Dynamic Workflow Pulling
The Strings

Joseph Wilk
<joseph.wilk@imperial.ac.uk>

Project supervisor:
Dr. Alexandra Russo
<ar3@doc.ic.ac.uk>

Second Marker:
Mr. Jim Cunningham
<rjc@doc.ic.ac.uk>
Abstract

Complex HTML form systems can become a mess of code and logic. An example HTML form system could be seen as Amazons on-line ordering system. The world is rapidly changing and so are the environments in which these systems are situated. HTML form systems due to the tangling of code and logic tend to be out of date with the real world. Workflow, the explanation of how a task takes place helps separate the logic and the code. In order to change with the real world Workflow Management Systems provide management and specification of workflow. However, Workflow Management Systems still have their limitations. They tend to have verbose specifications of workflow that are difficult or time-consuming to change. They are a good solution to complex HTML form systems but not good enough. This project takes the HTML form systems as a model and builds a Workflow Management System that uses artificial intelligence planning methodologies and Event Calculus workflow specifications to try to overcome some of the problems of Workflow Management Systems.

The development of the Workflow Management System with AI principles has uncovered interesting issues in modelling situations in the Event Calculus and the problems that need to be overcome to use AI with workflow. The problems and solutions developed in the project cover a wide spectrum of domains, looking at logic programming, server-side languages and getting the two to talk to each other. Areas covered include such interesting topics as typing of HTML to new frameworks for Prolog running as CGI.

Information and the location of the source code are detailed in the Appendix, Section 12.
Acknowledgements

I would like to thank Dr. Alessandra Russo for supervising my project. Her enthusiasm and advice have been invaluable.

I would also like to thank Dr. Murray Shanahan for allowing me to use his Event Calculus planner and for his insightful chats.

Finally, I would like to thank all the Physiotherapists and Pain specialists at Teddington Hospital who without their continued support and help I would never have been in the situation to complete this project.
## Contents

Abstract........................................................................................................................................1

Acknowledgements ....................................................................................................................2

Contents .......................................................................................................................................3

1 Introduction.........................................................................................................................6
  1.1 Overview & Motivation.................................................................................................6
  1.2 Objectives ..................................................................................................................8
  1.3 My Contribution........................................................................................................9
  1.4 Report Structure.......................................................................................................9

2 Background .......................................................................................................................11
  2.1 Overview..................................................................................................................11
  2.2 Event Calculus ........................................................................................................11
    2.2.1 The Frame Axiom Problem .............................................................................13
    2.2.2 Circumscription (Frame Axiom Solution).....................................................14
    2.2.3 Ramification Problem ..................................................................................15
    2.2.4 State Constraints (One Possible Ramification Solution)............................15
    2.2.5 Compound Actions ....................................................................................16
    2.2.6 Reasoning About the Event Calculus .........................................................16
  2.3 Planning ....................................................................................................................17
    2.3.1 Plan Execution ...............................................................................................19
  2.4 Abduction..................................................................................................................20
    2.4.1 Introduction to Abduction.............................................................................20
    2.4.2 Abduction and Event Calculus ....................................................................21
  2.5 Prolog .......................................................................................................................22
    2.5.1 Meta Interpreters .........................................................................................22
  2.6 Issues in Prolog Event Calculus Planners...............................................................23
    2.6.1 Negation in the Context of Incomplete Information.................................23
  2.7 Workflow ..................................................................................................................23
    2.7.1 Workflow Management System (WMS).......................................................24
    2.7.2 Intelligent Workflow Management System (IWMS).................................25
  2.8 Hospital Workflow Model .........................................................................................26

3 System Overview.............................................................................................................28
  3.1 Design Decisions ......................................................................................................28
    3.1.1 How to Deliver the Application? .................................................................28
    3.1.2 Which Server-Side Language to use? .........................................................30
    3.1.3 Which Client-Side Scripting Language to use? .........................................30
    3.1.4 Which Planner to use? ..............................................................................30
    3.1.5 How to Deal With Uncertainty in Planning ..................................................31
    3.1.6 Which Logic Programming Language to use? ...........................................33
  3.2 The System................................................................................................................33
  3.3 The Main Components.............................................................................................35
    3.3.1 Server-Side Domain ....................................................................................35
    3.3.2 Logic Programming Domain .......................................................................36
### 4 Logic programming Domain

#### 4.1 Overview

#### 4.2 Event Calculus Workflow Formalisation

- **4.2.1** Level of Abstraction
- **4.2.2** Specification of the Low Level Plan
- **4.2.3** Advanced Plan Functionality

#### 4.3 ASLDICN Event Calculus Planner

- **4.3.1** ASLDICN Event Calculus Planner Structure
- **4.3.2** Converting to a Conditional Planner
- **4.3.3** Advanced Planner and Event Calculus Features
- **4.3.4** Mapping Partially Ordered Plans into Total Ordered Plans
- **4.3.5** Reducing Loops
- **4.3.6** Knowledge Base Integration

#### 4.4 HTML Type Checking

- **4.4.1** Valid Attribute Value Pairs
- **4.4.2** Valid Children Elements of a HTML Tag

### 5 Server-side Domain

#### 5.1 Overview

#### 5.2 XML Specifications

- **5.2.1** Updating Databases with Form Posted Data
- **5.2.2** Converting XML Specifications to PHP

#### 5.3 Core iWFMS Index Page

- **5.3.1** Workflow Unit State Specification
- **5.3.2** Generating HTML From XML Specifications
- **5.3.3** Group Based Form Access
- **5.3.4** GUI

#### 5.4 Server-Side Planner

- **5.4.1** Plan Representation
- **5.4.2** Plan Processing
- **5.4.3** Plan Execution Architecture
- **5.4.4** Handling Interruptions in Plan Progression
- **5.4.5** Client Preparation Planning and Server Condition Planning
- **5.4.6** Segmentation between the Treatment of Conditional Actions
- **5.4.7** Saving Workflow States for Iterations
- **5.4.8** Plan Execution

#### 5.5 Client-Side Planner

- **5.5.1** Plan Representation
- **5.5.2** Plan Execution Architecture
- **5.5.3** Plan Execution

#### 5.6 Core Libraries

- **5.6.1** Messaging
- **5.6.2** Database Management
- **5.6.3** Session Management
- **5.6.4** XML Processing

### 6 Logic and Server-Side Meeting

#### 6.1 Overview
6.2 Representing Prolog in PHP ................................................................. 117
   6.2.1 Parsing Prolog into PHP ................................................................. 120
6.3 Moving Prolog Plans to the Server-Side Domain ............................... 122
6.4 Prolog Functioning as CGI ................................................................. 124
6.5 Prolog Form Generation .................................................................. 126
6.6 DTD to Prolog Definite Clause Grammar Rules ................................. 127

7 Extensions ............................................................................................ 129
   7.1 Overview .......................................................................................... 129
   7.2 Visualisation of Workflow ................................................................. 129
   7.3 Automatic Documentation Generation ........................................... 134

8 Testing .................................................................................................... 135
   8.1 Overview .......................................................................................... 135
   8.2 Unit Testing ..................................................................................... 135
   8.2.1 PHP Unit Testing ......................................................................... 135
   8.2.2 Prolog Unit Testing ..................................................................... 135
   8.3 Prolog Event Calculus Plan Generation Testing ............................... 136
   8.4 Hospital Scenario Testing ................................................................. 136
   8.4.1 HTML Forms System .................................................................. 137
   8.4.2 Database integrity ........................................................................ 137
   8.5 Integration Testing .......................................................................... 137
   8.6 Performance Testing ....................................................................... 138

9 Evaluation .............................................................................................. 139
   9.1 Overview .......................................................................................... 139
   9.2 Quantitative Evaluation .................................................................. 139
   9.2.1 Performance of Prolog Form Generation ....................................... 139
   9.2.2 Comparison of the Event Calculus Planner and Extensions .......... 140
   9.2.3 Comparison of the Extensions to the Event Calculus Planner ....... 141
   9.2.4 Prolog and PHP Performance ..................................................... 142
   9.3 Qualitative Evaluation .................................................................... 144
   9.3.1 Event Calculus Specifications ..................................................... 144
   9.3.2 iWFMS Evaluation ..................................................................... 150
   9.3.3 Handling Change ........................................................................ 151
   9.3.4 Overall Evaluation ..................................................................... 152

10 Conclusion ............................................................................................ 155
   10.1 Further Extensions ......................................................................... 156

11 Bibliography ........................................................................................ 158

12 Appendix ............................................................................................. 161
   12.1 Source Code .................................................................................. 161
   12.1.1 Obtaining the Source Code ......................................................... 161
   12.1.2 Dependencies .......................................................................... 161
1 Introduction

1.1 Overview & Motivation

What is workflow and what is a WorkFlow Management System (WFMS)?

Workflow has a wide range of definitions and has uses both in the computing domain and in business. It has become a very important way of explaining to computers and people how tasks take place in the real world. The following figure illustrates an example of workflow.

![Figure 1: The workflow of eating toast](image)

The figure represents the model of eating some toast. Note there is an order implied by the workflow, we only eat the toast once we have added the honey. There are also conditions for the progression to new stages. Only if the toast is ready and the honey is found can the honey be added and then eaten.

