 9.4.11) In this case it clearly makes sense for the second rule to check first that the humidities are not in fact similar, and only if that is true should it then go on to find the textentry from the most preferred source and add it to the output report. If the order of the conditions in the second rule is reversed, it both makes the logic of the pair of rules harder to discern, and it involves some uneccessary computation. But note, however, that the rules would still execute in the same, satisfactory, manner. This is because, first, the logical variable `X` would receive the binding it expects; and, second, it would still be the case that at most only one of the pair would execute. If the first rule was executed then, although the first condition of the second rule would succeed, the second condition would fail, preventing the second rule from executing as well.

Example 9.4.11 *A pair of rules from the demonstration case study for merging weather reports dealing with the entry for relative humidity. The pair are designed so that for any given set of reports, at most only one rule will be executed; the first, if the humidities are all within an accepted range, the second if they are not.*

```
Rulecode is 10 (Status = optional)

SimilarHumidities(set//weatherreport/relativehumidity, X)
IMPLIES AddAtomicTree(X, relativehumidity, weatherreport)

Rulecode is 11 (Status = optional)

NotSimilarHumidities(set//weatherreport/relativehumidity)
AND UsePreferredSource(set//weatherreport/source, set//weatherreport/relativehumidity, X)
IMPLIES AddAtomicTree(X, relativehumidity, weatherreport)
```

If the order of the conditions in the second rule were reversed this would not affect the satisfactory operation of the pair of rules, but it would make their logic much harder to understand, and be computationally wasteful.

```
Rulecode is 11 (Status = optional)

UsePreferredSource(set//weatherreport/source, set//weatherreport/relativehumidity, X)
AND NotSimilarHumidities(set//weatherreport/relativehumidity)
IMPLIES AddAtomicTree(X, relativehumidity, weatherreport)
```

9.5 Actions

The purpose of fusion rule actions is to construct the output merged report. Because this report is an XML document it has a tree structure. The twelve actions that have been developed allow the merged

output report tree to be built up in a number of different ways. But however an action contributes to the construction of the merged report, it must specify at least two things: *what* is to be added, and *to where* it is to be added—the target—in the merged report. (The one exception to this is the Initialize action which, beause it specifies the initial structure of the merged report, does not specify a target.)

The choice of an appropriate action or actions to use for a given rule will depend on the actions of previous rules because an action can only add textentries or subtrees to the merged report tree if the specified target already exists. And whether the target exists depends, of course, on the outcomes of previous actions. Obviously this fact will impose certain constraints on the ordering of rules within a rule set and on the ordering of actions within a rule. (See Section 9.11 for further disucssion of this issue and examples.)

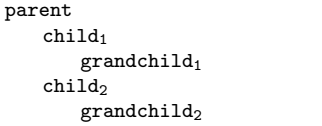
The twelve actions described in the sequel are built into the fusion tool. Without editing the source code it is not possible to add more actions. However, we believe these actions provide a considerable amount of flexibility for designing a set of rules for a given application. It is also worth pointing out that because merged reports consist only of XML elements and their textual content, all of these actions add structure of only that kind to the merged report. None of the actions add any other kind of XML features, such as attributes and their values, namespaces, or processing instructions, to the output report. So even if you are not familiar with XML, as long as you understand the idea of a tree, as used in mathematics, you should have no difficulty in understanding what the actions are trying to do.

In the following sections we describe the twelve built-in actions, explain what they do, specify both their arity and the types of arguments they take, and give examples of their use. The argument types appearing in brackets after the action name specify the legal argument types, and their positions, for each action. The vertical bar is used to signify disjunction.

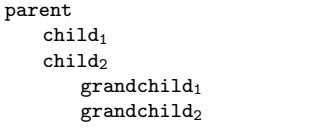
9.5.1 Initialize

Initialize(Constant) — Constant specifies the initial skeleton (i.e. the set of tags or elements, together with their structure) of the output merged report. The initial structure may be expanded by subsequent rules adding further nodes, textentries, or subtrees. The tree specified by the constant may be as minimal or elaborate as desired, but at a minimum it must specify a root node.

The tree structure is described using nested brackets in the following, intuitive, way: child nodes of a parent are enclosed within round brackets following the name of the parent; sibling nodes are separated by commas. Thus the constant $\texttt{parent}(\texttt{child}_1(\texttt{grandchild}_1), \texttt{child}_2(\texttt{grandchild}_2))$ represents the following tree.



And the constant $\texttt{parent}(\texttt{child}_1, \texttt{child}_2(\texttt{grandchild}_1, \texttt{grandchild}_2))$ represents the distinct tree shown below.



Whereas the constant $\texttt{parent}_1(\texttt{child}_1), \texttt{parent}_2(\texttt{child}_2, \texttt{child}_3)$ does not define a tree because no root node is specified for the two nodes with the greatest height, $\texttt{parent}_1$ and $\texttt{parent}_2$.

```
parent1
    child1
parent2
    child2
    child3
```

If the value of the constant does not represent a valid tree, then an error (Error 13, see Appendix A) will result and the reports will not be merged.

To take an example, if Constant was weatherreport(source, date, city, today, temp(max, min)), then the action Initialize(weatherreport(source, date, city, today, temp(max, min))) would produce a skeleton for the merged output report that looked like the tree in example 9.5.1.

Example 9.5.1 *An example of the skeleton of a merged report produced by an* Initialize *action with the constant* Initialize(weatherreport(source, date, city, today, temp(max, min))).

```
⟨weatherreport⟩
        ⟨source⟩⟨/source⟩
        ⟨date⟩⟨/date⟩
        ⟨city⟩⟨/city⟩
        ⟨today⟩⟨/today⟩
        ⟨temp⟩
                ⟨max⟩⟨/max⟩
                ⟨min⟩⟨/min⟩
        ⟨/temp⟩
⟨/weatherreport⟩
```

Special constraints apply to the occurrence and positioning of these actions. Please refer to section 7.6.

9.5.2 AddText

AddText(**Variable** | **LogicalVariable**, **Constant**) — the textentry that instantiates the variable or the logical variable is attached to the element or tag at the end of the branch specified by the constant. Constant therefore specifies the target node. For example, if the text entry used to ground or bind the logical variable was rain or showers and the value of the constant was weatherreport/today, then this action would add the textentry to the tag today at the end of the branch weatherreport/today in the output report.

Example 9.5.2 *Suppose in the action* AddText(X, weatherreport/today) *the logical variable* X *is bound by the textentry* rain or showers. *This would make the following addition to the merged report in example 9.5.1:*

```
⟨weatherreport⟩
        ⟨source⟩⟨/source⟩
        ⟨date⟩⟨/date⟩
        ⟨city⟩⟨/city⟩
        ⟨today⟩rain or showers⟨/today⟩
        ⟨temp⟩
                ⟨max⟩⟨/max⟩
                ⟨min⟩⟨/min⟩
        ⟨/temp⟩
⟨/weatherreport⟩
```

Note that if the branch specified by the constant does not exist in the structure of the merged report, as already constructed, then an error (Error 10) will result.

If the output merged report contains several branches with the *same* path, and you have a list of textentries that you wish to add, one to each branch, then the action `AddText` will simply add each text entry to the *first* such branch in the report. Repeatedly using this action would, then, result in each textentry overwriting the previous one. Obviously this is not the desired behaviour. In such cases you should use the action `RepeatAddText`. (9.5.9)

9.5.3 AddNode

AddNode(Constant$_1$, Constant$_2$) — the tag or element specified by Constant$_1$ is added to the merged output report as a child of the tag or element at the end of the branch specified by Constant$_2$. Constant$_2$ therefore specifies the target node. For example, if Constant$_1$ is the node `pressure` and Constant$_2$ is `weatherreport` then `pressure` would be added as a child of `weatherreport` in the merged report.

Example 9.5.3 *Consider the action* `AddNode(pressure, weatherreport)`. *This would make the following addition to the merged report in example 9.5.1:*

```
⟨weatherreport⟩
        ⟨source⟩⟨/source⟩
        ⟨date⟩⟨/date⟩
        ⟨city⟩⟨/city⟩
        ⟨today⟩⟨/today⟩
        ⟨temp⟩
                ⟨max⟩⟨/max⟩
                ⟨min⟩⟨/min⟩
        ⟨/temp⟩
        ⟨pressure⟩⟨/pressure⟩
⟨/weatherreport⟩
```

Note that the value of Constant$_1$ must be an individual node or element and not a branch. The value of Constant$_2$ may be, and often will be, a branch. Again, if the branch specified by the target node, Constant$_2$, does not exist in the structure of the merged report, as already constructed, then an error (Error 10) will result.

9.5.4 AddAtomicTree

AddAtomicTree(Variable | LogicalVariable, Constant$_1$, Constant$_2$) — the textentry that instantiates the variable or the logical variable is attached to the element or tag specified by Constant$_1$ to form an "atomic tree". This subtree is then added to the merged output report as a child of the tag or element at the end of the branch specified by Constant$_2$. Constant$_2$ therefore specifies the target node.

For example, if the text entry used to ground or bind the logical variable was `1021 − 1023mB`, the value of Constant$_1$ was `absolutevalue`, and the value of Constant$_2$ was `weatherreport/pressure` then this action would add the atomic tree ⟨`absolutevalue`⟩`1021 − 1023mb`⟨`/absolutevalue`⟩ to the end of the branch `weatherreport/pressure`.

Example 9.5.4 *Suppose that in the following action the logical variable* `X` *is bound by the textentry* `1021 − 1023mB`.

`AddAtomicTree(X, absolutevalue, weatherreport/pressure)`

This would make the following addition to the merged report in example 9.5.3:

```
⟨weatherreport⟩
        ⟨source⟩⟨/source⟩
        ⟨date⟩⟨/date⟩
        ⟨city⟩⟨/city⟩
        ⟨today⟩⟨/today⟩
        ⟨temp⟩
                ⟨max⟩⟨/max⟩
                ⟨min⟩⟨/min⟩
        ⟨/temp⟩
        ⟨pressure⟩
                ⟨absolutevalue⟩1021 − 1023mB⟨/absolutevalue⟩
        ⟨/pressure⟩
⟨/weatherreport⟩
```

This action is provided as a utility action since it simply combines the two previous actions, `AddText` and `AddNode`, into a single action. Note that, as before, the value for $\mathrm{Constant}_1$ must be a single node or element and not a branch. And note once more that if the branch specified by the target node, $\mathrm{Constant}_2$, does not exist in the structure of the merged report, as already constructed, then an error (Error 10) will result.

9.5.5 AddTree

AddTree(**Variable** | **LogicalVariable**, **Constant**) — The textentry that is the value of Variable or LogicalVariable represents a tree. This tree is then attached to the element or tag at the end of the branch specified by Constant. If the first argument is a variable then the tree that grounds Variable is the subtree of the input report tree whose root node is the tag at the end of the branch specified by the variable. This root node will be the child of the target node in the merged report, which is specified by Constant.

For example, if Variable was `3//weatherreport/regionalreport` and Constant is `weatherreport/regionalreports` then this action would take the third report, and add the subtree whose root node is `regionalreport` to the merged output report at the node `regionalreports`.

It is important to say something about the form of the textentry expected by the first argument for this action. (Similar remarks apply to actions `ExtendTree`, `AddTrees`, and `MultiExtendTree`.) That textentry should represent a tree in the manner shown in example 9.5.5.

Example 9.5.5 *Consider the following tree:*

```
⟨regionalreport⟩
        ⟨region⟩SouthEast⟨/region⟩
        ⟨maxtemp⟩14C⟨/maxtemp⟩
⟨/regionalreport⟩
```

If this tree is to be added to the merged report using the action `AddTree` *then the first argument to that action must have as its value a textentry of the form:*

```
regionalreport(region(SouthEast),maxtemp(14C))
```

It is envisaged that in most cases the first argument to this action will be a *variable* rather than a logical variable. This is because if a variable is used, and the `extract` attribute of the ⟨`branch`⟩ element takes the value "subtree" (as it should do), not the value "texentry", then the effect of this is that the fusion tool

will construct the correct string representation of the subtree whose root node is specified by the variable's branch, and this subtree can then be added to the merged report. However, it is possible to use this action in cases where its first argument is a logical variable. Such a case would have to involve the knowledgebase computing a query and providing a binding in the form of a textentry representing a subtree. Reconstructing a string representation of a subtree is not perhaps a typical function of a knowledgebase and so we do not think it very likely that this action (nor `ExtendTree`) will be used with a logical variable.

Given that we envisage that in most cases this action will take a variable as its first argument, it follows that the kind of situation we had in mind where this action might be used is where the application developer simply wants to add chunks of an input report straight into the output report, as in example 9.5.6.

Example 9.5.6 *Suppose that the third input report loaded into the fusion tool is as follows:*

```
⟨weatherreport⟩
        ⟨source⟩BBC London⟨/source⟩
        ⟨date⟩10th Feb 2003⟨/date⟩
        ⟨regionalreport⟩
                ⟨region⟩SouthEast⟨/region⟩
                ⟨today⟩cloud⟨/today⟩
                ⟨temp⟩
                        ⟨max⟩3C⟨/max⟩
                        ⟨min⟩ − 2C⟨/min⟩
                ⟨/temp⟩
        ⟨/regionalreport⟩
⟨/weatherreport⟩
```

And suppose that the merged report as so far constructed is as follows:

```
⟨weatherreport⟩
        ⟨source⟩BBC Wales⟨/source⟩
        ⟨date⟩10th Feb 2003⟨/date⟩
        ⟨regionalreports⟩
                ⟨regionalreport⟩
                        ⟨region⟩Wales⟨/region⟩
                        ⟨today⟩rain⟨/today⟩
                        ⟨temp⟩
                                ⟨max⟩5C⟨/max⟩
                                ⟨min⟩2C⟨/min⟩
                        ⟨/temp⟩
                ⟨/regionalreport⟩
        ⟨/regionalreports⟩
⟨/weatherreport⟩
```

Then the action

`AddTree(3//weatherreport/regionalreport, weatherreport/regionalreports)`

would lead to the following addition to the merged report:

```
⟨weatherreport⟩
        ⟨source⟩BBC Wales and BBC London⟨/source⟩
        ⟨date⟩10th Feb 2003⟨/date⟩
        ⟨regionalreports⟩
                ⟨regionalreport⟩
                        ⟨region⟩Wales⟨/region⟩
                        ⟨today⟩rain⟨/today⟩
                        ⟨temp⟩
                                ⟨max⟩5C⟨/max⟩
                                ⟨min⟩2C⟨/min⟩
                        ⟨/temp⟩
                ⟨/regionalreport⟩
                ⟨regionalreport⟩
                        ⟨region⟩SouthEast⟨/region⟩
                        ⟨today⟩cloud⟨/today⟩
                        ⟨temp⟩
                                ⟨max⟩3C⟨/max⟩
                                ⟨min⟩ − 2C⟨/min⟩
                        ⟨/temp⟩
                ⟨/regionalreport⟩
        ⟨/regionalreports⟩
⟨/weatherreport⟩
```

It is important, then, to note two things regarding this action if a variable is used as the first argument: (i) the variable in question must be ground by an individual input report and not by the set of all the input reports. In terms of the ruleML, this means that the variable should contain a ⟨source⟩ element which should have a natural number as a textentry. (ii) The extract attribute of the ⟨branch⟩ element must take the value "subtree", not the value "texentry". If the intention is to add subtrees from more than one input report, these subtrees should be specified separately using different actions. But if the intention is to add subtrees to the output report from all of the input reports, then action AddTrees may be used instead. (See Section 9.5.11.)

9.5.6 ExtendTree

ExtendTree(**Variable** | **LogicalVariable**, **Constant**) — The textentry that is the value of Variable or LogicalVariable represents a tree. This tree is then used to extend the merged report tree by attaching the descendants of the root node (but *not* the root node) of the tree to the element or tag at the end of the branch specified by Constant. As with action AddTree, if the first argument is (as envisaged) a variable, then the tree that grounds Variable is the subtree of the input report tree whose root node is the tag at the end of the branch specified by the variable. But unlike action AddTree, this root node itself will *not* be added to the merged report; instead, its children will be the children of the target node in the merged report, which is specified by Constant.

For example, if Variable is ground by the tree represented by 1//weatherreport/precipitation and Constant is given the value weatherreport/precipitation, then this action would take the first input report and add the child nodes (and all their descenants) of node ⟨precipitation⟩ to the end of the branch weatherreport/precipitation in the merged report.