The world is rapidly evolving especially in the business environment and we quickly find that the workflow no longer represents the actual task. Therefore for workflow to be of use it needs to seamlessly evolve alongside the real world. Workflow also needs an execution engine that allows the workflow to guide and constrain the real world interaction. The engines that facilitate the management of change and execution are referred to as Workflow Management Systems. These systems are nothing new and companies have been selling them for many years with the most popular being IBM’s FlowMark system [8].

Complex HTML form based systems have serious limitations.

WFMSs tend to be specialised for particular domains. An example of a domain that a WFMS could be applied to is the running of a web based system. A web based system consists of a set of HTML forms. Workflow can be used to explain how the forms can be interfaced with and indicate the ordering of dependencies between forms. The workflow is trying to achieve some workflow goal. Despite the suitability of this domain, few WFMSs have been adapted to it. Consequently, we find most solutions for these web based systems use hard coded systems. These systems have a number of problems:
• Changing the system is a complex and arduous task due to the amount of code tangling between the control flow and the system.
• Implementing the control flow is time consuming and error prone. There is no way to gain an overview of how the system works which complicates reasoning.

That is not to say that these systems do not have a number of powerful features.
• Constraining how a user inputs information into a form. This helps ensure the user uses good practices and helps stop them from producing invalid forms.
• The systems are very fast.

WFMSs are good but they are not good enough.

A WFMS would seem to be the answer to the form based systems problems. Unfortunately, WFMSs still have a number of problems.
• Labour intensive to create workflow specifications.
• Labour intensive and conceptual difficult to change the workflow specifications
• The overall WFMS is difficult and sometimes impossible to change and customise.

Primarily the problems occur as the workflow specification facilitates the running of the system and therefore tends to be very detailed with little abstraction. However, WFMS provide a number of very strong features
• Visual editors to edit workflow
• Facilitate the running of an entire system from workflow specifications

Improve WFMSs through making them intelligent

Now initially artificial intelligence planning research seems far removed from the world of business and workflow. Artificial intelligence has always been regarded in the business environment as being vapour ware. Perhaps that is why there seems to be little investigation of how the benefits of planning could be adapted to WFMSs. Only recently have companies such as BT research labs been looking into the advantages of AI planning methodology within WFMSs [19]. The combination of a WFMS and artificial intelligence has been labelled as an Intelligent WorkFlow Management System (iWFMS).

So how is AI planning going to help?

AI Planning in a WFMS brings about the improved ability to deal with new situations and requirements. A more flexible plan can be specified aimed towards being easy to change for the users rather than for running the system. The planner takes the flexible
plan and makes it into a more concrete plan suitable for running the system. One example feature of planning that makes change easier is the levels of abstraction that can be introduced. This means changes at the higher levels can be made without having to know what is happening at the lower levels. In a perfect world which we are still a long way from, an iWFMS would take a description of the world and you would tell it what you want it to achieve and it would work out how to do it. Now this project does not expect to achieve this or propose that such a thing is even possible for workflow. What it does intend to do is investigate what is needed in a workflow specification language using artificial intelligence principles. How the Event Calculus can be used as such a workflow specification language and how plans generated from the specification can be used to run and managed a HTML form based system.

What is the Event Calculus and what does it have to do with workflow?

The Event Calculus is a first order logic formalisation that can be used in AI planning. It has an explicit concept of time. Workflow is all about specifying the order in which tasks can occur. In our example in Figure 1: The workflow of eating toast, eating the toast must not occur until the honey has been added. The Event Calculus with its concept of time provides the facility to explain such workflow situations. Therefore, the Event Calculus seems a good base to talk about workflow and introduce AI planning techniques to generate plans for how workflow should be undertaken.

1.2 Objectives

1. To investigate the level of integration between AI planning, the Event Calculus and workflow.

2. To develop and implement a framework for an Intelligent WorkFlow Management System for HTML form based systems using the Event Calculus and total ordered plans. Providing the Event Calculus representation to model and plan about actions. Modelling the interaction with the user and the HTML form system.

3. To produce an example workflow specification that represents the real world example of a hospital. Looking at the workflow of how patients are admitted, assessed and prescribed drugs. Ensuring that the frameworks and workflow specifications developed are expressive enough to support this workflow example. Ensuring that the workflow specification can successfully be used to govern and run a fully functional HTML form based system.

WFMS are huge in complexity and systems like IBMS FlowMark [8] have taken years to develop. Therefore, the objective is not to produce a complete WFMS but to produce the basic components. The main component of an iWFMS that will be missing from this project is a visual editor to the workflow specifications.

Although the project looks at the specific domain of workflow over HTML forms, the theory could be applicable to any iWFMS.
1.3 My Contribution

This project was aimed at creating the building blocks for an iWFMS for HTML form based systems. The product of the project has been:

- **Workflow specification language**
  Using the Event Calculus and extensions to specify workflow.

- **HTML form typing**
  A typing engine for ensuring that the HTML form element specifications are correct when used in workflow specifications.

- **A Visualisation tool for Event Calculus plans**
  A tool that generates Scalable Vector Graphic [36] graphs for Event Calculus plans.

- **A HTML/PHP iWFMS engine**
  Using the plans generated from the workflow specifications to support the running and management of a system.

- **A JavaScript plan execution engine**
  Facilitates the following of workflow plans in a scripting language that runs while the user is viewing and interacting with a web page.

- **Logic programming running as Common Gateway Interface (CGI)**
  A framework for the use of high-level declarative programming languages functioning as CGI.

- **Logic programming and server-side language interaction model**
  An Interaction model allowing server-side languages used for generating web pages to interact with logic programming languages.

- **A Hospital model working example**
  An example of how the specification can be utilised for a real world scenario in a hospital. Providing the full functionality within the iWFMS to run and manage this system.

1.4 Report Structure

*Background*
This section provides information on a number of areas that were required to be understood before undertaking the project. This covers such topics as planning, the Event Calculus and workflow.

*System overview*
This section looks at the design decisions that were made at the outset of the project and the reasoning behind these decisions. Then an overview of the system architecture is given which details the main components of the project.
1. INTRODUCTION

Logic programming domain
This section examines the interesting problems and solutions that involved Prolog. This includes how the Event Calculus was used to represent workflow and how the Event Calculus planner was developed.

Server-side domain
This section focuses on the interesting problems and solutions that involved PHP. This looks at all the components that were required in order to execute the plans generated in the logic programming domain and the glue that was required to make the system fully functional.

Logic programming and server-side interaction
This section focuses on how Prolog and PHP can interact with each other through a formal object model. Also looking at how Prolog can run as CGI.

Extensions
This section looks at the extensions which were not part of the original goals of the project. One such extension covered is the visualisation of Event Calculus plans.

Testing
This section details the testing methodologies that were used in the project and interesting problems that were uncovered through testing.

Evaluation
This section provides a quantitative and qualitative evaluation of the project. Looking at the performance of a number of components of the iWFMS and looking at the path of the hospital model specification from start to finish.

Conclusion
This section summarises the system, its achievements and its limitations. Then it looks at interesting conclusions uncovered from the project. Finally extensions for further work with the project are given.
2 Background

2.1 Overview

This section looks at a number of concepts that were required in order to undertake this project. This text is aimed at MEng students and will explain these fundamental concepts. Some basic knowledge of logic is expected.

2.2 Event Calculus

The Event Calculus is a first order logic based representation where the underlying ontology consists of actions/events, fluents and time points.

Actions/events: activities which have an effect on the fluents in the world around them.
Fluents: like a Boolean variable that can hold or not hold
Time points: specific time references

The Event Calculus was first introduced by Kowalski and Sergot [10] as a mechanism to formalise about events and their effects. The notation is similar to that of situation calculus except for the explicit representation of time. The notation has varied over its lifetime but we will deal with the specific notation as used by Shanahan [28], within this section. Variables are always lower case and predicates always start with an upper case letter. All variables are regarded as being universal quantified unless stated otherwise.

We use a number of predicates in the language of the Event Calculus:

<table>
<thead>
<tr>
<th>Formula</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiates($\alpha, \beta, \tau$)</td>
<td>Action $\alpha$ initiates fluent $\beta$ at some time point $\tau$.</td>
</tr>
<tr>
<td>Terminates($\alpha, \beta, \tau$)</td>
<td>Action $\alpha$ terminates fluent $\beta$ at some time point $\tau$.</td>
</tr>
<tr>
<td>Releases($\alpha, \beta, \tau$)</td>
<td>Action $\alpha$ releases fluent $\beta$ from the natural law of inertia</td>
</tr>
<tr>
<td>$\tau_1 &lt; \tau_2$</td>
<td>$\tau_1$ occurs before $\tau_2$</td>
</tr>
<tr>
<td>Initiallyp($\beta$)</td>
<td>At the start time point the fluent $\beta$ holds</td>
</tr>
<tr>
<td>Initiallyn($\beta$)</td>
<td>At the start time point the fluent $\beta$ does not hold</td>
</tr>
<tr>
<td>Happens($\alpha, \tau_1, \tau_2$)</td>
<td>Action $\alpha$ occurs at time point $\tau_1$ and finishes at time point $\tau_2$.</td>
</tr>
<tr>
<td>Happens($\alpha, \tau_1$)</td>
<td>A two variable version of Happens which is regarded as being: Happens($\alpha, \tau_1, \tau_1$)</td>
</tr>
<tr>
<td>HoldsAt($\beta, \tau$)</td>
<td>Fluent $\beta$ holds at time point $\tau$</td>
</tr>
<tr>
<td>Clipped($\tau_1, \beta, \tau_2$)</td>
<td>Fluent $\beta$ is terminated between $\tau_1$ and $\tau_2$</td>
</tr>
<tr>
<td>Declipped($\tau_1, \beta, \tau_2$)</td>
<td>Fluent $\beta$ is initiated between $\tau_1$ and $\tau_2$</td>
</tr>
</tbody>
</table>

Figure 2: The Event Calculus predicates
2. BACKGROUND

So what does the Event Calculus do?

<table>
<thead>
<tr>
<th>Plan (temporal ordering, sequence of Happens(…)) and a description of the initial state</th>
<th>Event Calculus axioms (domain independent axioms)</th>
<th>Goal (Sequence of HoldsAt(…))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain dependent sentences. Information about the effects of actions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: The Event Calculus

The Event Calculus has been used to model a number of applications such as Kowalski and Sergots initial database domain [10], network security policies [3] and representing protocols [38] among a few.

Event Calculus representations consist fundamentally of:

1. Domain dependent sentences (Σ)

   Sentences which explain the effects of actions and provide information about initial states.

   - \( \text{Initiates}(α, β, τ) \)  
     Action \( α \) initiates fluent \( β \) at some time point \( τ \)

   - \( \text{Terminates}(α, β, τ) \)  
     Action \( α \) terminates fluent \( β \) at some time point \( τ \)

2. Domain independent axioms (EC)

   The core behind Event Calculus. Axioms which help determine which fluents hold and do not hold at specific time points.

   - \( \text{Clipped}(t_1, f, t_4) \) \iff \( ∃ a, t_2, t_3 [ \text{Happens}(a, t_2, t_3) \land t_1 < t_3 \land t_2 < t_4 \land [ \text{Terminates}(a,f,t_2) \lor \text{Releases}(a,f,t_2) ] ] \) (EC1)

   - \( \text{Declipped}(t_1, f, t_4) \) \iff \( ∃ a, t_2, t_3 [ \text{Happens}(a, t_2, t_3) \land t_1 < t_3 \land t_2 < t_4 \land [ \text{Initiates}(a,f,t_2) \lor \text{Releases}(a,f,t_2) ] ] \) (EC2)

   - \( \text{HoldsAt}(f, t) \) \iff \( \text{Initiallyp}(f) \land \neg \text{Clipped}(0,f,t) \) (EC3)

   A fluent holds at some time \( t \) if it was initially true and has not been terminated (clipped)

   - \( \text{HoldsAt}(f, t_3) \) \iff \( \text{Happens}(a, t_1, t_2) \land \text{Initiates}(a,f,t_1) \land t_2 < t_3 \land \neg \text{Clipped}(t_1, f, t_3) \) (EC4)

   A fluent holds at some time \( t_3 \) if an event happens before \( t_3 \) which initiates the fluent and the fluent is not terminated during the event (clipped).

   - \( \neg \text{HoldsAt}(f, t) \) \iff \( \text{Initiallyn}(f) \land \text{Declipped}(0,f,t) \) (EC5)
A fluent does not hold at some time $t$ if it initially held but was terminated (declipped)

$$
\neg \text{HoldsAt}(f, t_3) \leftarrow \text{Happens}(a, t_1, t_2) \land \text{Terminates}(a, f, t_1) \land t_2 < t_3 \land \neg \text{Declipped}(t_1, f, t_3)
$$

A fluent does not hold at some time $t_3$ if an event happens before $t_3$ which terminates the fluent and the fluent is not initiated during the event (declipped).

Here I have used *Initiallyn* and *Initiallyn* as we shall regard time as non-negative. The conjunction of the domain independent axioms will be represented as EC.

3. **Goal (Γ)**
A finite conjunction of *HoldsAt(…)* predicates and optionally *Happens(…)* clauses, indicating specific times certain events occurred.

4. **Narrative (Δ)**
A finite sequence of *Happens(…)* predicates and temporal orderings: $t_1 < t_2$. Meaning that $t_1$ occurs before $t_2$.

5. **Initial situation (Δ₀)**

- *Initiallyn*(β) Fluent β initially holds at the start time point
- *Initiallyn*(β) Fluent β initially does not hold at the start time point

These are used to describe the state of fluents at the initial time. The initial situation is not mandatory as in certain models there may be no knowledge about the initial situation.

6. **Uniqueness of names (Ω)**

UNA[action1,action2] Action1 and action2 are unique

This defines a common sense rule that actions are unique and are not identical to other actions. This is achieved through a simple conjunction specifying for each action that it is not equal to any other action. We formalise this as UNA[action1,action2]. I shall use $\Omega$ to represent these in our Event Calculus formula. Note that this is not mandatory, in some models certain actions may not be in the UNA as the modeller wants the possibility that they are identical to each other.

Therefore, we can know formalise our graphical description of the Event Calculus in Figure 3: The Event Calculus:

$$
\sum \land (\Delta_0 \land \Delta) \land \text{EC} \land \Omega \neq \Gamma
$$

### 2.2.1 The Frame Axiom Problem
This problem represents unexpected effects of actions. As an example to illustrate:

We have the actions write, publish and winAward (which does nothing so we specify nothing for it).

Initial situation $\Delta_0$:

\begin{align*}
    \text{Initially}(\text{offline}). \\
    \text{Initially}(\text{live}).
\end{align*}

Domain dependent sentences $\Sigma$:

\begin{align*}
    \text{Initiates}(\text{write}, \text{offline}, t).
    \\
    \text{Initiates}(\text{publish}, \text{live}, t) :&\text{- HoldsAt}(\text{offline}, t).
    \\
    \text{Terminates}(\text{publish}, \text{ offline}, t).
\end{align*}

Narrative/plan $\Delta$:

\begin{align*}
    \text{Happens}(\text{write}, t_0), \text{Happens}(\text{winAward}, t_1), \text{Happens}(\text{publish}, t_2), t_0 < t_1 < t_2 < t_3
\end{align*}

We would expect this to hold for our $\Delta$:

$$\Sigma \land \Delta_0 \land \Delta \land \text{EC} \land \Omega \models \text{HoldsAt}(\text{live}, t_3)$$

But it does not!

Here the action write initiates the offline fluent. Despite our lack of definition of the winAward action there is no reason why it could not have some effect on fluents such as offline. We have not mentioned explicitly the fluents that winAward does not change. So winAward could make offline not hold and thus the publish action could not go ahead and the goal is not reached.

There are a few ways to solve this problem but I shall deal specifically with the most common approach using circumscription.

### 2.2.2 Circumscription (Frame Axiom Solution)

Circumscription is a non-monotonic formalisation which represents the common sense rule of inertia. This being that a fluent is assumed to be true until something explicitly makes it not hold. Alternatively, if a fluent does not hold it is assumed to not hold until something explicitly makes it hold.

Circumscription can be viewed as the minimisation of the possible extensions of specified predicates. The key to the solution with the Event Calculus is the applying of circumscription to $\Sigma$ and $\Delta$ separately. With $\Sigma$ we want to ensure that the actions do not have any unexpected effects. With $\Delta$ we want to ensure that events (actions at specific time points) do not have any unexpected effects. $\Sigma$ and $\Delta$ will be horn clauses in most examples we are interested in and thus the problem can be reduced to
2. BACKGROUND

predicate completion. The full details are beyond the scope of this report but can be found in [14].

Circumscription helps create completeness in our model. It enables us to take a closed world approach. The circumscription of $\Sigma$ minimising predicate $p$ is written as: $\text{CIRC}[\Sigma; p]$ . We can now rewrite our Event Calculus model as:

$$\text{CIRC}[\Sigma; \text{Initiates}, \text{Terminates}, \text{Released}] \land \text{CIRC}[\Delta_0 \land \Delta; \text{Happens}] \land \text{EC} \land \Omega \not\models \Gamma$$

Note that circumscription is not applied to the temporal orderings of $\Delta$

It is important to note that when we come to abducting plans with the Event Calculus the circumscription is not applied to the $\text{Happens}$ predicates until there is a complete plan. This is due to the fact that we need an open world view on these predicates to hypothesise their possible values. Circumscription would give a closed world view and thus fail when we want to abduce.

2.2.3 Ramification Problem

The ramification problem is the frame problem but with indirect actions. We have seen previously how circumscription was used to ensure actions/events do not have any unexpected effects. Here we have the problem that actions may have indirect effects not explicitly mentioned in the axioms. Indirect actions could be represented through direct actions but it is desirable to be able to model them explicitly in the Event Calculus as it can produce substantially smaller and more modular representations.

2.2.4 State Constraints (One Possible Ramification Solution)

State constraints provide an effective way to represent the indirect effect of actions. It is important to note that state constraints are not the only way to represent indirect actions. Other possibilities are Effect Constraints and Causal Constraints which are mentioned in detail in [29]. In some models state constraints may not be appropriate. Such an example is demonstrated by the “walking turkey shoot” [2].

State constraints are $\text{HoldsAt}(f,t)$ predicates that are quantified universally over all time. For example:

$$\text{HoldsAt}(\text{published},t) :\neg \rightarrow \text{HoldsAt}(\text{empty},t)$$

This says whenever published holds empty must not hold at the same time.