As was the case with AddTree, if a variable is used as the first argument, the variable must be ground by an individual input report, and not the set of input reports (it must be a "source" variable, not a "set" variable), and the extract attribute must take the value "subtree" and not the value "textentry". Again, as was the case with AddTree, it is expected that the first argument of this action will be a variable and not a logical variable and that this action will be primarily used simply to add chunks of one of the input reports straight into the output report, as in example 9.5.7.

Example 9.5.7 *Suppose that the first input report loaded into the fusion tool is as follows:*

```
⟨weatherreport⟩
      ⟨source⟩BBCi⟨/source⟩
      ⟨date⟩10th Feb 2003⟨/date⟩
      ⟨city⟩London⟨/city⟩
      ⟨precipitation⟩
            ⟨type⟩snow⟨/type⟩
            ⟨amount⟩2cm⟨/amount⟩
            ⟨probability⟩0.6⟨/probability⟩
      ⟨/precipitation⟩
⟨/weatherreport⟩
```

And suppose that the merged report as so far constructed is as follows:

```
⟨weatherreport⟩
      ⟨source⟩BBCi⟨/source⟩
      ⟨date⟩10th Feb 2003⟨/date⟩
      ⟨city⟩London⟨/city⟩
      ⟨precipitation⟩⟨/precipitation⟩
⟨/weatherreport⟩
```

Then the action

`ExtendTree(1//weatherreport/precipitation, weatherreport/precipitation)`

would lead to the following addition to the merged report:

```
⟨weatherreport⟩
      ⟨source⟩BBCi⟨/source⟩
      ⟨date⟩10th Feb 2003⟨/date⟩
      ⟨city⟩London⟨/city⟩
      ⟨precipitation⟩
            ⟨type⟩snow⟨/type⟩
            ⟨amount⟩2cm⟨/amount⟩
            ⟨probability⟩0.6⟨/probability⟩
      ⟨/precipitation⟩
⟨/weatherreport⟩
```

9.5.7 AddAtomicTrees

AddAtomicTrees(Variable | LogicalVariable, Constant$_1$, Constant$_2$) — the textentry that instantiates the variable or the logical variable should be in the form of a comma-separated list. For each item in that list an "atomic tree" will be created, consisting of that item as a textentry together with the node specified as the value of $\mathrm{Constant}_1$. Each of these "atomic trees" will then be added to the merged report as the child of the node at the end of the branch specified by $\mathrm{Constant}_2$.

As was the case with action `AddAtomicTree`, the value of $\mathrm{Constant}_1$ must be an individual report node or element, and not a branch. And the branch which is the value of $\mathrm{Constant}_2$ must already be in the merged output report, or an error (Error 10) will occur.

An example of when this kind of action might be used is where a condition produces a list of textentries which will be added to the merged report. However, rather than displaying those textentries as a list, all added to a single node in the merged report, it is decided that the merged report would be clearer if each item in the list were displayed as a separate tagged textentry, the whole group being child nodes of a single parent.

Example 9.5.8 *The following example comes from the initial stages of a case study in bioinformatics. The objective of this case study was to take a group of proteins and determine their common molecular function. The output report was also supposed to display the keywords and phrases from amongst the functional annotations of the proteins in the grouping. Used in combination with a suitable set of input reports, and where the logical variable* X *is bound by a list of these keywords, the action*

AddAtomicTrees(X, keyword, biofusion/keywords)

produced the following merged report:

⟨biofusion⟩
⟨selectedproteins⟩1c4tA0, 1c4tC0, 1dpc00, 1eaa00, 1eaf00 and 3cla00⟨/selectedproteins⟩
⟨commonfunction⟩acyltransferase activity (GO : 0008415)⟨/commonfunction⟩
⟨keywords⟩
⟨keyword⟩Gene Ontology⟨/keyword⟩
⟨keyword⟩molecular function⟨/keyword⟩
⟨keyword⟩catalytic activity⟨/keyword⟩
⟨keyword⟩transferase activity⟨/keyword⟩
⟨keyword⟩acyltransferase activity⟨/keyword⟩
⟨keyword⟩transferring acyl groups⟨/keyword⟩
⟨keyword⟩transferring groups other than amino − acyl groups⟨/keyword⟩
⟨/keywords⟩
⟨/biofusion⟩

By contrast, the action AddAtomicTree(X, keywords, biofusion) *would produce a merged report that is somewhat harder to read:*

⟨biofusion⟩
⟨selectedproteins⟩1c4tA0, 1c4tC0, 1dpc00, 1eaa00, 1eaf00 and 3cla00⟨/selectedproteins⟩
⟨commonfunction⟩acyltransferase activity (GO : 0008415)⟨/commonfunction⟩
⟨keywords⟩Gene Ontology, molecular function,
catalytic activity, transferase activity, acyltransferase activity
transferring acyl groups, transferring groups other than
amino − acyl groups⟨/keywords⟩
⟨/biofusion⟩

9.5.8 RepeatAddNode

RepeatAddNode(**Constant$_1$**, **Constant$_2$** | **LogicalVariable**, **Constant$_3$**) – Constant$_1$ specifies the name of the node to be added to the output report. Constant$_2$ or LogicalVariable specifies the number of times this node is to be added to each branch specified by the value of Constant$_3$. For example, if Constant$_1$ were the node vehicle, Constant$_2$ had the value 5, and Constant$_3$ was the branch usedvehicles, then this action would result in the node vehicle being added five times to the node usedvehicles in the merged report.

It may not always be known in advance of querying the knowledgebase how many nodes are to be added to each target branch; hence the second argument specifying the number of nodes to be added may on occaison be a logical variable rather than a constant.

Example 9.5.9 *Suppose you are developing an application to merge information about used cars. The output report has a root element* ⟨usedvehicles⟩ *and is to have a child element,* ⟨vehicle⟩*, for each vehicle included in the output report. Then, given an initial skeleton containing only the root element* ⟨usedvehicles⟩ *the action* RepeatAddNode(vehicle, 5, usedvehicles) *would produce the following addition to the merged report:*

```
⟨usedvehicles⟩
        ⟨vehicle⟩⟨/vehicle⟩
        ⟨vehicle⟩⟨/vehicle⟩
        ⟨vehicle⟩⟨/vehicle⟩
        ⟨vehicle⟩⟨/vehicle⟩
        ⟨vehicle⟩⟨/vehicle⟩
⟨/usedvehicles⟩
```

The value of $\mathrm{Constant}_1$ should be a node or element, and not a branch. Note also that if the branch specified by $\mathrm{Constant}_3$, the target, does not exist in the structure of the merged report, as already constructed, then an error (Error 10) will result.

This action is included purely as a utility action. Its effect is identical to executing the action `AddNode` n times, where n is the value of the second argument to `RepeatAddNode`.

9.5.9 RepeatAddText

RepeatAddText(Variable | LogicalVariable, Constant) — each textentry in the list that instantiates the first argument (the variable of the logical variable) is attached to a branch of the merged report specified by the second argument (the constant). In this case, then, the textentry that instantiates the variable or the logical variable should be a list; and the size of the list should be the same as the number of branches in the output merged report specified by the constant. For example, if the list that instantiates the first argument is [one, two, three], and the value of the constant is numberreport/numbers, then there should be three branches of the form numberreport/numbers in the merged report. This action will then add one to the first branch, two to the second, and three to the third.

Example 9.5.10 *Suppose you are developing an application to merge information about used cars. The output report has a root element* ⟨usedvehicles⟩ *and child elements for* ⟨vehicle⟩, ⟨year⟩, ⟨make⟩, ⟨model⟩, ⟨mileage⟩, ⟨lowestprice⟩, *and so on. As a result of a query you have a list of lowest prices for each vehicle that you wish to add to the merged report. The list is* $3,450, $5,666, $7,899, *and it is used to bind the logical variable* X. *Then, given a suitably constructed initial skeleton, the action* RepeatAddText(X, usedvehicles/vehicle/lowestprice) *would produce the following change to the merged report:*

```
⟨usedvehicles⟩
        ⟨vehicle⟩
                ⟨year⟩⟨/year⟩
                ⟨make⟩⟨/make⟩
                ⟨model⟩⟨/model⟩
                ⟨mileage⟩⟨/mileage⟩
                ⟨lowestprice⟩$3,450⟨/lowestprice⟩
        ⟨/vehicle⟩
        ⟨vehicle⟩
                ⟨year⟩⟨/year⟩
                ⟨make⟩⟨/make⟩
                ⟨model⟩⟨/model⟩
                ⟨mileage⟩⟨/mileage⟩
                ⟨lowestprice⟩$5,666⟨/lowestprice⟩
        ⟨/vehicle⟩
        ⟨vehicle⟩
                ⟨year⟩⟨/year⟩
                ⟨make⟩⟨/make⟩
                ⟨model⟩⟨/model⟩
                ⟨mileage⟩⟨/mileage⟩
                ⟨lowestprice⟩$7,899⟨/lowestprice⟩
        ⟨/vehicle⟩
⟨/usedvehicles⟩
```

Note that if the branch specified by the constant does not exist in the structure of the merged report, as already constructed, then an error (Error 10) will result.

If the output merged report contains several branches with the $same$ path, and you wish to add textentries to each of these branches, then you should use this action rather than repeatedly using the action `AddText`, as the latter will merely result in each textentry being added to the *first* branch with the specified path, and so overwriting the previous textentry.

9.5.10 RepeatAddAtomicTrees

$\mathbf{RepeatAddAtomicTrees}(\mathbf{LogicalVariable}, \mathbf{Constant_1}, \mathbf{Constant_2})$ — the textentry that instantiates the logical variable should be a list of lists. For each list in the list we do the following: for each item in that list an "atomic tree" will be created, consisting of that item as a textentry together with the node specified as the value of $\mathrm{Constant}_1$. Each of these "atomic trees" will then be added to the merged report as the child of the node at the end of the branch specified by $\mathrm{Constant}_2$.

As was the case with action `AddAtomicTree`, the value of $\mathrm{Constant}_1$ must be an individual report node or element, and not a branch. And the branch which is the value of $\mathrm{Constant}_2$ must already be in the merged output report, or an error (Error 10) will occur.

An example of when this kind of action might be used is where a condition produces a list of lists of textentries which will be added to the merged report. Each list of textentries is to be attached to a separate node. However, rather than displaying those textentries as a list, it is decided that the merged report would be clearer if each item in the list were displayed as a separate tagged textentry, the whole group being child nodes of a single parent.

Example 9.5.11 *The following example comes from a later stage of a case study in bioinformatics. (Cf. 9.5.8) The objective of this case study was to take a group of proteins and from that group select subgroups that had a common molecular function. The output report was also supposed to display the keywords and phrases from amongst the functional annotations of the proteins in each subgrouping. Used in combination with a suitable set of input reports, and where the logical variable* `X` *is bound by a list of lists of these keywords, the action*

`AddAtomicTrees(X, keyword, biofusion/functionalgroup/keywords)`

produced the following merged report:

```
⟨biofusion⟩
    ⟨functionalgroup⟩
        ⟨selectedproteins⟩1c4tA0, 1c4tC0, 1dpc00, 1eaa00, 1eaf00 and 3cla00⟨/selectedproteins⟩
        ⟨commonfunction⟩acyltransferase activity (GO : 0008415)⟨/commonfunction⟩
        ⟨keywords⟩
            ⟨keyword⟩Gene Ontology⟨/keyword⟩
            ⟨keyword⟩molecular function⟨/keyword⟩
            ⟨keyword⟩catalytic activity⟨/keyword⟩
            ⟨keyword⟩transferase activity⟨/keyword⟩
            ⟨keyword⟩acyltransferase activity⟨/keyword⟩
            ⟨keyword⟩transferring acyl groups⟨/keyword⟩
            ⟨keyword⟩transferring groups other than amino − acyl groups⟨/keyword⟩
        ⟨/keywords⟩
    ⟨/functionalgroup⟩
    ⟨functionalgroup⟩
        ⟨selectedproteins⟩1c4tA0, 1cFb01, 1cFb02 and 3hla00⟨/selectedproteins⟩
        ⟨commonfunction⟩transferase activity (GO : 0003824)⟨/commonfunction⟩
        ⟨keywords⟩
            ⟨keyword⟩Gene Ontology⟨/keyword⟩
            ⟨keyword⟩molecular function⟨/keyword⟩
            ⟨keyword⟩catalytic activity⟨/keyword⟩
            ⟨keyword⟩transferase activity⟨/keyword⟩
            ⟨keyword⟩transferring acyl groups⟨/keyword⟩
            ⟨keyword⟩transferring groups other than amino − acyl groups⟨/keyword⟩
        ⟨/keywords⟩
    ⟨/functionalgroup⟩
    ⟨functionalgroup⟩
        ⟨selectedproteins⟩1b4tA0, 1bFb01, 1cFb02 and 1kla00⟨/selectedproteins⟩
        ⟨commonfunction⟩lyase activity (GO : 0016417)⟨/commonfunction⟩
        ⟨keywords⟩
            ⟨keyword⟩Gene Ontology⟨/keyword⟩
            ⟨keyword⟩molecular function⟨/keyword⟩
            ⟨keyword⟩catalytic activity⟨/keyword⟩
            ⟨keyword⟩transferase activity⟨/keyword⟩
            ⟨keyword⟩transferring acyl groups⟨/keyword⟩
            ⟨keyword⟩transferring groups other than amino − acyl groups⟨/keyword⟩
        ⟨/keywords⟩
    ⟨/functionalgroup⟩
⟨/biofusion⟩
```

For each list in the list of lists the following was done: each textentry in that list was combined with the node ⟨keyword⟩ *to create an "atomic tree". This subtree was then added to the node specified by the target branch* biofusion/functionalgroup/keywords.

Note that the number of lists in the list of lists should equal the number of occurrences of the target branch specified by $\mathrm{Constant}_2$.

9.5.11 AddTrees

AddTrees(**Variable** | **LogicalVariable**, **Constant**) — The textentry that is the value of Variable or LogicalVariable is a list of strings each of which represents a tree. Each tree that is represented by this list is then attached to the element or tag at the end of the branch specified by Constant. If the first argument is a "set" variable then the trees that ground Variable are the subtrees of the input reports whose root node is the tag at the end of the branch specified by the variable. The root nodes of the trees added will be the children of the target node in the merged report, which is specified by Constant.

For example, if Variable was set//weatherreport/regionalreport and Constant is weatherreport /regionalreports then this action would take from each input report the subtree whose root node is regionalreport and add those subtrees to the merged output report at the node regionalreports.

It is important to note that the textentry that is the value of the first argument is a list, and the elements of that list will be representations of trees. (See example 9.5.12) This means that if a variable is used as the first argument (as will usually be the case) it must be a "set" variable. And, of course, the `extract` attribute of the ⟨`branch`⟩ element must take the value "subtree", not the value "textentry".

Example 9.5.12 *Consider the action:*

`AddTrees(set//weatherreport/regionalreport, weatherreport/regionalreports)`

The following is an example of a string that might be used by the fusion tool to ground the variable in this action.

```
regionalreport(region(SouthEast), maxtemp(14c)), regionalreport(region(NorthEast), maxtemp(11c)),
      regionalreport(region(NorthWest), maxtemp(12c)), regionalreport(region(SouthWest), maxtemp(14c))
```

Example 9.5.13 *Suppose that there are four input reports and they contain the following subtrees:*

```
⟨regionalreport⟩
      ⟨region⟩SouthEast⟨/region⟩
      ⟨maxtemp⟩14C⟨/maxtemp⟩
⟨/regionalreport⟩
```

```
⟨regionalreport⟩
      ⟨region⟩NorthEast⟨/region⟩
      ⟨maxtemp⟩11C⟨/maxtemp⟩
⟨/regionalreport⟩
```

```
⟨regionalreport⟩
      ⟨region⟩NorthWest⟨/region⟩
      ⟨maxtemp⟩12C⟨/maxtemp⟩
⟨/regionalreport⟩
```

```
⟨regionalreport⟩
      ⟨region⟩SouthWest⟨/region⟩
      ⟨maxtemp⟩14C⟨/maxtemp⟩
⟨/regionalreport⟩
```

And suppose that the merged report, as so far constructed, is as follows:

```
⟨weatherreport⟩
      ⟨source⟩BBCi⟨/source⟩
      ⟨date⟩10th April 2003⟨/date⟩
      ⟨regionalreports⟩⟨/regionalreports⟩
⟨/weatherreport⟩
```

Then the action

`AddTrees(set//weatherreport/regionalreport, weatherreport/regionalreports)`

would lead to the following addition to the merged report:

```
⟨weatherreport⟩
       ⟨source⟩BBCi⟨/source⟩
       ⟨date⟩10th April 2003⟨/date⟩
       ⟨regionalreports⟩
              ⟨regionalreport⟩
                     ⟨region⟩SouthEast⟨/region⟩
                     ⟨maxtemp⟩14C⟨/maxtemp⟩
              ⟨/regionalreport⟩
              ⟨regionalreport⟩
                     ⟨region⟩NorthEast⟨/region⟩
                     ⟨maxtemp⟩11C⟨/maxtemp⟩
              ⟨/regionalreport⟩
              ⟨regionalreport⟩
                     ⟨region⟩Northwest⟨/region⟩
                     ⟨maxtemp⟩12C⟨/maxtemp⟩
              ⟨/regionalreport⟩
              ⟨regionalreport⟩
                     ⟨region⟩SouthWest⟨/region⟩
                     ⟨maxtemp⟩14C⟨/maxtemp⟩
              ⟨/regionalreport⟩
       ⟨/regionalreports⟩
⟨/weatherreport⟩
```

As was the case with actions `AddTree` and `ExtendTree`, although this action could take a logical variable as its first argument, it is expected that in most cases it will take a variable, and that its primary use (as in example 9.5.13) will be in situations where we wish simply to add subtrees from the input report straight into the output report.