We shall represent state constraints as $\Psi$. This gives us our final Event Calculus formula:

$$\text{CIRC}[\Sigma; \text{Initiates}, \text{Terminates}, \text{Released}] \land \text{CIRC}[\Delta_0 \land \Delta; \text{Happens}] \land \text{EC} \land \Psi \land \Omega \not\models \Gamma$$
Care needs to be taken when using state constraints. As defined in the EC axioms a fluent initiated by an event cannot then be terminated indirectly by a state constraint unless the fluent was released previously and a fluent terminated by an event cannot then be initiated by a state constraint again unless released previously. Also fluents $\text{Initially}p(f)$ cannot be terminated indirectly and fluents $\text{Initiallyn}(f)$ cannot be initiated indirectly.

## 2.2.5 Compound Actions

Compound actions are actions composed of other actions. An example of a compound action is:

$\text{Initially}(\text{full})$

$\text{Initiates}(\text{publish}, 	ext{live}, t) : \neg \text{HoldsAt}(\text{full}, t)$

$\text{Initiates}(\text{email}, \text{book-sent}, t) : \neg \text{HoldsAt}(\text{full}, t)$

$\text{Happens}(\text{publishBook}, t_1, t_2) : \neg \text{Happens}(\text{email}, t_1) \land \text{Happens}(\text{publish}, t_2) \land$

$t_1 < t_2 \land \neg \text{Clipped}(t_1, \text{email}, t_2)$

Here we have the compound action $\text{publishBook}$ which consists of two other actions $\text{email}$ and $\text{publish}$. We require the $\neg \text{Clipped}$ to ensure that no other event occurs within the sub-events that causes fluents required not to hold. We see the usefulness of how an event can be said to occur over a certain time span allowing further events to occur within that time span.

## 2.2.6 Reasoning About the Event Calculus

With the Event Calculus there are a number of possible reasoning methods. With all these methods $\Delta_0$ is not always needed as a certain degree of uncertainty about the initial situation might exist.

Deduction (e.g. prediction/projection)

Given a narrative ($\Delta \land \Delta_0$) and a description of the effects of actions ($\Sigma$) work out through the EC axioms what the goal ($\Gamma$) could be.

Induction (e.g. theory formation/learning)

Given the goals ($\Gamma$) and a narrative ($\Delta \land \Delta_0$) work out with the EC axioms the effects of actions ($\Sigma$).

Abduction (e.g. planning/diagnosis)

Given a description of the effects of actions ($\Sigma$), the goals ($\Gamma$) and the initial state ($\Delta_0$) work out with the EC axioms the plan ($\Delta$).

We will be focusing on planning with the Event Calculus and thus look at abduction.
2.3 Planning

A plan is a sequence of actions that allows you to achieve a desired goal. There are total ordered plans and partial ordered plans. Total ordered plans are a sequence of actions that is totally ordered. No parallel execution is possible with these plans.

![Figure 4: Total order plan](image)

Partial ordered plans are those that consist of a partially ordered list of actions. Actions are either ordered before or after another and some actions are unordered. This does support parallel actions.

![Figure 5: Partial order plan](image)

Planners that produce partial ordered plans have to deal with threats. Referring to Figure 5 a threat could occur if action C required a post-condition p1 and action A makes not p1 hold. Resolution of threats is achieved through demotion or promotion. In this example the resolution through demotion on A would be used indicating that B and C have to occur before A. Resolution by promotion would have said A occurs before B and C but this would only work if B made p1 hold (A would make p1 not hold and B does not make p1 hold so C’s precondition is not met).

A Hierarchical plan is a plan that uses levels of abstraction where each level can provide greater detail for the relevant actions in the previous abstraction level. Plans can consist of abstract actions which refine into more details.
The plan publish book consists of write book, proof and send to publisher. Notice that we can now refine the plan by looking at the abstraction write book and seeing at another level of abstraction a more detailed plan for writing the book.

Within planning there are a number of ways of dealing with incomplete information. Conditional planning, which is also referred to as contingency planning, provides one solution. Certain actions can be defined as conditional actions. A plan is generated for every possible condition and when the plan is executed the executee through sensing the world takes different actions based on the conditional plan and the value of the sensor.

An example of a plan with conditional actions is as follows:
After submitting the drugs we reach a decision point in the plan. The conditional actions are either drugs are acceptable or drugs are incorrect. One has to be selected and that path will be followed.

Another method of dealing with uncertainty is Re-planning. This is where planning is interleaved with gaining information about the environment and using this information to change the plan.

Finally there is also a method called Probabilistic planning. A plan is generated for the most probable plan paths required.

Fundamental to planning in AI is developing a notation to represent actions and change. We have already encountered one form of representation, the Event Calculus.

### 2.3.1 Plan Execution

As well as generating plans there needs to be a way of executing plans. A formalisation exists which is commonly used to describe the architecture of a robot and how it interacts with its environment. Multi-agent systems [4] deals with intelligent like programs called agents. The formalisation has been used to express how an agent interacts with its environment using plans. A number of different derivations are possible but we shall focus on the formalisation that indicates an agent with memory and plans.

The environment represents a set of states.

\[ S = \{s_1, \ldots, s_k\} \]

\( P \) is the set of percepts that the agent can sense from the environment.

\[ P = \{p_1, \ldots, p_k\} \]

\( A \) is the set of actions that the agent is capable of.

\[ A = \{a_1, \ldots, a_n\} \]

\( M \) represents the memory of the agent

The environment can be seen as a function that maps an action and a state to a new state.

\[ \text{env}: A \times S \rightarrow \text{powerset}(S) \text{ (set of all subsets)} \]

The see function maps the state of the environment to a set of percepts

\[ \text{see}: S \rightarrow P \]

The action function maps a set of percepts to an action.

\[ \text{action}: P^* \rightarrow A \]
The response function maps memory and a set of percepts to a new memory and an action.

\[ \text{response: } M \times P \rightarrow M \times A \]

The selectPlan function maps memory and the plans to a plan

\[ \text{selectPlan: } M \times \text{Plans} \rightarrow \text{Plans} \]

![Figure 8: Agent architecture formalisation](image)

## 2.4 Abduction

### 2.4.1 Introduction to Abduction

We will look at abduction in a general sense. Abduction is a means of reasoning which tries to infer the case from the general rule and result. Abduction is non-deterministic, fallacious and non-monotonic. Abduction is fallacious as the explanations it provides might be true rather than always true. It is non-monotonic as we can find out something true and then by adding new elements to the previous domain we can find what was previously true no longer holds with the new domain. Finally it is non-deterministic as at certain steps in the computation there is more than one choice that could be followed.

Abduction Example:

\[
\begin{align*}
published\text{-}book & \leftarrow printed \\
published\text{-}book & \leftarrow \text{uploaded-onto-cd} \\
in\text{-}bookstore & \leftarrow \text{published}\text{-}book
\end{align*}
\]

If we know \textit{in\text{-}bookstore} two possible explanations/hypothesis as to why this holds are:

1. \{\text{printed}\}
2. \{\text{uploaded-onto-cd}\}
We can see it is fallacious as the explanation printed does not mean that the book was printed, just that that’s one possibility. If we accept uploaded-onto-cd and then add new elements to our domain such as:

*Integrity constraint → [ uploaded-onto-cd ]* (Integrity constraints are explained later)

We find that our explanation no longer holds for this new domain. This demonstrates the non-monotonic property. We obviously see it is non-deterministic as there is more than 1 explanation.

We want to limit the explanations we give in the abductive process. This would help prevent explanations being generated that were just other effects. This is achieved through the use of abducibles. Only abducibles should be used in the explanation. Abducibles can be seen as the possible hypotheses or possible causes or assumables. Being able to specifically specify the abducibles allows explanations to be tailored to their domain.

Another element that is used in abduction is a set of integrity constraints. These help refine the abductive reasoning process by providing restrictions on the explanations we can have. This helps limit the number of explanations ruling out invalid ones. Integrity constraints can also be used to represent differences from the normal and represent negative information. One possible example of integrity constraints could be the representation of certain laws or principles that exists in the problem domain.

Generally the abductive process can be seen as two stages, the abduction stage and the consistency stage. In the abduction stage an abducible is added to the explanation and then the consistency stage checks that the explanation is consistent with the specification and the integrity constraints. If not the abducible is not added to the explanation.

We can formalise abduction in logic as:

\[
\text{Given } T (\text{theory}), O(\text{observation}), \text{IC(}\text{Integrity constraints)}
\]
\[
E \text{ is an explanation such that}
\]
\[
1. \ T \cup E \not\models O
\]
\[
2. \ T \cup E \text{ satisfies IC}
\]

The explanation should only consist of abducibles and it should be minimal.

### 2.4.2 Abduction and Event Calculus

We want to find \( \Delta \) for \( \Gamma \) such that the following holds:

\[
\text{CIRC}[\Sigma; \text{Initiates}, \text{Terminates}, \text{Released} \land \text{CIRC}[\Delta_0 \land \Delta; \text{Happens}] \land \text{EC} \land \Omega \land \Psi \models \Gamma}
\]
2.5 Prolog

For our discussion of Prolog we assume Sicstus syntax. In Prolog we will refer to the temporal ordering t1 < t2 as before(t1,t2).