9.5.12 MultiExtendTree

MultiExtendTree(**Variable** | **LogicalVariable**, **Constant**) — The textentry that is the value of Variable or LogicalVariable is a list of strings each of which represents a tree. Each tree that is represented by this list is then used to extend the merged report tree by attaching the descendants of the root node (but *not* the root node) of the tree to the element or tag at the end of the branch specified by Constant. As with action `AddTrees`, if the first argument is (as envisaged) a variable, then each tree that grounds Variable is the subtree of the input report tree whose root node is the tag at the end of the branch specified by the variable. But unlike action `AddTree`, this root node itself will *not* be added to the merged report; instead, its children will be the children of the target node in the merged report, which is specified by Constant.

For example, if Variable was `set//weatherreport/regionalreport` and Constant is `weatherreport /regionalreports`, then this action would take from each input report the subtree whose root node is `regionalreport` and add the child nodes (and all their descenants) of node ⟨`regionalreport`⟩ to the end of the branch `weatherreport/regionalreports` in the merged report.

As is the case with action `AddTrees`, the textentry that is the value of the first argument is a list, and the elements of that list will be representations of trees. (See example 9.5.12) This means that if a variable is used as the first argument (as will usually be the case) it must be a "set" variable. And, of course, the `extract` attribute of the ⟨`branch`⟩ element must take the value "subtree", not the value "textentry".

Example 9.5.14 *Suppose that there are four input reports and they contain the following subtrees:*

```
⟨regionalreport⟩
       ⟨region⟩SouthEast⟨/region⟩
       ⟨temp⟩
              ⟨maxtemp⟩14C⟨/maxtemp⟩
              ⟨mintemp⟩4C⟨/mintemp⟩
       ⟨/temp⟩
⟨/regionalreport⟩
```

```
⟨regionalreport⟩
       ⟨region⟩NorthEast⟨/region⟩
       ⟨temp⟩
              ⟨maxtemp⟩11C⟨/maxtemp⟩
              ⟨mintemp⟩2C⟨/mintemp⟩
       ⟨/temp⟩
⟨/regionalreport⟩
```

```
⟨regionalreport⟩
       ⟨region⟩NorthWest⟨/region⟩
       ⟨temp⟩
              ⟨maxtemp⟩12C⟨/maxtemp⟩
              ⟨mintemp⟩3C⟨/mintemp⟩
       ⟨/temp⟩
⟨/regionalreport⟩
```

```
⟨regionalreport⟩
       ⟨region⟩SouthWest⟨/region⟩
       ⟨temp⟩
              ⟨maxtemp⟩14C⟨/maxtemp⟩
              ⟨mintemp⟩5C⟨/mintemp⟩
       ⟨/temp⟩
⟨/regionalreport⟩
```

And suppose that the merged report, as so far constructed, is as follows:

```
⟨weatherreport⟩
       ⟨source⟩BBCi⟨/source⟩
       ⟨date⟩10th April 2003⟨/date⟩
       ⟨regionalreports⟩⟨/regionalreports⟩
⟨/weatherreport⟩
```

Then the action

```
MultiExtendTree(set//weatherreport/regionalreport, weatherreport/regionalreports)
```

would lead to the following addition to the merged report:

```
⟨weatherreport⟩
       ⟨source⟩BBCi⟨/source⟩
       ⟨date⟩10th April 2003⟨/date⟩
       ⟨regionalreports⟩
              ⟨region⟩SouthEast⟨/region⟩
              ⟨temp⟩
                     ⟨maxtemp⟩14C⟨/maxtemp⟩
                     ⟨mintemp⟩4C⟨/mintemp⟩
              ⟨/temp⟩
              ⟨region⟩NorthEast⟨/region⟩
              ⟨temp⟩
                     ⟨maxtemp⟩11C⟨/maxtemp⟩
                     ⟨mintemp⟩2C⟨/mintemp⟩
              ⟨/temp⟩
              ⟨region⟩Northwest⟨/region⟩
              ⟨temp⟩
                     ⟨maxtemp⟩12C⟨/maxtemp⟩
                     ⟨mintemp⟩3C⟨/mintemp⟩
              ⟨/temp⟩
              ⟨region⟩SouthWest⟨/region⟩
              ⟨temp⟩
                     ⟨maxtemp⟩14C⟨/maxtemp⟩
                     ⟨mintemp⟩5C⟨/mintemp⟩
              ⟨/temp⟩
       ⟨/regionalreports⟩
⟨/weatherreport⟩
```

As was the case with actions `AddTree`, `ExtendTree`, and `AddTrees`, although this action could take a logical variable as its first argument, it is anticipated that in most cases it will take a variable, and that its primary use (as in example 9.5.14) will be in situations where we wish simply to add subtrees from the input report straight into the output report.

ActionName	1st Argument	2nd Argument	3rd Argument
Initialize	constant	-	-
AddText	variable | logical variable	constant	-
AddNode	constant	constant	-
AddAtomicTree	variable | logical variable	constant	constant
AddTree	variable | logical variable	constant	-
ExtendTree	variable | logical variable	constant	-
AddAtomicTrees	variable | logical variable	constant	constant
RepeatAddNode	constant	constant | logical variable	constant
RepeatAddText	variable | logical variable	constant	-
RepeatAddAtomicTrees	logical variable	constant	constant
AddTrees	variable | logical variable	constant	-
MultiExtendTree	variable | logical variable	constant	-

Table 9.1: The twelve defined actions and their legal arguments. The vertical bar signifies disjunction.

9.5.13 Summary of actions and their arguments

Table 9.1 summarizes the legal arguments of the twelve defined actions. Often, though by no means always, logical variables can, as a matter of syntax, be used wherever variables are used, and vice versa. This does not mean, however, that as a matter of semantics it makes sense to do this. In any given situation it will only make sense to use either a variable or (exclusive "or") a logical variable; it just depends on what the role of the action is in the context of the rule. A case in point is the first rule from the weather reports demonstration. (See example 9.5.15) This rule serves several functions, but focus on the second and third actions. Both add textentries to nodes in the merged report, but whereas the first `AddText` action requires a logical variable as its first argument, the second `AddText` action requires a variable. Given the roles of these actions and the purpose of this rule, although from a syntactical point of view it would be fine to swap these arguments around, it would obviously make no sense to do that.

Example 9.5.15 *A fusion rule from the weather reports demonstration. Although the second, third and fourth actions are of the same kind,* `AddText`*, the first of these requires a logical variable as its first argument, whilst the second and third occurrences require a variable. This illustrates the point that although from a syntactical point of view the action* `AddText` *may take either type of argument as its first argument, in any given instance only one kind of argument will be appropriate.*

```
AcceptedSource(set//weatherreport/source, X)
AND SameDate(set//weatherreport/date)
AND SameCity(set//weatherreport/city)
IMPLIES Initialize(weatherreport(source, date, city, today, temp(max, min)))
AND AddText(X, weatherreport/source)
AND AddText(1//weatherreport/date, weatherreport/date)
AND AddText(1//weatherreport/city, weatherreport/city)
```

There are further constraints on what counts as a legal action. These constraints concern the nature of the *variables* that may occur in a particular kind of action. They are summarized in table 9.2. For an explanation of why these constraints apply, please consult the subsection for each action. If a rule set that violates any of these constraints is used to merge a set of reports then an error (Error 15, see Appendix A) will occur and the reports will not be merged.

9.5.14 Ordering of actions

In most cases, though not all, the order of actions in a single rule is significant. The reason why order is often important for actions is simply that when a rule has more than one action, generally (though not

Action Name	Variable	
	source | set	extract
AddText	source | set	textentry
AddAtomicTree	source | set	textentry
AddTree	source	subtree
ExtendTree	source	subtree
AddAtomicTrees	set	textentry
RepeatAddText	set	textentry
AddTrees	set	subtree
MultiExtendTree	set	subtree

Table 9.2: Where a variable may occur as an argument to an action, constraints apply to the nature of the variable. This table summarizes those constraints. The vertical bar signifies disjunction.

always), that will be because one action will be adding a new node to a report that will be the target node for subsequent actions. And, of course, if an action is to add a text entry or a subtree to a specified target node, then that target must already be in the output report tree. That is why, for example, if a rule contains an Initialize action, that action must be the first action of the rule. (See 9.6.2, below.) If the target node is absent, then an error (Error 10, see appendix A) will result, and the reports will not be merged.

Example 9.5.16 *A fusion rule from the weather reports demonstration for merging the textentries for air pressure. The rule has two actions; the first action adds a node,* `pressure`*, which is the target for the second action. Clearly, if the order of the two actions were reversed, the* `AddAtomicTree` *action would be attempting to add a node,* `absolutevalue`*, and a textentry, to a target node that had not yet been added to the output report. The result would be an error (Error 10), and the reports would not be merged.*

```
Rulecode is 12 (Status = optional)

SimilarAbsoluteValues(set//weatherreport/pressure/absolutevalue, X)
IMPLIES AddNode(pressure, weatherreport)
AND AddAtomicTree(X, absolutevalue, weatherreport/pressure)
```

9.6 Foundational rules and Initialize actions

9.6.1 The rationale for foundational rules

Some rules ought to have a privileged status. We classify rules with this status as *foundational*. Their priviliged status consists in the fact that if any foundational rule fails to fire, then no merged report is to be constructed. We may say, then, that fusion rules have a veto over the production of a merged report. There is, however, one exception to this. If a fusion rule is deemed foundational because it contains an Initialize action (see (1) below), the first such rule to fire causes any subsequent foundational rule with Initialize actions to be disabled.

There are two reasons why a rule is designated as foundational: (1) it contains an action whose role is to initialize the construction of the merged report. At the very least this will specify the root node of the report. We call such actions Initialize actions. (2) it contains some condition whose success is essential to the integrity of the merged report. In the first case, if a rule containing an Initialize action were to fail, then we would not want any other rules to execute because without this initial skeleton or report structure there would be no merged report to which other rules could add text, nodes, etc.

In the second case, the reason why rules involving certain conditions may be given foundational status is that the failure of those conditions is deemed (by the application developer) to fatally threaten the integrity

of the merged report. In the case of a set of rules for merging weather reports, for example, if the input reports were not for the same geographical location (city, region, country), or were not for the same time period, or were not from accepted sources, then (in most cases at any rate) there is simply no point in trying to construct a merged report. (What point is there in merging today's weather in New York and today's weather in Tokyo, or yesterday's weather in London and tomorrow's weather in Rome?) By contrast, if the windspeeds, temperatures, humidities, etc., fail to be similar enough to be included as-is in a merged report, then typically the conflict will be resolved by applying some aggregation function (e.g. voting, preference over sources, etc.) and the result will be added to the merged report. And even if it was not thought useful to do this, that should not prevent a merged report from being constructed that included other information on, for example, air pollution or visibility. Hence rules involving these input report elements would usually be considered to be optional, not foundational.

The requirement for some rules to have foundational status results, in part, from the nature of the information being merged. An input report can be viewed as being concerned with an entity in some loose sense (a weather report, a protein domain, an opinion poll, etc.) and a number of attributes of that entity. Some elements in the report serve to identify the entity in question, whilst other elements serve to express the attributes of that entity. A weather report, for example, might have elements for source, date, and location, serving to individuate that report, whilst it might also have elements with textentries for temperature, humidity, and so on, specifiying the attributes of that weather report.

The significance of this distinction is that rules that are concerned with these two types of report elements can be viewed as having different functions. Rules that deal with elements that individuate reports serve to determine if merging should take place; if we do not have the right kind of entities, a pre-condition for merging does not obtain. Hence, conditions dealing with such elements tend to be placed in foundational rules. By contrast, rules that deal with the attributes of the entities being merged serve to determine the substantive content of the merged report. If these attributes are in conflict (conflicting reports for today's weather, etc.) then some rules will fail. But that is not a reason for not merging; rather, that is an important fact that must be dealt with and included in the merged report. Hence, rules dealing with this second type of report element tend to be given optional status.

It should be noted that sometimes it is not necessary to formulate rules for the individuating or identifying elements of input reports. This is because some form of preprocessing or preselecting of the input data has taken place. In a case study that involved merging protein domains so as to find functionally similar subgroups, for example, the set of input data all come from the same superfamily, so the rules could simply go ahead and attempt to merge the domains. Similarly, if we wished to rely on the user to make sure that all the input weather reports were for the same time and place, and were all from accepted sources, then no rule would be required to check this. This is not to say, however, that the idea of a foundational rule goes by the board; this special status for a rule is still required because of the need of every rule set to contain a rule with an Initialize action.

9.6.2 Constraints on rule sets and their execution

The existence of foundational rules and Initialize actions imposes several constraints on both the construction of rule sets, and on their execution by the fusion tool.

The following considerations help to explain the constraints listed below: (a) every set of rules seeks to construct a merged report, and so must contain at least one Initialize action. A well engineered set of rules must, then, contain at least one foundational rule. (b) There is no reason why a set of rules should not contain more than one rule with an Initialize action, but for obvious reasons — we cannot build a merged report with more than one root node — only one such action should be executed for any set of input reports.

We impose the following eight constraints on the use and occurrence of foundational rules and Initialize actions:

1. All foundational rules must *preceed* all non-foundational rules. This is because the success of these rules is in one way or another essential to the construction of the merged report.

2. Every set of rules must contain at least one rule with an Initialize action.

3. Any Initialize action must occur in a foundational rule.

4. A single rule may contain at most only one Initialize action.

5. An Initialize action must be the *first* action in a rule. Quite often, for example, a single rule contains an Initialize action followed by several actions adding text or nodes to the the skeleton thus created. Clearly, if the order of these actions was reversed, then some actions would be seeking to add text and nodes to a skeleton that doesn't yet exist.

6. For a similar reason, rules with Initialize actions should preceed all other rules (including, of course, foundational rules without Initialize actions).

7. For any given set of input reports, $exactly\ one$ rule with an Initialize action should fire. The conditions of these rules should thus be engineered accordingly.

8. If any foundational rule *without* an Initialize action fails, then the reports should not be merged.

Inspection of these constraints reveals that the last two (7 and 8) are distinct in that it cannot be determined whether or not a set of rules meets them prior to querying the knowledgebase. For this reason the fusion tool checks the first six constrains prior to merging, and if any of these constraints is violated an error (Error 12, see Appendix A) results, and the reports will not be merged.

On the other hand, whether constraints 7 or 8 are violated can be determined only after querying the knowledgebase, so if either of these constraints fails a distinct error (Error 7, see Appendix A) occurs. In this case, then, an error occurs if either a foundational rule without an Initialize action fails, or if no foundational rule with an Initialize action succeeds.