2.5.1 Meta Interpreters

Prolog has a powerful ability to model Meta interpreters. A Meta interpreter interprets a Meta language. The object language is the language we use to try and describe the domain we are interested in. A Meta language is a language for talking about items in the object language. A formal definition is given:

“A meta interpreter is an interpreter that uses language features of the interpreting language to directly implement behaviour of the interpreted language”

Krishnamurthi [11]

So if we were using Prolog the interpreting language would be Prolog and the interpreted language would be whatever we where trying to describe and reason about. Here we shall create a meta-interpreter that mimics a basic version of Prolog’s own execution strategy. Therefore, we are using the interpreting language of Prolog to implement the behaviour of the interpreted language Prolog! This example is thus not a realistic example but demonstrates the power of Meta interpreters.

```prolog
demo([]).
demo([Goal | GoalList ]) :-
    clause(Goal, Conditions),
    append(Conditions, GoalList, NewGoalList),
    demo(NewGoalList).
demo([ \\+ (Goal) | GoalList]) :-
    \\+ demo([Goal]),
    demo(GoalList).
```

Through using:
```
:- dynamic predicate/arity.
```

Prolog can access the dynamic predicate using `clause(Goal,Condition)` to match Goal to the head of the clause and Condition to the tail of the clause. e.g.

```prolog
:- dynamic initiates/3.

initiates(action,fluent,timepoint) :-
    holds_at(fluent2, timepoint).
```

Clause(Goal,Conditions) in this example would bind Goal to:

`initiates(action,fluent,timepoint)`

and conditions to:

`holdsAt(fluent2, timepoint)`
2.6 Issues in Prolog Event Calculus Planners

A number of Prolog Event Calculus planners have been proposed which are based on ASLDICN (Abductive SLD with Integrity constraints and proof by Negation). We will cover some of the issues dealt with within some of these Prolog programs.

2.6.1 Negation in the Context of Incomplete Information.

The negative axioms EC(6) and EC(5) are not horn clauses.

\[ \neg\text{holdsAt}(a,t) \]

It is desirable to make them horn clauses as this allows us to maintain the ability of the predicate calculus in dealing with incomplete information. We make them horn clauses by removing the negation and replacing it with:

\[ \text{holdsAt}(\neg(a),t) \]

We also now no longer need to use \textit{Initiallyp} and \textit{Initiallyn}.

\textit{initiallyp}(f) replaced by \textit{initially}(f)
\textit{initiallyn}(f) replaced by \textit{initially}(\neg(f))

Note that Prolog takes a closed world assumption. It uses proof by negation which is not identical to logical negation. If it fails to prove negation succeeds. In this domain where we want to generate a plan we have incomplete information. Specifically about certain fluents holding and not holding at certain times and temporal orderings that are not explicitly in the model. Therefore we really want to create our own way of dealing with negation as failure. The Prolog program maintains a list of negative goals. Whenever something is added to the residual the negative goals are check to see if there are any that no-longer hold due to this addition.

To disprove negative goal \texttt{holdsAt}(\neg(f), t_2) it is sufficient to prove \texttt{holdsAt}(f, t_2).
To disprove negative goal \texttt{holdsAt}(f, t_2) it is sufficient to prove \texttt{holdsAt}(\neg(f), t_2).

If we encounter a negative \texttt{holdsAt}(\neg(f), t_2) goal we want to try and prove \texttt{holdsAt}(f,t_2). We have incomplete information about the \texttt{holdsAt}(f,t_2) predicate and thus need someway to achieve it. We can do this by adding the \texttt{holdsAt}(f,t_2) predicate into the residual of abducibles (the plan). A check must be made that this addition to the residual is consistent as with every addition to the residual. The similar idea can be applied to the \textit{before} predicate. When we encounter the negative goal \texttt{before}(x,y) it is sufficient to try and prove \texttt{before}(y,x).

2.7 Workflow

Workflow is the relationships between all different stages/tasks of a project from start to finish. It allows the flow of tasks or information from one user to another based on
conditions. The stages of workflow are referred to as activities or workflow tasks. An activity has to interact with either a human or a machine to achieve execution. Workflow is often referred to as modelling the business process where business process is simply what is supposed to happen to achieve a business objective. Since workflow has been hugely popular in business it has tended to adopt notation specific to business. Workflow is very diverse and can be used to represent numerous relationships and domains.

The following figure looks at some examples of workflow.

![Workflow examples](image)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task A occurs then Task B occurs and finally task C occurs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are two different types of workflow commonly referred to as, Dataflow and Control flow. Dataflow workflow is generally associated with a high level of abstraction but it does not have an explicit representation of time. Dataflow is used within the grid architecture of ICENI [7]. Control flow workflow is associated with low level scripting and has an explicit representation of time. Control flow is used to co-ordinate web services.

The Workflow Management Coalition was established in 1993 and has been working on providing standards in systems that deal with workflow. Their main goal is to achieve interoperability between different workflow systems.

### 2.7.1 Workflow Management System (WMS)

Workflow management systems are systems which allow dynamic workflow through management of the workflow relationships and activities and the execution of workflow. They also provide the ability to define new workflow. The system controls the order which activities take place and it interprets the process definitions (see below). It interacts with a number of participants in the workflow process. There are a number of commercial WMS’s such as:

- FlowMark (IBM)
- Lotus Notes (IBM/Lotus)
- Ad hoc
- WorkMAN (Reach Software)

The Workflow Management Coalition has devised a workflow framework.
Relevant parts of this model for this project have been defined in more detail below.

Process definitions
This is the representation of the workflow and the relationships. Often in conventional WMS this is referred to as the model of the business logic or the workflow specifications. The representation has to be geared towards the automated processing by the WMS. It thus tends to be quite verbose.

Workflow engine
The unit that is responsible for the execution and scheduling of the workflow. It works from the process definitions.

Workflow Enactment Service
Responsible for dealing with communication with external applications and optionally managing multiple workflow engines. It is responsible for providing a service to manage, execute and edit workflow instances.

Workflow Client application.
The client who interacts with the Workflow enactment service through the workflow API.

2.7.2 Intelligent Workflow Management System (IWMS)

IWMS are similar to WMS but they incorporate Artificial intelligence. Karen Myers & Pauline Berry [20] identify three main AI areas directly relevant to Workflow management.
1. Reactive control
   Applied to achieve intelligent process management. A reactive controller
   performing execution for specifically defined tasks and dealing with unexpected
   events.
2. Scheduling
   Applied to Resource and task allocation
3. Planning
   The ability to deal with new situations and requirements.

One of the groups working on an IWMS are building what they call a Continuous
Planning and Execution Framework (CPEF). The main concept of this system is the
idea that plans are open. With the ability to be refined and adapted to deal with new
information and requirements. The main planning theory is based on Hierarchical task
network planning. This planning mechanism is based on the idea of planning at
incremental levels of abstraction until a set of executable tasks is produced.

2.8 Hospital Workflow Model

Within a hospital there are a number of complex interactions and roles within many
activities. It provides the perfect domain to provide an example of the workflow for a
process. In the hospital model we will specifically focuses on the path of a patient
admitted to a hospital and how their drug medication is prescribed, checked and
administered. This is a real model discovered by research into a number of hospitals
and from talking to a number of hospital staff.
This is a process which is not computerised and as a result tends to be slow and cumbersome. There would be great benefits for using an iWFMS to help run the process. For example in the current model the pharmacist comes up to the wards once during the day for each ward. Via a computerised solution the pharmacist could see all the drugs through the iWFMS without having to go to the wards. Also in the NHS processes are continual being refined to promote efficiency and effectiveness. Through using an iWFMS the workflow could be changed and adapted very easily to meet new requirements.
3 System Overview

3.1 Design Decisions

A number of fundamental design decisions had to be made before undertaking the project. These decisions were based on the background reading and further research and experimentation of the different available avenues. In this section we look at these fundamental design decisions and the justification for why those decisions were made.

3.1.1 How to Deliver the Application?

The requirements for the application are that it can generate HTML forms. A centralised source is required to store the workflow since it would need to be shared by a number of users at different computer terminals.

*The Considered solutions*

1. Standard Desktop application
A central server is used to store the workflow. An application running at the client provides access to this server allowing the editing and adding of workflow. The application also provides a conversion from the workflow stored at the central store to HTML forms.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system interoperability achievable through using Java.</td>
<td>It is not possible to produce HTML forms within the application without having to produce some form of web browser. The other option is to generate static HTML forms but this would mean the forms would have to be regenerated at all the clients whenever the workflow changed.</td>
</tr>
<tr>
<td>A wide range of languages are possible each with strong libraries and development environments.</td>
<td>Although possible, languages used in application building are not naturally geared towards dynamic generation of HTML forms.</td>
</tr>
<tr>
<td>Jasper library provides an API to interact with Sicstus Prolog.</td>
<td>There are security implications of generating workflow plans at the client and delivering to the central server. The client could manipulate the plan generation process on the client-side producing erroneous plans.</td>
</tr>
<tr>
<td>The interaction with Prolog is performed on the client offloading the load from the server.</td>
<td>Every client requires Prolog to be installed on their machines.</td>
</tr>
</tbody>
</table>
2. Java Applet
Rather than an application, a Java applet is used to provide management of the workflow at the central server and generation of the form system through the applet.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The interaction with Prolog is performed on the client offloading the load from the server.</td>
<td>Security restrictions when running as an applet mean that it would not be possible to use the Jasper library to interact with Prolog.</td>
</tr>
<tr>
<td>Forms can be delivered through applets but there performance is particularly slow.</td>
<td>Every client requires Prolog to be installed on their machines.</td>
</tr>
</tbody>
</table>

3. Web based Server-side Application
The system would be based on a web server using a server-side language.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralised management is simple since all requests are directed to the same server.</td>
<td>There is a load on the server since Prolog has to be run on the server.</td>
</tr>
<tr>
<td>Web based applications are specially orientation to the dynamic generation of HTML and Forms.</td>
<td>There is no support for interacting with Prolog.</td>
</tr>
<tr>
<td>Since all the code is located and run at the server all client operating systems and platforms are supported.</td>
<td></td>
</tr>
<tr>
<td>Offers better performance for workstations with slower CPUs.</td>
<td></td>
</tr>
</tbody>
</table>

**The Selected Solution**

I decided to use the web based server-side application solution. The inability to access Prolog through an applet really ruled out the applet solution. In the desktop application solution although the reduced server overhead of having Prolog run on the client-side is desirable it is outweighed by the need to refresh all forms whenever the workflow changes. One user changing the workflow would require the application to inform all the other users that they need to regenerate the static forms or every application needs to constantly check for updates to the database. Also the ease with which the web based solution can deal with form generation is highly desirable since the solution is primarily an intensive form based system. Substantial time in the project would have to be spent implementing either a simple browser or a complicated static form generation process for the Desktop application solution. With the server-side solution these problems are completely bypassed with the ability to use any number of browsers already available and dynamic deliverance of forms rather than static. Another important point is that the web based solution only requires a single installation of Prolog on the server. Having to install Prolog on every client machine
is undesirable in terms of license costs and the time to install the software on every machine.

### 3.1.2 Which Server-Side Language to use?

In using a web base application a plethora of server-side languages are available.

*The considered solutions*

1. PHP – Hypertext Pre-processor
2. JSP – Java server pages
3. ASP – Active server pages
4. Perl

*The Selected Solution*

I decided to use PHP primary due it being more mature in terms of open source tools and libraries than any of the other server-side languages. The PEAR [22] repository provides a vast resource of PHP libraries. Tools such as Simple Test [34] are available which provides a unit testing framework. Although JSP has stronger object support, its performance and maturity as a server-side technology is considerably weaker than PHP. Also as of PHP 5 strengthening object orientated support is available.

### 3.1.3 Which Client-Side Scripting Language to use?

A client-side scripting language will be required in order to allow validating of the users actions in forms.

*The considered solutions*

1. JavaScript
2. VBScript

*The Selected Solution*

I decided to use JavaScript. Really either language could have been used since they both provide very similar functionality. JavaScript was chosen primary because I am more familiar with it than VBscript.

### 3.1.4 Which Planner to use?

A planner is required to produce plans from the Event Calculus workflow specifications. A number of possibilities were investigated and experimented with. Since it is likely that the planner will need to be changed it is very important that the code is well documented and there are papers available documenting design.
3. SYSTEM OVERVIEW

The considered solutions

1. Oliver Ray [25] and Tony Kakas ALP program

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well documented code with research papers available which detail the design.</td>
<td>In its current form it does not support Event Calculus axioms although it could be altered to add this support.</td>
</tr>
</tbody>
</table>

2. Murray Shanahan’s planners
   i. Research paper version [28] (Version 4.2)

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The planner as documented in his research paper.</td>
<td>Uncommented.</td>
</tr>
<tr>
<td></td>
<td>Inefficient.</td>
</tr>
<tr>
<td></td>
<td>Not effective for hierarchical planning.</td>
</tr>
</tbody>
</table>

   ii. Optimised version (Version 1.9a)

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particularly fast implementation.</td>
<td>Not effective for hierarchical planning.</td>
</tr>
<tr>
<td>Good with partial-order planning.</td>
<td></td>
</tr>
</tbody>
</table>

   iii. Special Hierarchical planner version (Version 1.15)

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special code for hierarchical decomposition.</td>
<td>Not as efficient at handling partial-ordered planning as Version 1.9a.</td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to use Shanahan’s planner, specifically version 1.15. This planner already supports reasonably complex Event Calculus plans unlike the ALP program. In addition, the code has highly detailed commenting and support for hierarchical planning. This will keep open the choice of using compound actions when deciding on how the Event Calculus specification will be implemented. Since only total ordered plans will be used the inefficiency at handling partial-ordered plans is not relevant.

3.1.5 How to Deal With Uncertainty in Planning

In workflow different paths are possible in achieving a goal. As we are generating the plan to govern the actual process the planner has to deal with uncertainty. It does not know which of the plan paths will be taken. Within the Event Calculus and planning we need to decide how to deal with this uncertainty.
3. SYSTEM OVERVIEW

The Considered Solutions

The considered solutions are all detailed in Section 2.3.

1. Conditional planning

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fits naturally with Prolog’s ability to backtrack and find new solutions.</td>
<td>The number of plans can grow rapidly.</td>
</tr>
<tr>
<td>Allows simple workflow specifications, leaving the planner to do the work of generating and elaborating the different ways of solving the plan.</td>
<td></td>
</tr>
</tbody>
</table>

2. Re-planning

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoids the need to generate all possible plans, which leads to a faster plan generation process.</td>
<td>The re-planning would require planning to be performed throughout the progress of workflow through the iWFMS. This would reduce the overall performance of the iWFMS.</td>
</tr>
</tbody>
</table>

3. Probabilistic planning

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The plan generation overhead is reduced since not all the plans are generated.</td>
<td>Within workflow it is very hard to predicate with any accuracy the probability of a path. The probability swings with time.</td>
</tr>
<tr>
<td></td>
<td>Complexity in dealing with situations that are unplanned for if the user takes the improbable route.</td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to use conditional planning to represent the different decisions that can be taken in workflow. The primary reason for adopting conditional planning was that it fits with exactly what we want in workflow. We need to know every single path the user could follow. This does have the consequence of complex models generating a very large number of plans and taking considerable time to produce in planning. This will be overcome by introducing the option to use caching. The user invokes the planner to generate plans representing the different ways workflow can be undertaken. This information is cached in a database. Then when other people invoke the planner to generate plans instead of repeating the planning process the cached version of the workflow is used.
3. SYSTEM OVERVIEW

3.1.6 Which Logic Programming Language to use?

There are a number of compilers and development environments available for writing Prolog.

The Considered Solutions

1. Sicstus Prolog
2. Q-Prolog
3. SWI-Prolog
4. LPA MacProlog 32

The Selected Solution

I decided to use Sicstus Prolog. This was primarily because Sicstus provides libraries and features that are more advanced than the other versions. Also that the debugging features of Sicstus Prolog are extremely powerful. This will be vital when changing and developing Murray Shanahan’s Event Calculus planner. The downside to using Sicstus is that Murray Shanahan’s Event Calculus planner will have to be converted to Sicstus Prolog.

3.2 The System
3. SYSTEM OVERVIEW

Figure 12: Overview of the iWFMS architecture and dataflow
The server-side domain focuses on the processing in PHP. The logic programming domain focuses on all the processing performed in Prolog. The iWFMS can be regarded as containing the server-side domain and the logic programming domain. More specifically the iWFMS is the overall interface provided for the user to interact with and all the backend processing seen in the logic and server-side domain. The iWFMS interface is used to present the HTML forms and run the workflow plans for the user.

Control flows into the iWFMS when the client web browser makes a request to the server. The data that a user passes to the server is required by both the server-side domain and the logic programming domain. This client’s posted data represents the data that is posted when a user submits a form. The logic programming domain and server-side domain have to interact with each other while processing a request from the client web browser.

The system architecture maps to the workflow management coalitions workflow reference model highlighted in Figure 10. The process definitions are the Event Calculus workflow specifications. The enactment service is provided through the server-side domain executing the plans. The workflow client application is the user’s web browser. The iWFMS has the new component not seen in the reference model of the planner.

### 3.3 The Main Components

#### 3.3.1 Server-Side Domain

The server-side domain consists of the follow main components:

- **Prolog Interpreter**
  This process converts Prolog plans produced in the logic programming domain into directed graphs. These are then used to govern the running of the iWFMS and its interaction with the user.

- **PHP Planner**
  This process deals with the execution of the workflow by following the directed graphs. It involves generating HTML forms and JavaScript for the client based on the direct graph plans. Also checking that data posted from the Client HTML form is valid according to the plan. The PHP planner uses the workflow to constraint if the user can move to a new HTML form, based on the data they posted to the server. Finally the planner deals with saving the information posted in the HTML form to a database dictated by XML specifications.

- **DTD to Prolog HTML rules**
  This process deals with converting W3C HTML DTD specifications into type rules about HTML which can be used by the logic programming domain to ensure
that the user has the correct form element specifications in the workflow specifications.

3.3.2 Logic Programming Domain

The logic programming domain consists of the following main components:

- **Prolog CGI**
  This process can be regarded as the manager for the rest of the logic programming domain. All interactions with any of the elements go through Prolog CGI. Its focus is dealing with the generation of HTML form elements and handling the interaction with the Event Calculus planner. The data posted from the client web browser has to be converted to a format that Prolog can understand.

- **Workflow specifications**
  These are the Event Calculus formalisations used to represent workflow. They model the different stages and constraints required for HTML form systems. The workflow specifications are taken by the Event Calculus planner and turned into plans.

- **Event Calculus Planner**
  This process is the new version of Murray Shanahan’s Event Calculus planner. It takes the Event Calculus workflow specifications and generates every possible
plan. The planner also interacts with a knowledge base to find out about the user invoking the planner.

- Type Checker
  This process links into the Event Calculus planner. It ensures that the HTML form elements used within the workflow specifications are valid. This prevents any unexpected behaviour when the plans generated are used to generate HTML forms in the iWFMS.