9.7 Rule grouping

We turn now to a discussion of how a set of rules should be constructed if they are to have the correct logical behaviour with respect to a set of input reports. The fact that for a given set of rules and for any set of input reports, at most one rule with an Initialize action should fire brings it out that, although rules are individually numbered, the rules in a well-engineered rule set often should not act in isolation from each other. In many cases, certain rules bear close relations to some rules that they do not bear to others. In particular, when more than one rule contains actions that seek to modify the same target node in the merged report, quite often it will be the intention of the developer that at most one of those rules should be executed on any given occaison. This will certainly be true when the actions in question seek to add *textentries* to the merged report since subsequent actions would simply *overwrite* the result of prior actions. That is not so in the case where subtrees are being added to the target node, since in that case subsequent subtrees will simply be *added* as sibling child nodes of the target. But even in such cases, often it will be the developer's intention that no more than one rule in the subgroup should execute.

Where the grouping of rules is an issue, we formulate the following condition:

Condition 1 Rules whose actions have the same *target element* in the merged report should be such that, for any given set of input reports, *at most one such rule should be executed*.

However, whether or not this condition is violated by a group of rules whose actions all deal with the same output report element cannot be determined simply by inspecting the logical form of their conditions. For example, two rules (whose actions deal with the same merged report element) with conditions of the form (1) p and q, and (2) p and r, may seem to violate this condition in that both sets of conditions could be true, given a suitable set of input reports. However, it may be that, given the background knowledge in the knowledgebase, whenever q is true r is false, and vice versa, so that in fact there is no possibility of both rules firing.

It should be noted, however, that it is not the case that in such a grouping of rules (whose actions all have the same target element in the output report), that for any set of input reports, *at least one rule should be fired*. Rather, the conditions of a group of rules that belong together should be formulated so that they can all *fail*. Intuitively, this is desirable behaviour. If none of the input reports has a textentry for the report element with which the conditions of the rules in the group are concerned (in this case the schema variable, if there is one, would be ground by a list of `nulls`), we want no action to be taken, and so all the rules in the group should fail. For this reason negation-as-failure is in general not employed in formulating conditions for groups of rules (though of course it is frequently used with predicates in a Prolog knowledgebase), purpose-built predicates being defined instead. As an instance of this consider Example 9.7.1: if the set variable that is the argument to `EquivalentTerms` in rule 12 is ground with a list of textentries that are all `null`, then the predicate will fail. This means, of course, that were `NOT EquivalentTerms` to be used with the same set variable as an argument in rule 13, it would succeed, leading to the risk that if other conditions in the rule were also to succeed, a `null` textentry would be added to the merged report. As this is deemed undesirable, a new predicate, `NotEquivalentTerms` is defined that also fails if the set variable is ground with a list of textentries that are all `null`. Hence, if no input report contains a tag for `today` neither will the output report.

Example 9.7.1 *A pair of rules for merging weather reports dealing with the entry for today's weather. The rule code indicates that these have been registered as the twelfth and thirteenth rules in the rule set. The status of both is optional. Rule 12 says that if all the textentries for today's weather are from the same equivalence class, the preferred term from that class is selected and that text is added to the* `today` *tag in the merged report. If the first condition of rule 12 fails, then, provided that the values of the textentry variables for the report element* `today` *are not all* `null`*, the first condition of rule 13 will succeed. The second condition simply constructs the disjunction of each textentry for today's weather from the set of input reports (duplicates and nulls being removed), and the action adds the disjunction as the textentry for the branch* `weatherreport/today`.

```
Rulecode is 12 (Status = optional)

EquivalentTerms(set//weatherreport/today)
AND PreferredTerm(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)

Rulecode is 13 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND Disjunction(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

Of course it is possible to group rules into subgroups in other ways. In particular, it may seem natural to consider some rules as belonging to a single subgroup if they share some of their conditions, or if they have conditions which stand in the relation of negation. One might want to group rules in this way because there is, as far as we can see, no reason why the actions of rules in such subgroups should have to have the same output report element as a target.

In the simplest case, for example, there is no reason why we might not have two rules of the form: (1) If p succeeds then execute action A_1; (2) If $not\ p$ succeeds then execute action A_2, where A_1 and A_2 have different output report elements as their targets. Of course, in these sort of rule groupings, if more than one rule fires for any given set of input reports, there is no danger of later rules in the grouping overwriting the same output report element. However, it is presumably still the intention of the rule engineer that at most

one rule in such groupings should succeed, and that if both actions were to be executed for a given input, the logical integrity of the merged report would be violated.

A problem in determining which rules belong to this kind of subgroup is deciding how many conditions the rules should share. Clearly they need not share all of them. The following seems a reasonable subgrouping: (1) If p succeeds and q succeeds, then execute action A_1; (2) If p succeeds and q fails, then execute action A_2; (3) If p fails and r succeeds, then execute action A_3; and (4) If p fails and r fails, then execute action A_4. However, merely sharing one condition is not a sufficient condition for us to group some fusion rules together, for it is clear that in some cases such a condition is too weak. This is because of the use of logical variables in the formulation of rules. The same symbol, for example `X`, might occur in many different rules but with different denotations, with the result that, at least syntactically, the same condition can appear in rules that clearly do not belong together. For example, in a set of rules that merge conflicting textentries in weather reports by using weighted voting, the condition `WeightedPopularSharing(set//weatherreport/source, X, Y)` occurs in rules dealing with today's weather, pressure (direction of change), air visibility, and sunindex. It may be that such groupings can be delimited, instead, by taking into account the presence of variables indicating with which input report element the rules are concerned, though the widespread occurrence of variables denoting the source of a report (as in the condition just mentioned) makes this difficult.

Whichever way we choose to group rules into subgroups there is a further problem that needs to be avoided. Suppose we have two rules in a subgroup with conditions of the form (1) p and q, and (2) $not\ p$. In this case the first condition succeeds and at most one of the pair can fire, since if the first rule's conditions succeed the second rule's must fail, and vice versa. But if p succeeds and q fails neither rule will fire and the merged report may well be lacking information it ought to contain. For example, there may be a node in the merged report with no textentry. However, whether another rule should be added (with conditions p and $not\ q$) depends, once again, on the knowledgebase. If the knowledgebase is such that if p succeeds q must also succeed then the rules are satisfactory as they stand. It seems then, that there is a further condition on an adequately engineered set of rules:

Condition 2 For any input set of reports (excepting the possibility that all the schema variables are ground to null by the input reports), and for each subgroup of rules, *at least one rule should be executed*.

However, whether or not this condition is violated by a group of rules cannot be determined simply by inspecting the logical form of their conditions because, as already mentioned, the knowledgebase also plays a crucial role.

9.8 Rule coverage

The phrase "rule coverage" is ambiguous. It could mean coverage with respect to the *input* reports or it could mean coverage with respect to the merged *output* report. In the former case, as we have already seen, there is no reason why a set of rules must have rules that deal with each element in the input reports. Whether the coverage of a set of rules is complete in this sense just depends on whether the developer is interested in merging all the information in the input reports.

With respect to coverage of the output report, there can in fact be situations where, although a node has been added to the merged report, no rules exist which are intended to add content to that node. This will be so in cases where an action has constructed an initial report skeleton, but no further rules deal with some of the nodes in that skeleton. Of course, the application developer may not care if the rules produce no content (textentry or subtree) for a particular output report element — an empty element may be acceptable in a merged report. But in general we want it to be the case that for each node in the output report there is

at least one rule whose actions seek to add content to that node, else why add the node to the report in the first place.

Condition 3 A set of rules is minimally adequate with respect to coverage if, for each element in the *output* report, the set contains at least one action of which that element is the target.

9.9 How many rules should a rule set have?

As a limiting case, it would of course be possible to deal with any application by formulating a single rule, with a predicate name of, say, `merge`, taking as its arguments a set variable for each element in the input reports. Subsidiary predicates in the knowledgebase would then deal with these set variables as required. The knowledgebase would thus have to be marginally more complex, having the extra `merge` predicate in addition to those predicates dealing with individual input report elements. But there are serious disadvantages with this approach.

First, it would not be possible to tell, simply by inspecting the rule, how the different report elements were to be merged; that is, the rule itself would not specify how conflicts should be resolved — should conflicts between textentries for today's weather be settled by some form of voting, or by preferences over sources, etc. Instead, the reader would have to determine this by inspecting the knowledgebase.

Second, the rule itself would likely be difficult to understand because an action must be included for each output report element. Depending on the size of the output report, then, the single rule could be quite unwieldy.

Third, it is easier to change the aggregation predicate involved in merging the textentries for a particular report element by editing a fusion rule than it is to recode part of the knowledgebase. Suppose, for example, that we wish to resolve conflicts for today's weather by using majority voting rather than first-past-the-post voting. Using a normal set of fusion rules all this need require, assuming the definitions for the predicates already exists in the knowledgebase, is the substitution of one predicate name for another in the appropriate fusion rule. By contrast, even if the code for all the required aggregation predicates exists in the knowledgebase, if there were only one rule, the subsidiary predicate dealing with today's weather would have to be recoded so that it called a different helper aggregation predicate. Although such recoding is not difficult, editing a fusion rule is clearly simpler.

What emerges from this are the advantages, where possible, of using more rather than less fusion rules to input data into a knowledgebase. First, and as was the case with the number of conditions in a rule, it makes it clearer than it would otherwise be what aggregation predicates are being used to merge conflicting information. (Though to be sure, inspection of the knowledgebase is still required to appreciate the exact definition being used.) And, second, given that the knowledgebase already contains a library of aggregation predicates, changes as to which predicates are to be used are more easily effected by editing fusion rules than by editing the knowledgebase itself.

Looking at the matter in general terms, then, there seem to be advantages to rules sets having more rather than less rules. But given a particular application — a particular set of input reports — can we say with any confidence how many rules are required of a well-engineered set? A crude way of estimating this would be to make the following three assumptions: (1) That we are interested in every element in the input reports (that is, none of the information is irrelevant to our interests). (2) That in determining the structured report to be output, the information contained in each input report element is to be treated independently of the information contained in the other input elements. And (3), that we will need a pair of rules to deal with each input report element, one specifying what action to take if a certain condition holds, the other specifying what action to take if the condition fails. If these three assumptions hold, then we can say that the total number of rules will be roughly twice the number of input report elements. And, indeed, in

the weather reports demonstration, this is what we find. In that case the input reports contained between them twelve separate elements. The elements for source, location, and date in effect identify the individual reports, and so were combined into a single, foundational rule. The remaining nine elements were merged independently of each other, and so required two rules each. Thus a total of nineteen one rules were required for this application, and that number remained the same for both sets of rules.

But in some cases these assumptions do not hold. The fact is that what determines the number of rules for an application is simply what we want to *do* with the information in the input reports. There may be cases where, despite the fact that the input reports contain a lot of elements that describe the attributes of the report entity, the purpose of the application is to use all of those attributes together to do one thing. In cases such as this, the number of fusion rules required will be small.

9.10 Relative positioning of conditions and actions

Although it obviously makes more sense to their human readers for rules to be written so that all the conditions preceed all the actions, as far as the fusion tool is concerned this need not be so—the actions could come first! This would not prevent the fusion tool from first checking the conditions against the knowledgebase, and only then proceeding to execute the actions of those rules whose conditions all succeed. But obviously placing the actions first makes the rules harder for people to read! For this reason the DTD for the fusion rule ML enforces the ordering that all conditions must preceed all actions.

9.11 Ordering of rules

Rules are executed by the action engine in the order in which they occur in the rule set. The fact that the output report takes the form of a tree places some constraints on the order of the rules. In particular, all rules with Initialize actions must preceed all rules lacking such actions so that there is a tree of some sort to which those other actions can add a text entry or a subtree.

But more generally, if a particular action adds a text entry or a subtree to a target node in the output report, then the ordering of the rules must be such so as to ensure that the target node is already present in the output report as so far constructed. Provided, however, that this condition is met, there is little significance to the ordering of the rules beyond the ordering of elements which the rule engineer seeks in the output report; if, for example, we want the entry for `temperature` in the output report to preceed that for `humidity`, then the rules should occur in a similar order.

Moreover, within a particular subgroup of rules (that is, a group of rules all of whose actions have the same output report element as a target) there also seems little significance to the order of the rules: for example, although it may be natural for a pair of rules whose condition literals are `SimilarTemperatures` and `NotSimilarTemperatures` to occur in that order, there is no reason, as far as their conditions are concerned, why the order should not be reversed. What is important (at least in this case) is not their order but that, for any set of input reports, the conditions and the knowledgebase are such that at most one rule in this pair should be executed. In fact, as far as the production of a correct output report is concerned, it is not even essential that all the rules in a particular subgroup occur *consecutively*, though it is obviously confusing to their human reader if the two rules concerned with temperatures are separated by a pair of rules dealing with humidity.

In general, then, there is a good deal of flexibility, both in the ordering *between* subgroups of rules in a rule set, and in the ordering of rules *within* subgroups of rules.

9.12 Rule numbering

The fusion rules in a rule set are numbered using the natural numbers. This numbering is purely for the convenience of the human readers of the rules, it makes no difference to their execution. In particular, although it is recommended that rules should be numbered consecutively starting from 1, the numbers make *no difference to the order in which the rules are executed.* The fusion tool simply executes rules *in the order in which they occur in the rule set, regardless of their rule numbers.* Thus if rules are not ordered consecutively, if the numbering contains gaps, or even if some rule numbers are duplicated, this will not effect their execution. Duplicated rule numbers may, however, lead to some confusing error messages!

The rule tool allows the user to automatically renumber rules in a rule set starting from 1. In addition, if a new rule is inserted into an existing set of rules, subsequent rules will be renumbered by the rule tool, but only such as to avoid duplicate numbers.

Chapter 10

The knowledgebase

10.1 Overview

The knowledgebase is the third central component of any application designed to merge potentially inconsistent input reports using fusion rules. As we have already seen (section 9.4.4), fusion rules themselves can contain knowledge about an application domain (for example, voting thresholds) and so they are, strictly speaking, part of the knowledgebase. However, in this chapter we shall be using the term "knowledgebase" to refer only to the external resource (for example, a database or a Prolog program) that usually contains the vast preponderance of the knowledge used to merge reports for a particular subject domain.

In developing the fusion tool a decision was taken to make the tool as versatile as possible with respect to what kinds of external resources it could exploit as a knowledgebase. As a result, rather than hard-code into the fusion tool a link to, for example, a particular Prolog engine and force the user to implement knowledgebases for different applications as different Prolog programs (i.e. as different .pl files), the fusion tool merely provides queries in a specified format, and likewise requires responses in another format.

The upside of this approach is that you, the application developer, are free to exploit any external resources you want as a knowledgebase—Prolog engines, SQL databases, XML, etc. You are not tied to writing Prolog progams if you don't like Prolog. The downside, obviously, is that the development of an interface between the fusion tool and the external resource imposes a substantial overhead on the application developer. There is, then, a trade-off between, on the one hand, versatility and, on the other, simplicity in selecting an architecture for the fusion tool, and in particular with regard to the knowledgebase. We chose versatility. In mitigation, note that this is a one-off task: do it once and, providing all your applications use the same kind of resource (Prolog, SQL, etc.), you can reuse it with any other application that you develop.

Note that it is important to test this interface, with the resource, $independently$ of the fusion tool. That way, of course, you can more easily determine that a bug is due to the interface and not to any other part of the application, such as the fusion rules or the input reports.

In the next two sections we discuss the two main components of a knowledgebase for a fusion rules application. These are (i) the interface between the fusion tool and the external resource used as a knowledgebase, and (ii) that external resource itself.

10.2 The knowledgebase interface

There are three tasks for the knowledgebase interface: (1) It must load the external resource being used as the knowledgebase. (2) It must parse the queries generated by the fusion tool into a format suitable for that external resource. (3) It must parse the responses provided by the external resource (including, most importantly, any error messages) into a format suitable for the fusion tool. We discuss each of these tasks in turn in the following four subsections.

Note: you may skip the following subsection if you are familiar with the concept of dynamic class loading in Java.

10.2.1 Dynamic class loading

The independence or decoupling of the fusion tool from the external resource is achieved by exploiting Java's dynamic class loading capability. Normally, when writing Java source code, the classes called to handle whatever is required by the application being developed can be hard coded into that application. What happens, however, when it is not known in advance (i.e. at compile-time) what the required classes might be? That is our situation. We, the fusion tool developers, do not know what resources you, the application developer, might wish to use. One solution—a poor solution—is simply to provide you with the source code and let you hack around with it. A better solution is to exploit dynamic class loading. Instead of you hard-coding a call to the methods in your classes somwhere in the midst of our fusion tool code, you simply enter the name of that class (minus the .class extension) in a dialog box (using the Select Resource menu option). This name is then passed as a parameter to Java's `Class.forName` method which loads the named class into the Java interpreter at run-time. The `newInstance` method then creates a new instance of this class.

We now have a new instance of whatever class that you, the application developer, have written, loaded into the Java interpreter. But how do we know what methods to call? The answer is that any class that you write that is to be dynamically loaded by the fusion tool must implement the `AnswerQuery` interface.

10.2.2 The `AnswerQuery` Interface

The resources that you wish to exploit as a knowledgebase must be loaded from a Java class that is itself dynamically loaded using the fusion tool. Obviously we cannot say anything about how you might load those resources—that's up to you. We can, however, describe some general constraints that the dynamically loaded class must observe.

Any class that is dynamically loaded must implement the `AnswerQuery` Interface. The JavaDoc for this interface, and associated classes, is supplied with this package, but we will explain here the methods that must be implemented. It is important to note that any classes that you write that implement this Interface must be placed in the folder `dynamicClasses`. The Interface itself contains two method signatures:

```
Response answerQuery(Condition groundCondition);

void tidyUp();
```

The most important method is `answerQuery`. The queries generated by the fusion tool are in the form of a `Condition` object, and the responses expected by the tool must be in the form of a `Response` object.

(Again, JavaDoc is provided for these classes.) Thus the method `answerQuery` requires a `Condition` object as a parameter, and returns a `Response` object. Details of these can be found in the following two subsections. You must implement the body of this method when writing the class that is dynamically loaded, otherwise the external resource will not be queried and the fusion tool will not get a response.

The other method, `tidyUp`, is provided as a utility method that you may or may not choose to implement. It is called whenever you load a new resource, or when you quit the fusion tool. Its purpose is simply to provide you with an opportunity to close down any external resources, such as a database or a Prolog engine.

10.2.3 Parsing queries

Different resources require queries in different formats. Hence any interface that you develop will have to parse the queries generated by the fusion tool into the format required by the resource you have chosen. The question now arises: in what form are the queries that are generated by the fusion tool? We have been representing those queries by such strings as:

`EquivalentTerms([sun, sunshine, sun, sunny spells, sunny])`
`UsePreferredSource([BBC, BBCi, Sky, CNN], [rain, rain, showers, drizzle], X)`

It may seem natural, then, if this was the form in which the fusion tool provided queries. However, this would not be a good idea. It's true that if the external resource being exploited was a Prolog engine then queries in this format would be ideal, since they are in fact (minor syntactical details aside) just Prolog queries. But this format is of no use for other kinds of external resource. They would be required to parse these strings so as to extract the individual arguments (either individual textentries, or lists of textentries) out of which to construct queries in the appropriate format.

It would be much better, then, simply to provide queries in a form in which the individual arguments are readily available to be parsed as needed. For this reason the queries generated by the fusion tool are simply `Condition` objects (see the associated JavaDoc) that have been ground by the reports, and that have had any available bindings added to their logical variables.

The structure of a `Condition` object mimics pretty closely, though not exactly, the XML structure of a condition. So `Condition` objects have the following fields:

- A `conditionName` field. This is a String naming the condition.
- A `conditionSign` field. This is a String that should have one or other of the two values "positive" or "negative".
- An array of `Arguments`. We do not know what kinds of arguments (variables, constants, or logical variables) a condition may have. `Argument` is a wrapper class that in turn consists of a string field, `argType`, whose value should be one of "variable", "logicalvariable", or "constant". There is also a `Variable` field, a `LogicalVariable` field, and a `Constant` field. For any given `Argument` instance, only one of these latter fields should be initialized. Consult the JavaDoc for details of these latter classes.

One central task, then, of any interface that you write is to parse these `Condition` objects into queries of the form required by the resource you have chosen.

10.2.4 Providing responses

Method `answerQuery` returns a `Response` object. The `Response` class has the following fields:

- A boolean field `result`. Unless an error occurs in the knowledgebase, every query must either succeed or fail. Naturally, if the query succeeds then this field takes the value `true`, if it fails, it takes the value `true`.
- A boolean field `errorCode`. Some queries may lead to errors when passed to the external resource being used as a knowledgebase. For example, an SQL query may have incorrect syntax, or an argument in a query to a Prolog engine may start with an upper case character without being enclosed in single quotes. Of course, if you have written the query parser correctly, these things shouldn't happen. But if they do, the fusion tool needs some way of knowing that something has gone wrong. In particular, it should not be misled into thinking that a query has failed because some constraint embodied in the knowledge has not been met, when in fact an error has occurred. Thus when an error does occur, and the external resource returns some error message, the `errorCode` field should be set to `true`. The significance of this is that the fusion tool can now be told what has happened, and itself will generate a knowledgebase error (Error 5, see Appendix A), preventing the reports from being merged. This is obviously preferable than going ahead and merging the reports in the erroneous belief that a particular rule has failed. (See below, and subsection 10.4.2, for details of how to pass on error messages from the external resource so that they can be displayed by the fusion tool.)
- A String field `errorMessage`. Having signaled that an error has occurred in the external resource we would obviously like to given the user of the fusion tool some information about what has gone wrong. The way to do this is to use the error message generated by the external resource to set the value of the `Response` object's String field `errorMessage`, using the `setErrorMessage` method. Then, if the `errorCode` variable is `true`, the fusion tool will display the associated error message and the user will know why the reports were not merged.
- An `ArrayList` field `bindings`. Many queries do not only succeed or fail, they also return information based on the query. This happens when the query contains a logical variable that expects a binding. In such cases we need to know both the name of the logical variable and the binding. This is both because single queries can bind more than one logical variable, and because the bindings are then used to bind subsequent occurrences of the same logical variable in the rule. Hence the responses to queries must associate the names of logical variables with their bindings. The simplest way to do this is to return a list of `LogicalVariable` objects, since these objects have fields for both the name of the logical variable and its binding. (See the associated JavaDoc).

When developing the interface to the knowledgebase it is your responsibility to make sure that the interface returns responses to the fusion tool in this format.

10.3 The knowledgebase proper

We cannot, of course, offer any advice here on how to develop particular kinds of external resources that might be used as a knowledgebase. We can, however, provide some more general remarks on the kinds of knowledge one might expect in a knowledgebase to merge potentially inconsistent reports.

The most important piece of advice we can offer, though, is to develop the external resource *first, independently* of the other parts of the application. For example, if you are using a Prolog engine as the external resource, develop and test your Prolog code first using whatever environment you would normally use. This means that when you come to query the knowledgebase using the fusion tool and some rules and

reports, if you get an error (or the fusion tool crashes!) you'll know that it's not the external resource that's at fault. To repeat: develop and test the external resource independently of the rest of the application. This is the Golden Rule of building an application.

10.3.1 The role of the knowledgebase in merging

In order to merge information in the kinds of ways we have indicated (as in, say, Examples 5.1.1 and 5.1.2) we use both domain specific and more general knowledge in a knowledgebase. The knowledge in the knowledgebase can be viewed as evaluating one of the following two types of query.

Instrument queries These are queries which test aspects of the input information such as testing whether or not particular parts of the information from the various input reports are similar in some particular way. So, for example, in merging weather reports, the conditions in a knowledgebase would typically test such things as whether the temperature, humidity, and pressure, etc., as specified by the various input reports were or were not similar, where what counts as being similar is determined by the domain specialist. To illustrate, consider the two input structured reports given in Example 5.1.1. The textentries for `date` are `19/3/04` and `20 March 2004` respectively, and so an instrument query involving these textentries might be $\texttt{SameDate}([\texttt{19/3/04}, \texttt{20 March 2004}])$ where $\texttt{SameDate}(\texttt{X})$ would be defined to hold if the list `X` contains one or more dates that are the same. In this case, the first of the following does not hold, whereas the second does hold.

$$\texttt{SameDate}([\texttt{19/3/04}, \texttt{20 March 2004}])$$
$$\texttt{SameDate}([\texttt{19/3/04}, \texttt{19 March 2004}, \texttt{19/03/2004}])$$

Further instrument queries may involve a variable that needs to be grounded. Consider the following instrument query $\texttt{PreferredTerms}(\texttt{List}, \texttt{X})$, where for the list of terms in `List`, the variable `X` is ground by the knowledgebase with a list that contains the preferred synonym for each term in `List`. For example, $\texttt{PreferredTerms}([\texttt{mostlysunny}, \texttt{sunny}, \texttt{sunshine}, \texttt{cloudy}], \texttt{X})$, if according to the knowledgebase `sun` is the preferred term for sunny weather, and `cloud` is the preferred term for cloudy weather, then `X` is ground by the knowledgebase with the list $[\texttt{sun}, \texttt{sun}, \texttt{sun}, \texttt{cloud}]$.

Aggregation queries These queries are used to aggregate a tuple of values. Their general form is a predicate, $P(X_1, .., X_n, Y)$, where P is the name of some aggregation predicate, $X_1, .., X_n$ are the arguments containing the terms to be merged from the input reports, plus terms providing contextual information, and Y is the output of aggregating those textentries (see Example 10.3.1). So for example, if the majority of the input reports agree on today's weather, then majority aggregation selects the term used by the majority of the input reports, and uses that in the output report. Or, in the case where the input information is a numerical value, such as temperature, pressure or humidity, if the input values are all within an allowable range, then the interval from the minimum to the maximum of the input values might be selected for inclusion in the output report. Here the aggregation predicate, might look like this: $\texttt{TempInterval}([\texttt{15C}, \texttt{13C}, \texttt{58F}, \texttt{13C}, \texttt{12C}], \texttt{X})$, where X is ground to $\texttt{12} - \texttt{15}$, assuming that the knowledge engineer has determined that a 3C variation is within the acceptable range. Matters are more interesting, however, if the input reports are not deemed to be similar by the knowledge engineer, and so some similarity condition in the knowledgebase fails. In such cases a much richer variety of aggregation functions are available. Conflicts can be dealt with by aggregation functions that implement various sorts of voting functions, preferences over sources, or if the conflicting terms fall into a class hierarchy, the most specific term that is general enough to include them all could be chosen. Whichever method of aggregation is selected, the choice is encoded in the knowledgebase by the knowledge engineer or user according to the needs of the domain in question.

To answer instrument queries and aggregation queries, the knowledgebase requires the definition of clauses that with an inference mechanism can be reasoned with to determine whether the query holds or not, and

if there is a variable in the query, finds groundings for the variable in the query. For our applications we have used Prolog for the language of the knowledgebase and for the inference mechanism. The choice of Prolog was made because of the availability of high performance implementations, associated software, the generality of the language, and the well-developed theoretical foundations. However, Prolog is not best suited for every knowledgebase, those that exploit the knowledge embodied in large ontologies, for example, would likely be better off using a relational database.

Example 10.3.1 *The following two examples show how instrument and aggregation predicates work together to merge information from the input reports. In the first example the instrument and aggregation predicates could be used in merging the reports from Example 5.1.1:*

$$\texttt{SameParty}(\texttt{Parties})$$

$$\texttt{Mean}(\texttt{Values}, \texttt{Mean})$$

Where `Parties` *is a list of parties, one from each report;* `Values` *is a list of poll results, again one from each report; and* `Mean` *is the mean of all the results in list* `Values`*. These two predicates would be applied to each of the three parties occurring in the input reports.* `SameParty` *is a query that is called first to check that the parties named in the list* `Parties` *are all in fact the same party, to ensure that* `Mean` *would be the mean of the polls for a single party. Note that as political parties sometimes have different names ("Conservative" and "Tory", "Republican" and "GOP", etc.), or because their names are sometimes abbreviated ("Labour" and "Lab", "Conservative" and "Cons", etc.), this condition has to do more than simply check that each element in the list* `Parties` *is the same: It would also require several domain-specific facts to be checked in turn. Like many instrument queries, it simply succeeds or fails. If it succeeds,* `Mean` *is a utility aggregation predicate that provides the mean. The* `Mean` *predicate is an example of a predicate that could of course be reused in other applications.*

With their arguments grounded with textentries from the two input reports, the two predicates would look like the following, and `X` *would be ground to* $\texttt{39.5}\%$*.*

$$\texttt{SameParty}([\texttt{Labour}, \texttt{Lab}])$$

$$\texttt{Mean}([\texttt{41}\%, \texttt{38}\%], \texttt{X})$$

Example 10.3.2 *The second example consists of an instrutment and an aggregation predicate that could be used in merging the reports from Example 5.1.2:*

$$\texttt{ConflictingRecommendations}(\texttt{Recommendations})$$

$$\texttt{UsePreferredSource}(\texttt{Sources}, \texttt{Recommendations}, \texttt{PreferredRecommendation})$$

Where `Recommendations` *is a list of recommendations, one from each report;* `Sources` *is a list of the sources of the recommendations, again one from each report; and* `PreferredRecommendation` *is the overall recommendation selected on the basis of a preference over the sources. The first predicate is the condition predicate that checks that the recommendations do in fact conflict. If they do, the second predicate then selects the recommendation from the most preferred source.*

With their arguments grounded with textentries from the four input reports, the two predicates would look like the following, and `X` *would be ground to* `sell`*.*

$$\texttt{ConflictingRecommendations}([\texttt{buy}, \texttt{sell}, \texttt{buy}, \texttt{buy}])$$

$$\texttt{UsePreferredSource}([\texttt{FinancialTimes}, \texttt{HSBC}, \texttt{Waterhouse}, \texttt{MorganStanley}], [\texttt{buy}, \texttt{sell}, \texttt{buy}, \texttt{buy}], \texttt{X})$$

If, instead, the overall recommendation was made on the basis of some form of voting, the sources could be used as a basis of weighting the votes (see below), or if weighting was not to play a part, the first argument could be omitted.

`WeightedMajority`(`Sources`, `Candidates`, `Winner`)

`WeightedMajority` *is an aggregation predicate that selects the element from list* `Candidates` *that has the most votes according to some weighting scheme defined over the sources in* `Sources`*, provided that the winning element has more than 50% of the votes cast. If no candidate meets this requirement, the output of the aggregation predicate,* `Winner`*, is the string* `NoMajority`*, which occurs in the merged report as the text entry for the tag* `overallrecommendation`*. Note that, once again, in both cases the aggregation functions are generic and so could be reused in other applications, though each does require domain-specific facts in order to select the preferred source, or compute the weighted vote totals.*

Queries to the knowledgebase reflect both the tagnames in the structured input reports and, of course, the interests we have in merging those reports. Thus in the case of merging weather reports, the queries will likely involve predicates for similar temperatures, similar pressures, and so forth. Other queries will reflect the kinds of aggregation functions used.

But the knowledgebase will contain much else besides these top-level predicates. In addition, there will need to be facts about, for example, which sources are acceptable, and perhaps a preference ordering over them as well, if that is to be used as a method of aggregation in the case of conflicts. Other facts might include equivalence classes of expressions, say for today's weather (for example, {rain, wet, inclement, downpours, heavy rain, prolonged rain}), preferred terms from amongst these equivalence classes, and class heirarchies where they are appropriate. Besides facts such as these, a number of rules will also be required in order to carry out the subsidiary tasks involved in implementing the top-level predicates. Many of these rules will implement utility functions such as constructing conjunctions or disjunctions of input textentries. Others will remove duplicates from the lists of such textentries, sum the values in a list, calculate their mean (see Example 10.3.1), or determine the maximum and minimum values in such lists.

In many cases, before any similarity comparisons can be made, or before any aggregation functions can be calculated, numerical textentries will need to have their unit suffixes removed ("C", "F", "mph", "mb", "%", etc.), and unit conversions made, if required (kph to mph, Fahrenheit to Centigrade, etc.); so, much of the subsidiary computation in the knowledgebase is concerned with the string manipulation required to do this.

10.3.2 What kind of knowledge is used in a knowledgebase?

The knowledge in the knowledgebase can be seen as falling into one of two categories: domain (or specific or domain-specific) knowledge and generic (or general) knowledge.[1]

Domain knowledge includes: (1) Ontological knowledge. For example, knowledge specifying which terms are synonyms or can be seen as belonging to the same equivalence class ("drizzle" and "light rain", "Bordeaux" and "Claret", etc.); and hierarchical information about more general and less general terms ("rain" is less general, or is a subclass of "precipitation", etc.). (2) Relational knowledge. For example, extra data about the input information that can be used to help resolve conflicts or extra information that can be added to fill gaps in the information to be output (such as "Paris is the capital of France", and "Shell is listed on the London Stock Exchange"). This data can often be sub-contracted out to relational databases. (3) Metalevel knowledge. For example, preferences over sources, or information about the reliability of sources.

[1]This distinction should not be confused with the distinction, familiar from work in description logics, between intensional and extensional knowledge. The former is often described as general knowledge, but this is because it is about *concepts* in the knowledgebase rather than individuals, which are the subject of extensional knowledge. However, intensional knowledge is still knowledge that applies to a particular domain, and so could not be reused across different applications about different subjects. Hence, it does not qualify as general knowledge as we understand it.

(4) Coherence knowledge. For example, which terms are consistent ("rain" and "wind") and which are not ("overcast" and "sunny"). (5) Knowledge about allowable ranges of various values. For example, what variations in temperature, windspeed and humidity are compatible with a set of input reports being seen as not conflicting (such as the the midday temperature in London being 5-10C is an acceptable range, whereas it being 5-25C is not acceptable since it is too vague).

Generic knowledge, on the other hand, is knowledge that can be used in multiple applications. You can draw on extensive research in knowledge representation and reasoning to define generic knowledge with well-understood behaviours.

Drawing on the literature in aggregation functions, you can implement a whole range of aggregation predicates. These include simple functions such as disjunction and conjunction, but also more complex ones such as semantic generalization, that is, finding the most specific term that is general enough to subsume the input terms. There are also many voting functions for resolving conflicts that can be included in the knowledgebase, and these can all take unweighted and weighted forms. Various kinds of preference functions defined over the sources of the input reports can also be used to resolve conflicts when merging. In the case of many of these functions the rules are generic, but they will also rely upon facts that are themselves domain-specific. So for example, both the facts embodying the concept hierarchies used in finding the semantic generalization of different input textentries, and the facts specifying the order of preferences for the sources of reports will be domain-specific. But the rules themselves are generic, and so using functions such as these it is straightforward to compile a library that can be used for resolving conflicts in a variety of applications.

You can also investigate the following types of generic knowledge: (1) Reasoning about time and events and using this for defining an event-based form of aggregation; (2) Reasoning about uncertainty and using this for defining a class of aggregation predicates that take uncertainty into account; and (3) Reasoning about inconsistency and using this for context-sensitive selection of anaggregation function.

10.4 Further aspects to developing a knowledgebase

In this section we deal with some further issues that arise in developing a knowledgebase that were passed over in the previous sections.

10.4.1 Dealing with `null` textentries

As we explained earlier (subsections 9.4.1 and 9.4.2), if an input report does not contain a branch corresponding to the branch in a variable, then the variable is ground with the textentry `null`. This indicates, not a particular kind of value, but rather the fact that it is not known if there is a value. It is important, therefore, that this textentry not be confused with values such as "zero" or "none".

This has implications for the implementation of the knowledgebase. In most cases when a `null` textentry is encountered you will simply want to discount it. For example, if some form of voting function is being used to resolve conflicts, you do not want to tally up the `null`s along with all the other kinds of textentries to see which one wins. Again, if you are determining whether a series of numerical values are all within a certain range you will want to ignore the `null`s.

At the implementation level there are several ways you could do this. One, of course, is to have the code in the knowledgebase proper ignore `null` textentries. Another approach would be to have some code in the interface to the knowledgebase strip out any `null` textentries, perhaps when the queries are being parsed into the format required by the external resource. But note, however, that if you adopt this latter approach, there may well be circumstances in which you do not want to ignore `null` textentries.

For example, you might decide, as we have done in the demonstration application, to resolve conficts by selecting the textentry from the report with the most preferred source. But this is not simply a matter of looking at the sources of all the input reports and selecting the relevant textentry from the most preferred; rather, you have to look at the sources of all the reports that have non-`null` textentries, and select the relevant textentry from amongst those. Clearly in this case, then, you cannot ignore `null`s.

A further consideration is what happens if *all* reports return "null"? What behaviour do you want in this case? Typically you would want the rule in question to fail. But what if that rule was one of a pair, such that if it failed, the other rule succeeds *by default*. (See example 10.4.1) In that case the second rule would result in a `null` being added to the merged report. If that is not the intended outcome for a particular subgroup of rules then you should make sure that when the conditions concerned are ground by textentries that are all `null`s, the conditions fail.

Example 10.4.1 *A pair of rules for merging weather reports dealing with the entry for today's weather. The pair are implemented so that if rule 12 fails, rule 13 will succeed by default.*