![Logic programming Domain Dataflow](image)

**Figure 14: Logic programming Domain Dataflow**

### 3.4 Terminology

All references to predicates in the Event Calculus will be in Prolog Sicstus format.

`before(t1,t2)` will be used to represent the temporal orderings

`holds_at` will be used to represent the HoldsAt predicate.

All workflow specifications examples will abstract from the Meta language they are contained in to increase clarity.
4 Logic programming Domain

4.1 Overview

In this section we look at the use of Prolog to provide logic programming. The main components examined consist of:

- Workflow specification
  Modelling in the Event Calculus the constructs used to represent workflow.

- Planning
  Improvement and adaptation of Murray Shanahan’s ASLDICN planner written in LPA MacProlog 32. Adding the additional functionality to handle conditional plan generation.

- Prolog Database integration
  How the knowledge base is interacted with from within Prolog.

- Typing
  Checking form element specifications within Event Calculus workflow specifications have the correct type. Dynamically generating type rules based on HTML DTD specifications from the W3C.

4.2 Event Calculus Workflow Formalisation

The Event Calculus workflow formalisation is used by the modified Prolog Event Calculus planner to generate plans. The workflow has to be expressive enough to support the functionality of the hospital model discussed in Section 2.8. This requires the following criteria:

- Multiple paths in the workflow
  The specification needs to be able to specify that there are a number of different ways of progressing from a workflow stage. Depending on the progression used the workflow path may be different to other possible progression paths. For example in the hospital specification when checking if a drug prescription is correct there are two possible progressions, when the drugs are correct or when they are incorrect.

- Specification of form contents
  Through the specification, HTML forms used in the system need to be specified. They need to be in the plan as they are a base for constraints throughout the plan.

- JavaScript constraints on the workflow
  Controlling the order in which the elements of a form may have to be entered in and the values that they can take.
4. LOGIC PROGRAMMING DOMAIN

- Group/role management incorporated in the planning
  Generating different HTML forms for different group members and restricting which members can access specific forms.

- Looping
  In the hospital model we see that loops are required. For example when the controlled drugs test fails the assessing of the patient has to be repeated and so on until the controlled drugs test succeeds.

4.2.1 Level of Abstraction

The Problem

Deciding upon the level of abstraction to use in the Event Calculus workflow specifications.

The considered solutions

1. Low level scripting workflow
   Producing all the workflow at the same level of abstraction.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher performance in the planning process since multiple levels of abstraction do not have to be unravelled.</td>
<td>The lack of abstraction will make designing of workflow harder as it will be more complex to reason about.</td>
</tr>
</tbody>
</table>

2. Hierarchical planning providing abstraction.
   Hierarchical planning is discussed in Section 2.3. This can be achieved in the Event Calculus through using compound actions. Utilising such a mechanism allows any level of abstraction to be realised.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows the reuse of workflow. Compound actions could be considered as procedures which can be called through the happening of a compound action in the plan. The compound action can be used many times removing the need to have to explicitly duplicate the actions contained within the compound action.</td>
<td>There is a performance cost due to the need to unravel the abstraction levels.</td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to adopt the Hierarchical planning solution but limit the levels of abstraction to four levels of compound actions. One level would be a single compound action, two levels would be were the compound action has a compound action within it and...
so on. This reduces the performance issues while still providing the benefits of abstraction. The deciding factor in using hierarchical planning is the ability to reuse workflow. This is highly desirable as often in a WFMS there are a small core group of workflow tasks which are repeated frequently.

It would seem that we have to use a control flow based representation since we are using the Event Calculus which has an explicit representation of time. However, control flow workflow is associated with low levels of abstraction which we have already decided against using. Therefore, we have used control flow but through hierarchical planning we have borrowed some of the ideals of dataflow based workflow. We have decided to call this Hierarchical Control flow.

**The Design**

I decided to adopt the analogy of direct graphs for the specification of the highest level of the workflow. This follows very naturally with workflow as the nodes can represent states and edges the transition to new states. By providing a concept above the level of the form system, workflow can be reasoned about without having to consider what is going on inside the forms.

- **Nodes - Workflow unit**
  A workflow unit can be visualised as a black box which contains all specifications required for dealing with a HTML form. For example the workflow unit SignInPatient may be broken down into the entry of a form, creating the form, submission of the form and processing of the data posted to the server by the form. A workflow unit is a compound action that would appear in the high level plan before hierarchical planning refines it to lower abstraction levels.

- **Edges - The flow of control between the different nodes.**
  Edges can be seen as the transition to a new workflow unit. At this level there are no conditions on these edges, they always happen. They also represent the dependencies between the workflow units. An edge indicates the edge source workflow unit must occur before the edge destination workflow unit. Edges are not actions that appear in the high level plan but constraints using hold_at predicates which ensure the workflow units are added to the plan.

**The Implementation**

The levels of abstraction were implemented in Prolog within a Meta language. The Meta language format is required by the Event Calculus planner seen in Section 4.3. The different levels of abstraction where achieved through using compound actions in the Event Calculus. A workflow unit action representing a compound action would be introduced into the plan. Upon refinement the compound actions added to the plan would be refined to represent the actions of the compound action and any pre-conditions associated with it.
4. LOGIC PROGRAMMING DOMAIN

4.2.2 Specification of the Low Level Plan

The Problem

How the high level nodes and edges relate to the low level representations. Ideally for ease of transition between the abstraction levels there should be a link between the nodes and edges of the directed graph and the lower level of abstraction. Ease of transition will help make it easier to reason about, design and implement the workflow. The problem is also ensuring that the HTML form process is represented in the workflow to enable the iWFMS to guide the systems running based on the workflow plan.

The Considered Solutions

In designing the workflow formalisation I considered a number of possible solutions. This was a progressive process with prototypes produced for the different formalisations to help decide if they were suitable. Rather than giving detailed explanation of each version I have highlighted the main design decisions, design changes and the reasoning involved.

Version 1

• High level edges between workflow units mapping to form submission

In describing the links between the different workflow units I decided that the form submission represented the transition to a new workflow unit. This was chosen since after the user has submitted the form they have ended their interaction with the form and thus the workflow unit. This models the real world interaction in terms of the user and the system.

Version 2

• High level edges between workflow units mapping to server validation

I discovered that we do not really want to exactly replicate the real world user HTML form interaction model due to the nature of submission. Users can submit a form and there can be no progression of workflow. The submission can be the cause of initiating the required conditions for progression or it may fail to meet those conditions. Irrelevant of meeting the conditions for workflow progress submission can occur.

Whether form elements have not had any data inputted can be detected by JavaScript. If this was the only constraint on the progression of workflow, submission could be prevented thus ensuring it only occurred when the required conditions were met to progress to a new workflow unit. The problem lies with the fact that the decision about whether to progress to a new workflow unit is sometimes taken on the server-side. For example, certain database lookups may be required which can only be provided at the server. Thus this highlights why submission cannot represent the transition to a new workflow item. Instead the transition between workflow units is the acceptance or rejection of the conditions at the server. We shall refer to this check at the server as the Edge progression check.
4. LOGIC PROGRAMMING DOMAIN

- Representing the failed server edge progressions check explicitly.
Failure of the edge progression check was represented through recursion. The
failure path caused the workflow unit to be repeated until the edge progression
check transition was made successfully.

Version 3
- Implicit failure of edge progression check
Recursion caused problems in generating multiple plan solutions. The recursive
clause could loop infinitely continuously producing new plans due to failing the
edge progression check. Due to this the recursive failure clause was removed and
the failure of the edge progression check was represented implicitly. The iWFMS
execution model would backtrack in the plan if the edge progression check failed.

Version 4
- Representing planning at the high level and the low levels of abstraction
Planning at the highest level of abstraction involved starting at the directed graph
abstraction level and working towards the executable plan. The goal of planning is
to try and reach the goal Node, the end point where all that was required to have
been undertaken is achieved. All high level plans eventual reach the goal Node.
Planning at the low level is where we start with executable actions (no compound
actions). Planning at the low level was added to provide the facility to re-plan
while the iWFMS was executing the plan. This requires planning at a low level
since the planner only has a knowledge of the past actions from following a low
level plan. Therefore it only knows its low level actions and not the high level
actions since they are never in the low level plan.

Version 5
- High level planning only
There were serious performance issues in the plan generation so the low level
planning was removed.

When planning from the high level compound actions a plan is produced with low
level non compound actions. Then all the low level plan constraints required to
indicate the dependencies between the low level actions also have to be evaluated.
This happens despite the fact that the plan is already completely generated. In high
level planning the first thing that is undertaken is the introduction of all the
compound actions thus indicating the ordering and dependencies between them.

These resolve to the lower level non compound actions through hierarchical
planning refinement. There is no need to indicate dependencies between the low
level actions since these are already in place from the high level compound
actions. The low level constraints are required in low level planning since the
compound actions are never used so none of the dependencies are in place. These
constraints are what ensure the correct actions are used. Therefore, the problem
was that the high level and low level could not be performed separately; high level
planning has to do the redundant low level planning. This introduced a
performance overhead for high level planning primary due to the redundant
planning repetition but also due to the extra time points introduced by the extra
holds_at predicates needed to enable low level planning. More time points meant
that the number of comparisons required with each addition of a before predicate
increased. To emphasize the difference this made version 4 took over 10 minutes to generate a complex hospital model while version 5 with the same model can produce the same solutions in less than 30 seconds.