```
Rulecode is 12 (Status = optional)

EquivalentTerms(set//weatherreport/today)
AND PreferredTerm(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)

Rulecode is 13 (Status = optional)

Not EquivalentTerms(set//weatherreport/today)
AND Disjunction(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

However, suppose none of the reports has a textentry for today's weather. The result is that a list of `null`s *grounds the variable* `set//weatherreport/today`. *In that case rule 12 will fail, but the first condition of rule 13, which uses a* `sign` *attribute whose value is "negative", will succeed. This means that the second condition in rule 13 will now be ground as* `Disjunction([null, null, null, null], null)`, *and therefore a* `null` *is added as the text entry in the merged report for today's weather.*

If this is not the desired behaviour then rule 13 should be altered so that it no longer uses a negative value for the condition's `sign` *attribute, together with the same predicate as rule 12. Instead, a new predicate,* `NotEquivalentTerms`, *is defined. This predicate succeeds if not all of the textentries are from the same equivalence class and not all of the textentries are* `null`.

```
Rulecode is 13 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND Disjunction(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)
```

10.4.2 Passing on external resource error messages

We saw above how the interface to the knowledgebase must parse the responses of the external resource into Java `Response` objects. (See subsection 10.2.4) But it's worth repeating what should happen if the original query resulted in an error. It would be desirable in such cases to pass on any error messages generated by the external resource so that they could be displayed by the fusion tool.

Suppose, for example, that you are using a Prolog engine as the external resource. Prolog expects any textentries that start with an upper case character to be enclosed in single quotes. The knowledgebase interface

should, then, deal with this. But suppose further that the textentry for the ⟨city⟩ element in an input report is "B'ham" rather than "Birmingham". This yields the query `SameCities(['Birmingham','B'ham'])`, which, because of the apostrophe, results in a syntax error. Of course, if you have adequately tested both the external resource and the interface to it, then your interface should have coped with this. But it's almost impossible to anticipate all possible errors, so you will need some way of informing the user of the fusion tool when such errors occur.

The way to do this is to use the error message generated by the external resource to set the value of the `Response` object's String field `errorMessage`, using the `setErrorMessage` method. Then, if the `errorCode` variable is `true`, the fusion tool will display the associated error message and the user will know why the reports were not merged.

10.4.3 Parsing queries containing "source" variables

If you intend to write rules that use variables that are to be ground by individual input reports, rather than by all of the input reports, then you must remember to engineer the query parser in the interface to deal with this. Suppose, for example, you define a predicate `UsePefrerredSource` in the knowledgebase. Its purpose is to select from amongst a list of textentries, the textentry from the report with the most preferred source. A natural way to do this, then, would be to define it with *three* arguments:

`UsePefrerredSource(Sources, Textentries, X)`

where `Sources` is a list of the sources of the input reports, `Textentries` is a list of their textentries, and `X` is the logical variable that will be bound by the textentry from the report with the most preferred source.

In most cases, when you come to write the fusion rule the first two arguments will be implemented using "set" variables, and of the course the third will be a logical variable. Hence the XML implementation of the rule's condition will have the same arity, three, as the corresponding predicate. It will then be a straightforward matter for the query parser in the knowledgebase interface to parse this condition into the form expected by the external resource. But suppose, instead, that the rule was implemented using individual "source" variables. In that case the number of arguments of the condition of the fusion rule would then depend on the number of input reports expected by the writer of the rule. For example, if the rule was to be used to merge four reports, then there would be four variables for source textentries ("BBC", "Sky", "ITN", etc.), and four more for, say, the windspeed textentries, resulting in a rule condition with an arity of $nine$! However, from the point of view of the knowledgebase, these two queries are the same, they're just expressed in different ways. Hence the knowledgebase expects the same query, with the same arity, and the query parser must be implemented so as to provide this. (See example 10.4.2)

Example 10.4.2 *Suppose we wish to resolve a conflict between the textentries for windpseed by using the textentry from the report with the most preferred source. Suppose, further, that we know there are exactly four input weather reports. We could do this in two different ways, either using eight "source" variables or, as is normal, by using two "set" variables. The resulting two conditions are as follows:*

```
UsePreferredSource(1//weatherreport/source, 2//weatherreport/source,
       3//weatherreport/source, 4//weatherreport/source,
       1//weatherreport/windpseed, 2//weatherreport/windpseed,
       3//weatherreport/windpseed, 4//weatherreport/windspeed, X)

UsePreferredSource(set//weatherreport/source, set1//weatherreport/windspeed, X)
```

Note that as specified by the fusion rules, these two conditions have different arities: the arity of the first is nine, and the arity of the second is three. But, intuitively, these two conditions are asking the same question. So, despite their different arities, they must be parsed by the knowledgebase interface to yield the same query:

`EquivalentTerms([BBC, CNN, Sky, ITN], [10mph, 12mph, 12MPH, 11mph], X)`

This will, of course, enable them to get the same answer.

10.4.4 Negative conditions

The ⟨`conditionname`⟩ element in the fusion rule ML has a `sign` attribute that takes the values "positive" and "negative". (9.2 and 9.4.2) The purpose of this attribute is to negate the condition. What this amounts to is that if a given positive condition succeeds when ground with a given set of input reports, changing the value to "negative" will result in the condition failing. Conversely, if a positive condition fails when ground with a given set of reports, changing the value to "negative" will result in the condition succeeding.

However, for this attribute to have the desired behaviour depends on the knowledgebase being implemented accordingly, and how that should be done depends on the external resource being used. For example, one way to implement negative conditions when the external resource is a Prolog engine is to have the query parser prepend "not" to the query. The query will then fail if and only if the unegated query would have succeeded. Other external resources may require different approaches to implementing this feature.

Implementing this feature offers obvious advantages in terms of the reuse of code in the knowledgebase. But as noted above (10.4.1), it should be used with care. In many cases it may be appropriate to define your own predicates to allow for failure when all the input textentries are `nulls`. In any event, you are not compelled to take advantage of this feature. If you do not, then you should make sure all conditions take the value "positive" because "negative" values will result in a "Not" being prepended to the condition as it is displayed by the fusion tool. This is purely for display purposes, and so will have no effect on the resulting merged report, but it will be very confusing.

Chapter 11

How to develop a fusion rules application

In previous chapters we have looked in detail at the various components of a fusion rules application, and in particular at the fusion rules themselves. Our intention in doing that was to provide not just technical information about those components (such as what are the legal arguments for various actions), but also practical information that will be of real use in building an application.

The purpose of this chapter is to bring together that information to provide a quick guide on how to build a fusion rules application. Obviously there is no algorithm for doing this, and there are other ways to proceed beside the approach that we advocate, but we hope that what follows will be of help in building such an application.

11.1 General considerations

1. An obvious place to start is to decide exactly what is to be merged, and how it is to be merged. Deciding this will tell you what kind of input you will require and what kind of output you expect. Because both the input and the output of the fusion tool are XML documents, try to think here in terms of elements, even sketching out the tree structures of a typical input report and the final output merged report. Doing this will help you to do two further things: (i) you will see the *form* into which you will need to get the input information. (ii) It will give you a framework within which to design the fusion rules. This is because the input report tags will be amongst the variables for the conditions (and possibly for some actions), and the output report tags will be amongst the constants for the actions. Looking at an envisaged input report, then, will tell you what some of the arguments for the conditions will be. In the case of the weather reports demonstration application, you know that the conditions will have arguments such as *temperature*, *humidity* and so on. Similarly, looking at the envisaged output report will tell you some of the arguments the actions will be taking.

2. Once you know what kind of input you require you can set about the task, if need be, of getting it into XML form. For a simple application you could do this by hand, but in any real application this would likely involve some non-trivial information extraction tasks.

3. Having got the input for the application you are now in a position to develop the knowledgebase proper. Looking at the input will not only tell you what the arguments to the knowledgebase queries will be, it will also tell you what kind of knowledge you will need in the knowledgebase. For example, inspecting a wide range of input weather reports will tell you what units you can expect for

numerical values such as temperature, pressure, and windspeed. You can then incorporate conversions between these into the knowledgebase. The input reports will also tell you what kinds of terms you will need to include in equivalence classes for merging non-numerical or qualitative textentries, such as those for today's weather or visibility. Inspecting the ouput report, on the other hand, will tell you what sorts of aggregation functions you will have to include in the knowledgebase. (See 10.3)

Because different kinds of external resource can be exploited as a knowledgebase, we cannot offer advice on how to implement this kind of knowledge. We can, however, repeat the Golden Rule that the knowledgebase should be developed and tested *independently* of the rest of the application so that any bugs can be more easily isolated.

4. Once you have developed the external resource to act as the knowledgebase proper, you can develop the interface between the fusion tool and that resource. This is the Java class (or classes) whose job it is to load the external resource, parse queries from the fusion tool into the format requied by the resource, and parse responses from the external resource into the form required by the fusion tool. Again, since the choice of external resource is up to you, we can offer no advice on how to load it (e.g. if you are using an SQL database you might use JDBC or the JDBC-ODBC bridge to talk to the database). However, now that the choice of external resource has been made you know into what form you will have to parse the queries generated by the fusion tool.

   Other factors that you may want to condsider here are: (i) Do you want to implement the `sign` attribute of fusion rule conditions? (See 10.4.4) (ii) Should you deal with `null` textentries here, or in the knowledgebase proper? (See 10.4.1) (iii) Are you going to use rules containing individual report "source" variables? If so, you will need to parse them appropriately. (See 10.4.3) (iv) Remember to pass on error messages generated by the external resource so that they can be displayed by the fusion tool. (See 10.4.2)

   As with the external resource, you should test the knowledgebase interface with the external resource *prior* to using it with the fusion tool. You can do this by constructing ground `Condition` objects and using them as test inputs. Since the interface is a Java class, probably the best was to do this is by using a unit test harness such as JUnit (available free from www.junit.org). The reason for doing this is that it will isolate any bugs as belonging to the interface and not to the set of fusion rules.

5. The final component to develop are the fusion rules themselves. Knowing what kinds of input and output the application expects, you know what kinds of arguments the conditions and actions should expect. And having developed the knowledgebase you'll also know what names to select for the conditions. We saw from Chapter 9 other, more specific, considerations that you should take into account when developing a set of rules:

   - In general, more rules are better than less rules. This is because the fusion rules are themselves part of the knowledgebase, and so should display that knowledge in as clear a fashion as possible. A rule set that contained just one rule, with a single condition, `Merge`, and lots of actions, would not be very informative and would be hard to read. In particular, it would not tell someone *how* the reports were being merged. So, if possible, you should opt for more rules rather than less. How many rules, though, will depend not just on how many input report elements there are, but also on how many of those elements are being merged *independently* of one another. (9.9)
   - You may not be interested in all of the information in the input reports, in which case there need not be a rule that deals with each input report element. But presumably you will want at least one rule for each *output* report element. (9.8)
   - Typically it will be the case that rules that deal with a single output report element should be such that, for any given set of input reports, at most one rule in such a subgroup should be executed. So, think about the desired behaviour of rules in such subgroups. Note that achieving the correct behaviour will of course depend on how the knowledgebase is engineered. (9.7)
   - Rule order can be important in some cases. Here it helps to have a sketch of the output report to hand so you can check what nodes or elements need to be added in what order. (9.11)

There are also some considerations that you should take into account when writing conditions for fusion rules.

- For the same reason that more rules are, in general, better than less rules, more conditions are better than less conditions — it makes it easier to see how the reports are being merged. (9.4.5)
- Within reasonable limits, and where there is a choice, try to include knowledge in the conditions rather than in the knowledgebase. (See 9.4.4) For example, accepted limits within which textentries such as temperature or windspeed are deemed to be the same can be included in conditions as constants. The advantage of doing this is, once again, that it makes it easy to see how the reports are being merged. It should be emphasized, however, that the overwhelming bulk of the knowledge will be in the external resource.
- Remember that what kinds of arguments are required by a condition is context-dependent. A condition such as `Majority` might require a variable and a logical variable as it occurs in one rule, whereas it might require two logical variables as it occurs in another. (9.4.3)

Finally, here are some points you should keep in mind when writing actions.

- As was the case with conditions, the kind of arguments required by an action are context-dependent. Note that just because a particular kind of argument is allowed for an action does not mean that it is correct in a given context. (9.5.13)
- The order of actions within a rule is usually significant. Make sure that if an action requires a specific node as a target then that node has already been added to the merged report, either by a previous action in this rule, or by a previous rule. If the latter is the case, make sure, of course, that that rule will have executed if required. (9.5.14)
- How much of the merged report structure do you want the `Initialize` action to construct? At a minimum, it must provide the root node. But often it will make sense to construct more than this. For example, conditions involving certain input report textentries are often deemed to be such that if they fail then no merged report will be produced. These conditions are then included in a Foundational rule along with the `Initialize` action. It makes sense, then, for that action to also include those elements in the initial structure of the merged report, though it need not.

  The point to emphasize here, though, is that the kind of action selected for a given rule will often depend on what actions have been used previously in the rule set. For example, if the `Initialize` action adds a node for `today` to the initial structure, then the subsequent rules for today's weather should use the action `AddText` and not, say, `AddAtomicTree`. This point will be illustrated in the following section (11.2).

11.2 The weather reports example

Several of the points just made can be better understood by considering an example. Example 11.2.1 shows the first five rules from one of the rule sets used in the weather reports demonstration application.

Example 11.2.1 *The first five fusion rules from the weather reports demonstration. These rules come from the rule set that deals with conflicts, where appropriate, by using majority voting. Rule 3 is an example where this is used.*

```
Rulecode is 1 (Status = foundational)

AcceptedSource(set//weatherreport/source, X)
AND SameDate(set//weatherreport/date)
AND SameCity(set//weatherreport/city)
IMPLIES Initialize(weatherreport(source, date, city, today, temp(max, min)))
AND AddText(X, weatherreport/source)
AND AddText(1//weatherreport/date, weatherreport/date)
AND AddText(1//weatherreport/city, weatherreport/city)


Rulecode is 2 (Status = optional)

EquivalentTerms(set//weatherreport/today)
AND PreferredTerm(set//weatherreport/today, X)
IMPLIES AddText(X, weatherreport/today)


Rulecode is 3 (Status = optional)

NotEquivalentTerms(set//weatherreport/today)
AND PreferredTerms(set//weatherreport/today, X)
AND Majority(X, Y)
IMPLIES AddText(Y, weatherreport/today)


Rulecode is 4 (Status = optional)

SimilarTemps(set//weatherreport/temp/max, X)
IMPLIES AddText(X, weatherreport/temp/max)


Rulecode is 5 (Status = optional)

NotSimilarTemps(set//weatherreport/temp/max, X)
IMPLIES AddText(X, weatherreport/temp/max)
```

The first rule illustrates several points. (i) It is given the status Foundational because it concerns conditions that check if the input reports are from accepted sources, for the same date, and for the same location. If either of these conditions fails, then there is no point in merging the reports. Because of this, the `Initialize` action includes all of those elements in the initial structure of the output report. However, inspection of the input reports revealed that all weather reports had at a minimum, besides information about source, date, and location, additional information about today's weather and maximum and minimum temperatures. For this reason, those elements were also included in the initial structure. Note that this has implications for the actions that occur in subsequent rules. Rules 2, 3, 4 and 5 all use action `AddText` because the nodes to which their textentries will be attached are already in the output report. (We know that will be the case because if rule 1 fails there will be no report!) By contrast, if the initial structure had included nodes only for source, date, and location, then these later rules would have had to have used an action such as `AddAtomicTree`. (ii) Note also that although the first rule adds nodes for today's weather and temperature to the initial structure, it does not include conditions for those features. This is because, by contrast with source, date, and location, if conditions concerning these features *fail*, that conflict may be able to be resolved and the resulting information included in the merged report. (Alternatively, the fact that there is a conflict might be useful information that we wish to include in an output report.) If, for example, not all of the reports agree on today's weather, that conflict can be resolved in a number of ways, and the result added to the merged report. Hence, conditions dealing with the `today` element should appear in separate rules. (iii) Finally, note that action `AddText` requires different kinds of arguments: in the second action of rule 1 it requires a logical variable as its first argument, whereas in the third and fourth actions it requires a variable.

The later rules also illustrate some important points. (i) The elements in the input reports are in this case merged independently of each other. That is, what textentry is to be added to a particular ouput report

node depends only on the textentries of a single input report node. Rules for today's weather, for example, require conditions only with variables grounded by textentries for today's weather as arguments.

(ii) The rules are grouped into pairs, each pair concerned with a particular output report element. The first rule in each pair executes if a certain condition is met, whilst the second rule in the pair executes if that condition fails. Thus, it is clear that rules 2 and 3 constitute one pair, dealing with today's weather, and rules 4 and 5 constitute another pair, dealing with maximum temperatures. (The rest of the rules in this set are also grouped into pairs.) Note, however, that such subgroups of rules dealing with a single output report element can be arbitrarily large. This is because they can include testing for combinations of any number of conditions, not just, as in this case, a single condition. So suppose for example, two conditions, *p* and *q*, and an output report element *E*. If it is the case that the textentry to be added to *E* depends on all possible combinations of the success or failure of *p* and *q*, then obviously a subgroup of four rules would be required.

(iii) The actions used in these later rules are `AddAtomicTree` and not `AddText` because the nodes to which the textentries will be added are not yet in the merged report.

Building up a set of rules in this way involves, then, keeping in mind the structure of the output report as it grows. Once you have developed a set of rules you will have completed the development of the whole application. As we have tried to emphasize, this set of rules should not be considered as a kind of extraneous attachment to the knowledgebase. Rather, the rules are themselves part of the knowledgebase. This is not only because they may contain knowledge such as voting thresholds or acceptable ranges, but also because they themselves embody or represent knowledge about what it takes to deal with conflicting information in the domain in question.

Chapter 12

The demonstration weather reports application

12.1 Overview

The software comes with a simple weather reports demonstration application. The purpose of this is obviously to allow you to test the fusion tool by merging the weather reports using the rules and the knowledgebase. It also provides an example of a set of fusion rules illustrating some of the points made in Chapter 9.

The demonstration application consists of:

- Two sets of input weather reports, one for December 10th 2002, the other for June 28th 2004. Each set contains five reports. The sources for 10/12/02 are: BBCi, BBC London, CNN, Sky, and The Weather Channel. The sources for 28/6/04 are: The Met Office, The Daily Telegraph, The Times, The Daily Express, and the Guardian. (All the reports are fictitious.)
- Two sets of rules. The first seeks to resolve conflicts by using, where appropriate, majority voting. The second set seeks to resolve conflicts by selecting, from amongst those reports that have textentries, the textentry from the report with the most preferred source.
- A weather knowledgebase. See section 12.2 for details of the knowledge that this contains.

You can run the demonstration by first loading the rules and reports. These are to be found in, respectively, the folders `demonstration/rules` and `demonstration/reports`. Then load the demonstration knowledgebase by choosing the **Select Resource** menu option, and typing:

```
DemoWeatherKBase
```

in the dialog box textfield.

You can experiment with this demonstration in several ways. Most obviously, you can try both sets of rules; either set will work with either set of reports. Equally obviously, you may vary the number of reports you merge (there must be at least two), though if you try to merge reports from different days an error (Error 7) will result because a foundational rule will have failed. More interestingly, it is very easy to write

new weather reports, so long as you respect the weatherreports DTD, and you make sure that the source is defined by the knowledgebase as an accepted source. Alternatively, you can simply edit the textentries of the existing weather reports. In either case, be sure that with numerical values such as windspeeds or air pressures, any units (MPH, mmHg, etc.) you use are known by the knowledgebase, and that any non-numerical textentries ("showers", "heavy showers", etc.) are also defined in the knowledgebase as belonging to some equivalence class.

But note that you cannot edit this knowledgebase. If you want to expand it by adding more predicates you will have to start from scratch, writing both your own knowledgebase interface (in Java), and supplying the knowledgebase proper. In addition, the knowledgebase interface must be placed in the `dynamicClasses` folder.

12.2 The weather knowledgebase

Accepted Sources BBCi, BBC LDN, BBC London, Sky, SKY, Sky News, SKY NEWS, CNN, The Weather Channel, ITN, ITV, Met Office, Meteorological Office, Sun, Mirror", Guardian, Times, Telegraph, Express, Mail, Daily Telegraph, Daily Express, Daily Mail. (All sources may be prepended by "The".)

Date Formats Days: 12, 13, 14, 1st, 2nd, 3rd, 15th, 21st, etc. Months: Jan, Feb, Mar, January, February, March, etc. Years: 1998, 2002, 99, 01, etc. Separators: " ", " / ", " . ", " \ ", " \t ", " |", " - ", " ; ", " : ", " , ". (Dates must be in *dd/mm/yy* order.)

Cities London, Birmingham, B'ham, Manchester, Liverpool, Leeds, Newcastle, Carlisle, Sheffield, York, Cardiff, Swansea, Bristol, Oxford, Cambridge, Norwich, Ipswich, Nottingham, Derby, Leceister, Coventry, Swindon, Exeter, Plymouth, Bournemouth, Southampton, Brighton, Reading, Dover, Edinburgh, Glasgow, Aberdeen, Belfast, Londonderry, Derry, Dundee, Inverness, Stafford, Stoke, Crewe, Wolverhampton, Newport, Worcester, Hereford, Gloucester, Luton, Portsmouth.

Today's Weather Preferred Term = cloud. Equivalence class = {cloudy, cloud, mostly cloudy, overcast, mostly cloud, heavy cloud }

Today's Weather Preferred Term = haze. Equivalence class = {haze, hazy}

Today's Weather Preferred Term = partly cloudy. Equivalence class = {scattered cloud, scattered clouds, intermittent clouds, partly cloudy, p/cloudy, p\cloudy, patchy cloud, sunny spells, sunny intervals, sunny periods, patchy sunshine, partly sunny, some sun, some sunshine}

Today's Weather Preferred Term = stormy. Equivalence class = {storm, storms, stormy, thunder, lightning, thunder storms, thundery storms, thunder and lightning, severe storms}

Today's Weather Preferred Term = showers. Equivalence class = {showers, showery, scattered showers, patchy rain, intermittant rain, drizzle, heavy showers, blustery showers, thundery showers}

Today's Weather Preferred Term = ice. Equivalence class = {ice , icy, frost, frosty, ground frost, black ice, hoar frost}

Today's Weather Preferred Term = snow. Equivalence class = {snow, snow showers, sleet, snow flurries, snowy, snowing, heavy snow, drifting snow, light snow}

Today's Weather Preferred Term = sun. Equivalence class = {sun, sunshine, sunny, mostly sunny, fair, bright}

Today's Weather Preferred Term = rain. Equivalence class = {rain, rainy, wet, precipitation, inclement, mostly wet, mostly rain, heavy rain, prolonged rain, downpour, downpours, frequent downpours, rain/snow}.

Temperature Accepted range is: 3c. Accepted units: C, c, F, f. (Values converted to centigrade.)

Pressure: Absolute Value Accepted range is: 5mb. Accepted units: mB, MB, mmHG, mmHg, MMHG, mmhg. (Values converted to mB.)

Pressure: Direction of Change Preferred Term = rising. Equivalence class = {rising, climbing, increasing}

Pressure: Direction of Change Preferred Term = falling. Equivalence class = {falling, dropping, decreasing}

Pressure: Direction of Change Preferred Term = steady. Equivalence class = {steady, no change, unchanged, same}

Visibility Preferred Term = very good. Equivalence class = {very good, excellent, unlimited}

Visibility Preferred Term = good. Equivalence class = {good}

Visibility Preferred Term = moderate. Equivalence class = {moderate}

Visibility Preferred Term = poor. Equivalence class = {poor, minimal}

Sun Index Rating Preferred Term = low. Equivalence class = {low}

Sun Index Rating Preferred Term = medium. Equivalence class = {medium}

Sun Index Rating Preferred Term = high. Equivalence class = {high}

Windspeed Accepted range is: 10mph. Accepted units: MPH, mph, KPH, kph. (Values converted to MPH.)

Relative Humidity Accepted range is: 10%. Accepted suffix: %.

Preferences Over Sources Ranking (most preferred first) is: Meteorological Office, Met Office, Weather Channel, BBC, BBCi, BBC LDN, BBC London, Guardian, Independent, Times, Telegraph, Daily Telegraph, International Herald Tribune, ITN, ITV, CNN, SKY, Sky, Sky News, SKY NEWS, Mirror, Express, Daily Express, Sun, Mail, Daily Mail. (The ranking is purely for illustrative purposes, and so is completely arbitrary.)

12.3 Explanation of the results

In this section we explain how the rules and the knowledgebase combine to merge the input reports by focusing on a particular example from the demonstration application.

Example 12.3.1 *The result of merging all five weather reports for December 10th 2002 with the rules that seek, where appropriate, to resolve conflicts by majority voting.*

```
⟨weatherreport⟩
        ⟨source⟩ BBCi, BBCLondon, CNN, Sky, Weather Channel⟨/source⟩
        ⟨date⟩ 10th Dec 2002 ⟨/date⟩
        ⟨city⟩ Birmingham ⟨/city⟩
        ⟨today⟩ cloud ⟨/today⟩
        ⟨temp⟩
                ⟨max⟩ 1.66 − 3.0C ⟨/max⟩
                ⟨min⟩ − 2.0 − 0.55C ⟨/min⟩
        ⟨/temp⟩
        ⟨windspeed⟩ 11.0 − 13.0MPH ⟨/windspeed⟩
        ⟨relativehumidity⟩ 63.0 − 64.0% ⟨/relativehumidity⟩
        ⟨pressure⟩
                ⟨absolutevalue⟩ 1021.0 − 1024.23mB ⟨/absolutevalue⟩
                ⟨directionofchange⟩ rising ⟨/directionochange⟩
        ⟨/pressure⟩
        ⟨visibility⟩ good ⟨/visibility⟩
        ⟨sunindexrating⟩ low ⟨/sunindexrating⟩
⟨/weatherreport⟩
```

There are several features to note about this merged report. (i) Despite the different formats of the input dates (they were 10th Dec 2002, 10/12/2002, 10 − 12 − 02, 10/December/2002, 10th Dec 02), they are determined as being the same. (ii) Similarly, despite the different names used for the location (they were Birmingham, B′ham, Birmingham, Birmingham, B′ham they too were determined as denoting the same city. (iii) The textentries for today's weather were: cloudy, cloudy, overcast, cloud, patchy cloud. These are not defined as belonging to the same equivalence class—patchy cloud is defined as belong to the class *partly cloudy*, whereas all the others belong to the class *cloud*—and so we have a conflict. Rule 2 accordingly fails, but rule 3 succeeds. It first finds the preferred form for each textentry in the list. These are: cloud, cloud, cloud, cloud, partlycloudy. A vote is now taken, and since cloud has a majority of the votes, it is selected for inclusion in the merged report. (iv) The maximum temperatures for this day were: 3C, 3C, 7C, 35F, 3C. The accepted range within which reported temperatures are not deemed to conflict is 3C. Before it can be determined whether this condition holds we must first convert the temperatures to a common measure (centigrade). 35F converts to approximately 1.66C, and is within range of three other values. Hence the value for 7C is discarded and the interval between the smallest and the largest values (1.66 − 3.0C) within range is included in the merged report. (v) The values for minimum temperature were: −1C, −1C, 33F, −1C, −2C. Once again unit conversion is required, but this time all the values are within the range, and so the interval between the smallest and the largest (−2.0 − 0.55C) is included in the merged report. (vi) The values for windspeed were 11mph, 13mph, null, null, 38kph. Note that in this case not all the reports have a textentry for windspeed and so in two cases the textentries are recorded as null. These are discarded first before the values are compared. Once again unit conversion is required, but since the accepted range for windspeed is 10mph, one value is discarded (38kph being just out of range at 23.613mph), and the interval is constructed from the remaining two, 11.0 − 13.0MPH. The other elements in the merged report are derived in similar ways.

Appendix A

Error codes

The error code and error name are displayed in the title bar of the warning dialog boxes displayed by the fusion tool software. The message is the short description in the dialog box.

Error 1: Illegal Input Report

Message: Illegal input report. Input report is not well-formed XML.

Explanation: File is either not an XML document or is not a well-formed XML document. Note that, unlike XML documents containing rules, input reports are not checked for validity.

Error 2: Invalid Rule Set

Message: Invalid rule set. Rules do not conform to rulefile DTD.

Explanation: The XML document containing the rules is either not well-formed, or does not conform to the DTD governing fusion rules. This can be found in the file $rulefile.dtd$.

Error 3: Failure to Load Resource

Message: Check that the name entered is the name of the Java class that you wish to dynamically load.

Explanation: There is no Java class file in the directory labelled "dynamicClasses" with the name (excluding the .class extension) corresponding to the name entered by the user. Note that this means that the user has mispelt the name of the class file. (e.g. "MyGlass" instead of "MyClass"). This results in the JVM throwing a ClassNotFoundException. If, instead, the user spells the name of the class correctly, but the *case* of some characters in the name is incorrect (e.g. "Myclass" instead of "MyClass"), the JVM throws a LinkageError. (See Error 4.)

Error 4: Failure to Load Resource (Linkage Error)

Message: Check the CASES of the characters in the name entered are the same as those in the name of the Java class that you wish to dynamically load.

Explanation: There is a Java class file in the directory labelled "dynamicClasses" with the name (excluding the .class extension) corresponding to the name entered by the user, and that class implements the `AnswerQuery` interface. However, the string entered by the user contains characters with the wrong case (e.g. "Myclass" instead of "MyClass") and, as a result, the JVM throws a LinkageError. This contrasts with the situation where the user has misspelt the name of a Java class file. (See Error 3.)

Error 5: Knowledgebase Error

Message: Execution of rule *rule_ code*, condition *condition* resulted in an error occurring in the knowledgebase. (+ *various error messages*)

Explanation: Any error condition arising in the resource loaded for use as a knowledgebase may

be the cause of this error. Two of the most common examples are (a) a condition name specified by a rule does not correspond to any predicate name in the knowledgebase; and, (b), the number of arguments for a condition used to query the knowledgebase does not match the number of arguments expected by the predicate as defined in the knowledgebase. Many other sources of these errors will be specific to particular knowledgebases. For example, a Prolog knowledgebase will expect any textentries starting with an upper case character, or containing whitespace (e.g. "BBC", or "sunny intervals"), to be enclosed in single quotes (thus 'BBC' and 'sunny intervals'), otherwise an error will result. Similarly, if a relational database is being exploited, any SQL syntax errors should result in a knowledgebase error. In such cases please consult the documentation of the proprietary resource. Remember the golden rule is: Develop and test your knowledgebase code PRIOR TO and INDEPENDENTLY of the fusion system.

Error 6: Rule with Invalid Report Number

Message: Rule *rule code*: Invalid report number *report_number*. [There are only *number_of_reports* reports.]

Explanation: Variables can contain natural numbers referencing individual input reports. If the rule contains an integer that is less than 1, or that is greater than the number of input reports, or if the rule does not specify an integer, then this error will result. The error message specifies the rule code of the offending rule, together with the erroneous integer or other character. This should be edited accordingly.

Error 7: Foundational Rule Failure

Message: Reports cannot be merged, Conditions of Foundational Rule(s) *list_of_rule_codes* not satisfied.

Explanation: The rules contain a Foundational rule that fails. If any Foundational rule fails then the reports will not be merged (but see the exception mentioned below). Rules are given Foundational status for one of two reasons: (a) They contain an Initialize action, and/or (b) they are concerned with conditions that are deemed vital in that if they fail, there is no point in merging the reports. If any Foundational rule fails, merging should not proceed because either there will be no initial merged report construced to which textentries and subtrees could be added, or because there is simply no point in merging the reports since they fail to meet some fundamental condition (say that in the case of weather reports, they are for the same location and date.). The exception is that if more than one Foundational rule contains an Initialize action, then only exactly one can succeed because we cannot construct more than one merged report. Hence, the first such rule to succeed results in any further such rules being ignored. To summarize: an error of this type occurrs, and we do not merge the reports, if either a foundational rule without an Initialize action fails or if no foundational rule with an Initialize action succeeds. (There are other conditions concerning Foundational rules and Initialize actions that must also be observed by any set of rules, but these can be checked prior to merging and so are treated as a different error—Error 12. Error 7, by contrast, can only be caught at "merge time".)

Error 8: Unknown Action

Message: Rule *rule_code*: Unknown action *action_name*.

Explanation: The specified rule has an action with an unknown name. Consult section 9.5 for a specification and explanation of legal actions.

Error 9: Illegal Action Argument

Message: Rule *rule_code*: Illegal Action Argument, *action_name*. Expected: *correct_argument_type*. But was: *actual_argument_type*.

Message: Rule *rule_code*: Action *action_name* has too few arguments. Expected: *expected_number_of_arguments* args. But was: *actual_number_of_arguments* args.

Message: Rule *rule_code*: Action *action_name* has too many arguments. Expected: *expected_number_of_arguments* args. But was: *actual_number_of_arguments* args.

Explanation: The specified rule has a known action, but does not have the correct arguments for that kind of action. Either this is because the argument is of the wrong type (e.g. a constant where

a variable or logical variable is expected), or because the rule has too many or too few arguments. Consult section 9.5 on Actions to determine the correct argument types for a given action.

Error 10: Invalid Report Banch

Message: Rule *rule_ code* contains a branch with the element, *element_ name*, that does not match any branch in the merged report.

Explanation: The specified rule contains an action that attempts to add a textentry, element, or subtree to a branch of the merged report that contains an element that does not (yet) exist in the merged report. Check the target branches specified in the constants of the actions of the specified rule to see that all elements are present in the output merged report. If they are, make sure that the rules are in the correct order, so that all output report elements in the target branch have already been added to the merged report by the previous rules.

Error 11: File not Saved

Message: *Various (Java IOException).*

Explanation: A Java IOException has occurred when trying to save the output merged report. This could be the result of a file not found, file corrupted, disk error, or some other condition.

Error 12: Rule Check

Message: *Various*

Explanation: A rule set must conform to various conditions concerning the positioning and occurrence of Foundational rules and Initialize actions. Since these can be tested prior to querying the knowledgebase, these constitute a distinct error to those conditions tested for, and found to have failed, in Error 7. These conditions are: (i) All rule sets must contain at least one Foundational rule. (ii) All rule sets must contain at least one rule with an Initialize action. (iii) Any rule with an Initialize action must be declared Foundational. (iv) A rule may have at most one Initialize action. (iv) All Foundational rules must preceed all non-Foundational rules. (v) All rules with Initialize actions must preceed all rules without Initialize actions. (vi) An Initialize action must be the first action of a rule. The message displayed with this error code will detail which rules have violated which conditions. Note that in addition to these conditions, rule sets must conform to two further conditions with respect to Foundational rules and Initialize actions: (a) All foundational rules must succeed, except (b) if a Foundational rule with an Initialize action succeeds, then any subsequent Foundational rules with an Initialize action must be ignored (i.e. not used to construct the merged report). Because these conditions can only be checked at "merge time", they are treated as a distinct error–Error 7.

Error 13: Invalid Tree Structure for Merged Report

Message: Rule *rule_ code*: Invalid tree structure for merged report: *tree_ structure*

Explanation: The value of the constant specified for an Initialize action does not represent a valid tree structure for the output merged report. Trees are represented by enclosing child nodes within round brackets, and having siblings separated by commas. There may be only one node that lacks a parent. For example, a valid tree would be represented by a string such as "weatherreport(source, date, city, temp(max, min), today)", whereas a string such as "weatherreport(source, date, city), temp(max, min)" would not represent a valid tree because it has two nodes which do not have a parent—"weatherreport" and "temp". (See Section 9.5.1)

Error 14: Failure to Load Resource

Message: Check that the class selected implements the AnswerQuery Interface. + *Various*

Explanation: This error occurs if the class that loads the external resource itself fails to load for any other reason besides those that lead to errors 3 and 4. The circumstance in which an error of this kind will most likely occur is where the name entered by the user does correspond to the name of a Java class in the `dynamicClasses` directory, but that class does not implement the `AnswerQuery` interface. If this happens the message displayed will inform you that a `ClassCastException` has occurred.

Error 15: Invalid Action Variable

Message: Rule *rule_ code*: Action *action_ name*: Variable expected is *correct_ variable_ type*, but

was: *actual_ variable_ type*.
Explanation: The specified action has legal arguments, but one of those arguments is a variable, and that variable is not of the correct type for that kind of action. For example, if the action is `AddTree` and the first argument is, as is allowed, a variable, then that variable must extract a *subtree* and not a textentry. It must also specify an *individual* input report, and not the set of all input reports. If either of these constraints is not observed, then the rules containing this action cannot be used to merge the input reports. Consult section 9.5 on Actions, or Appendix B, to determine the correct variable types for a given action.

Appendix B

Actions and their arguments

Table B.1 summarizes the legal arguments of the twelve defined actions. Often, though by no means always, logical variables can, as a matter of syntax, be used wherever variables are used, and vice versa. This does not mean, however, that as a matter of semantics it makes sense to do this. In any given situation it will only make sense to use either a variable or (exclusive "or") a logical variable; it just depends on what the role of the action is in the context of the rule. A case in point is the first rule from the weather reports demonstration. (See example B.0.2) This rule serves several functions, but focus on the second and third actions. Both add textentries to nodes in the merged report, but whereas the first `AddText` action requires a logical variable as its first argument, the second `AddText` action requires a variable. Given the roles of these actions and the purpose of this rule, although from a syntactical point of view it would be fine to swap these arguments around, it would obviously make no sense to do that.

Example B.0.2 *A fusion rule from the weather reports demonstration. Although the second, third and fourth actions are of the same kind,* `AddText`*, the first of these requires a logical variable as its first argument, whilst the second and third occurrences require a variable. This illustrates the point that although from a syntactical point of view the action* `AddText` *may take either type of argument as its first argument, in any given instance only one kind of argument will be appropriate.*

```
AcceptedSource(set//weatherreport/source, X)
AND SameDate(set//weatherreport/date)
AND SameCity(set//weatherreport/city)
IMPLIES Initialize(weatherreport(source, date, city, today, temp(max, min)))
AND AddText(X, weatherreport/source)
AND AddText(1//weatherreport/date, weatherreport/date)
AND AddText(1//weatherreport/city, weatherreport/city)
```

There are further constraints on what counts as a legal action. These constraints concern the nature of the *variables* that may occur in a particular kind of action. They are summarized in table B.2. For an explanation of why these constraints apply, please consult the subsection for each action in Section 9.5. If a set of rules violates these constraints and those rules are used to merge some reports, then an error (Error 15, see Appendix A) will occur, and the reports will not be merged.

ActionName	1st Argument	2nd Argument	3rd Argument
Initialize	constant	-	-
AddText	variable | logical variable	constant	-
AddNode	constant	constant	-
AddAtomicTree	variable | logical variable	constant	constant
AddTree	variable | logical variable	constant	-
ExtendTree	variable | logical variable	constant	-
AddAtomicTrees	variable | logical variable	constant	constant
RepeatAddNode	constant	constant | logical variable	constant
RepeatAddText	variable | logical variable	constant	-
RepeatAddAtomicTrees	logical variable	constant	constant
AddTrees	variable | logical variable	constant	-
MultiExtendTree	variable | logical variable	constant	-

Table B.1: The twelve defined actions and their legal arguments. The vertical bar signifies disjunction.

Action Name	Variable	
	source | set	extract
AddText	source | set	textentry
AddAtomicTree	source | set	textentry
AddTree	source	subtree
ExtendTree	source	subtree
AddAtomicTrees	set	textentry
RepeatAddText	set	textentry
AddTrees	set	subtree
MultiExtendTree	set	subtree

Table B.2: Where a variable may occur as an argument to an action, constraints apply to the nature of the variable. This table summarizes those constraints. The vertical bar signifies disjunction.

Appendix C

DTD for the fusion rules

This is the DTD for XML documents containing fusion rules. Rule sets must conform to this DTD to be loaded into the fusion tool.

```
⟨!ELEMENT rulefile (rule+)⟩
⟨!ELEMENT rule (condition*, action+)⟩
⟨!ELEMENT condition (conditionname, (constant | variable | logicalvariable)+)⟩
⟨!ELEMENT action (actionname, (variable | constant | logicalvariable)*, constant+)⟩
⟨!ELEMENT constant (#PCDATA)⟩
⟨!ELEMENT variable ((source | set), branch)⟩
⟨!ELEMENT logicalvariable (#PCDATA)⟩
⟨!ELEMENT source (#PCDATA)⟩
⟨!ELEMENT set (#PCDATA)⟩
⟨!ELEMENT branch (#PCDATA)⟩
⟨!ELEMENT conditionname (#PCDATA)⟩
⟨!ELEMENT actionname (#PCDATA)⟩
⟨!ATTLIST conditionname sign (positive | negative) #REQUIRED⟩
⟨!ATTLIST branch extract (textentry | subtree) #REQUIRED⟩
⟨!ATTLIST rule code CDATA #REQUIRED status (foundational | optional) #REQUIRED⟩
```

Figure C.1: The DTD for the fusion rules.

Appendix D

A Prolog knowledgebase

In developing applications ourselves, we have used a proprietary Prolog engine, *WIN-PROLOG* from *Logic Programming Associates Ltd* (`www.lpa.co.uk`), together with their Java interface, the *Intelligence Server*. If you have already have this software, or are interested in purchasing it in order to develop a more powerful knowledgebase, we have included some Java classes that implement the interface from the fusion tool to an LPA Prolog engine as a knowledgebase. (Please refer to Section 4.3, and to the first two sections in Chapter 10 for details of how a knowledgebase fits into the architecture of a fusion rules application.)

There follows a brief explanation of what these classes do, and what you must do if you wish to use them to exploit *WIN-PROLOG* as a knowledgebase. But, to emphasize, in order to do this you must have both *WIN-PROLOG* (version 4322, or later) and the *Intelligence Server* (version 4.320) installed.

`PrologKnowledgeBase` This class loads the *WIN-PROLOG* engine and the *Intelligence Server*, together with the Prolog program (the .pl file) that contains the domain or background knowledge for the application. It also sends queries to the knowledgebase and returns responses from the knowledgebase. The name of the file containing the background knowledge is passed as a parameter to the constructor of this class.

`PrologWeatherKBase` The purpose of this class is to provide a very simple class that you can use as a template (or even just edit) each time you want to specify a new .pl file to use as a knowledgebase. This class simply creates an instance of `PrologKnowledgeBase`, and the constructor of that class takes the name of a .pl file as a parameter. So, each time you develop a new application you should create a class exactly like this one (or, simply edit this class, if you prefer), changing only the String parameter to the constructor `PrologKnowledgeBase`. (See figure D.1.) It is this class that implements the `AnswerQuery` Java interface, and it is the name of this class, or its equivalent, that you should type in the dialog box of the fusion tool when you choose to load a resource. Because you will need to edit this class, in this case we have also provided the source code.

`Reification` This class is called by `PrologKnowledgeBase` to parse the queries generated by the fusion tool into the format required by the Prolog engine.

If you have the *WIN-PROLOG* software and would like to run a weather reports application, you can obtain some sets of reports, and some sets of rules, together with the Prolog file containing the background knowledge by emailing Tony Hunter at a.hunter@cs.ucl.ac.uk or by visiting the website and following the links to the weather reports case study (www.cs.ucl.ac.uk/staff/a.hunter/frt).

```
1 :   public class PrologWeatherKBase implements AnswerQuery {
2 :
3 :       //Instance of class that interfaces with LPA IS.
4 :       PrologKnowledgeBase prolog;
5 :
6 :       /** Creates a new instance of PrologWeatherKBase */
7 :       public PrologWeatherKBase() {
8 :           //The String parameter should be the name of the
9 :           //Prolog (.pl) file with the knowledgebase.
10 :           prolog = new PrologKnowledgeBase("test2.pl");
11 :       }
12 :
13 :       public Response answerQuery(Condition groundCondition) {
14 :           Response r = prolog.answerQuery(groundCondition);
15 :           return r;
16 :       }
17 :
18 :       public void tidyUp() {
19 :           prolog.tidyUp();
20 :       }
21 :
22 :   }
```

Figure D.1: The source code for the class `PrologWeatherKBase`. To develop an application that uses the proprietary software *WIN-PROLOG*, you must edit this class (or write a class like this, changing the name) in the following way: if you have developed a Prolog knowledgebase in a file called `MyKnowledgebase.pl`, then you should replace the String "test2.pl" in line 10 with "MyKnowledgebase.pl", and (re)compile the class.