**The Selected Solution**

Version 5 was my final solution based on the progression from versions 1 to 4.

**The Design**

The workflow unit consists of a number of elements.

![Workflow unit composition](image)

A node is a workflow unit. It is important to point out that within the workflow unit are both the workflow edge and the workflow node. The edges at the directed graph level are not actions in the plan. Hence the action that represents edge progression can only appear in the compound action of the workflow unit.

- **Form entry (Mandatory)**
  The event of a user selecting a form via clicking on a HTML link on the core index page of the iWFMS. This event is required to enable permission checking on whether the person is allowed to enter the form.

- **Database fetching (Optional)**
  The action of fetching information from the database and displaying it to the user. This is required since within forms it is sometimes necessary for the user to be able to see what data has been inputted into some form at a past stage. For example in the hospital model the pharmacist needs to be able to see the drugs prescribed by the doctor which were previously entered in order to check if they are correct.

- **Form Creation (Mandatory)**
  The event of the form being created. Consisting of each of the form elements required in this form. This is required to guide the construction of the form.

- **Form input (Optional)**
The event of the user entering information into the form elements. This is required to try and capture JavaScript constraints on the inputting of form data. For example ensuring the input is a number.

- Form Submission (Mandatory)
The event of the user selecting the submission button in the form. This segments the client preparation plan actions and the server condition planning actions. Client-side planning regards the preparation and generation of HTML/JavaScript, server condition planning being the checking of edge progressions.

- Edge progression (server-side check transition) (Mandatory)
Representing the progression test that is used to check whether the conditions for progression to the next workflow unit have been achieved. This is required to enable the iWFMS plan execution system to compare the actual data posted in the form and the constraints.

**The Implementation**

The workflow units were implemented in Prolog within the Meta language for the hospital model. The following workflow units were implemented:

- **patientAdmission** – The user is admitted to the hospital
- **assessPatient** – The patient is assessed by a doctor and prescribed drugs
- **drugAuthorise** – The pharmacy checks the prescription
- **drugSucess** – The pharmacy gives an expected pickup time
- **drugFailure** – The doctor checks the pharmacy’s correction of his/her prescription
- **pharmacy** – The pharmacy indicates the drugs are ready to collect.
- **controlledDrugAuthorise** – The nurse checks whether the controlled drugs are correct

**4.2.2.1 Form Entry**

**The Problem**

How form entry is specified within the workflow unit. The action needs to specify security restrictions on who can access a form.

**The Design**

```
formEntry(FormName, Group)
```

- **FormName**
  Indicates the form that the `formentry` applies to.

- **Group**
  Indicates the group the user must be a member of in order to gain form Entry
4. LOGIC PROGRAMMING DOMAIN

4.2.2.2 Database Fetching

The Problem

How to represent the database fetching in the workflow unit. It is necessary to avoid any details of the database within the actual Event Calculus workflow specification. This prevents having to have the details of database names, fields and tables which could change frequently, from being in the Event Calculus workflow specification. It also has been the goal through the workflow specification design to ensure everything is at the form level rather than at the database level. This allows the users to reason about something tangible in the system rather than the backend, which often remains hidden.

The Design

databaseFetch(XmlFileName)

- XmlFileName
  Indicates the XML file to use in order to identify what data to display in the form. The use of an XML file means we can abstract from the database within the plan.

The Implementation

The databaseFetch action was implemented in Prolog within the Meta language. It was used in a number of places in the hospital workflow model. An example being where the pharmacist checks that the doctor’s prescribed drugs for a patient are correct.

4.2.2.3 Form Creation

The Problem

How form creation is specified within the workflow unit. The action needs to allow maximum flexibility in the form elements’ specifications. It also needs to contain all the information required to generate the form element when the plan is being executed.

The Design

The formEntry action was added to the Event Calculus workflow specification of the hospital model. How the groups field of the predicate is used to indicate which groups can access a form is given in detail in Section 4.2.3.2.
createFormElement( FormElement,"Display Name", ActualName, Value, [Attributes] )

- **FormElement**
  Indicates the type of form element to create. The possible elements supported are:
  - Radio buttons
  - Text inputs
  - Textareas
  - CheckBoxes
  - SelectBoxes

- **"Display Name"**
The name that is shown when the HTML form element is displayed to the user.

- **ActualName**
The name of the form element. This is not shown to the user but is used when the form is posted. The variable that will contain the data of a specific form element will take this name.

- **Value**
  This has two meanings depending on which type of form element is used.
  - For Radio buttons, Checkboxes and Select boxes value represents the content that is posted to the server on submission if the relevant form element has been selected. This is never seen by the user.
  - For all other form elements value represents the content that the form element starts with. This is seen by the user.

- **Attributes**
  This represents the list of attributes that are to be applied to this form element. This defines visual and behavioural features of the form element.

*The Implementation*

The `createFormElement` action was implemented in Prolog within the Meta language. All the form creation actions were grouped into compound actions, each representing all the form elements for a particular form.

**4.2.2.4 Form Input**

*The Problem*

How the user inputting form data is represented within the workflow unit.

*The Design*

happens(entry(FormName, FormElementName, FunctionOperators, Value), T)

- **FormName**
4. LOGIC PROGRAMMING DOMAIN

The form this *entry* action is associated with.

- **FormElementName**
  The name of the form element that this *entry* action happens at.

- **FunctionOperators**
  Indicates how the Value is going to be matched against the actual value of the form element. The functionOperator mode specified may not use the value.

  - **FunctionOperators supported**
    - **Presence**
      The form element’s value is not empty
    - **Match**
      The form element’s value matches with the specified value.
    - **RelativeExpression**
      The Value represents a relative expression that must be matched with the form element Value
    - **Function**
      The Value represents a function that must hold with the value of the form element.

- **Value**
  Represents the value used in the FunctionOperators test. The value can be a function such as ‘greaterThan(X)’, a string or an integer.

*The Implementation*

The *entry* action was implemented in Prolog within the Meta language. Within the hospital model this action was used a number of times to provide JavaScript constraining. Since there could be many *entry* actions these where contained in a compound action. Full details of how the *entry* action was used to achieve JavaScript constraining are given in Section 4.2.3.3.

### 4.2.2.5 Form Submission

*The Problem*

How to represent the form submission action within the workflow unit.

*The Design*

formSubmission(FormName)

- **FormName**
Indicates the name of the form that is being submitted.

The Implementation

The formSubmission action was implemented in Prolog within the Meta language. The submission action was used in every single workflow unit in the hospital model.

4.2.2.6 Edge Progression

The Problem

How to represent the edge progression action within the workflow edge. The action needs to contain all the information in order to test the relevant form element after a form post.

The Design

edgeProgression(FormName, formElement( FormElementName, FunctionOperators, Value )

- FormName
  Indicates the form that the edge progression is happening at. This does not serve any purpose in the execution of the plan. It is used to make it easier for the modeller to associate progressions to forms.

- FormElementName
  The name of the form element that the edge progression test is going to be performed on

The FunctionOpertors and the Value have exactly the same format as seen in the form input FunctionOperators and Values. The only difference is that the Value in the server-side test can be a reference to a XML file which specifies the location of a single database table field. The value is fetched and used in the value test.

The Implementation

The edgeProgression action was implemented in Prolog within the Meta language. Exactly how edgeProgression was used to achieve multiple plan paths is detailed in Section 4.2.3.4.

4.2.3 Advanced Plan Functionality

With the overall design in place there needs to be additional functionality to achieve the requirements of the hospital workflow model. In designing the workflow representation I created many prototypes which looked at specific features. These primarily consisted of:
4. LOGIC PROGRAMMING DOMAIN

- Group permission
- JavaScript planning
- Iteration
- Knowledge base interaction

4.2.3.1 Restricting Workflow Generation Based on Groups

The Problem

Within the workflow there needs to be a way of specifying which groups can create certain workflow elements. This can be used to customise the workflow that is created based on the user that is invoking the planner. The reason we need such a feature is that the workflow process may be different depending on the creator. For example if a super user invoked the creation of workflow plans you may want it to feature an extra workflow stage that includes super user authorisation. Alternatively, perhaps if a nurse invokes the workflow creation certain workflow units can be skipped out.

The solution needs to be flexible enough to be placed at any point in the workflow specification. This means that the workflow can be tailored to handle different Entry compound actions, form creation compound actions, workflow nodes, edge progressions, directed graph nodes and directed graph edges based on the user invoking workflow generation.

The Design

An authenticatedForGroup predicate can be placed at any point in a workflow specification.

authenticatedForGroup([Groups])

The predicate takes a list of groups of which the user creating the workflow has to be at least a member of one of them in order to be able to use the relevant workflow element. The predicate is resolved in the planner based on an axiom which finds the user creating the workflow and then searches the knowledge base looking up the groups that the user is a member of.

authenticatedForGroup([Groups]):- user(User), knowledgeBase(User,[Groups]).

Figure 16: Authenticated for group axiom

If a group matches then the planner continues on that particular plan path. If the user is not a member of the required groups the authenticatedForGroup predicate fails when resolved causing the planner to backtrack and try to find another solution. If the user is not authenticated for any of the workflow paths then no plans are generated. The following figure illustrates this solution:
We see in the figure that if the user invoking the workflow generation is a member of the nurse or staff nurse group then they are expected to assess the patient entering their nurse ID number. If a doctor creates the workflow they are expected to assess the patient entering their doctor ID.

The Implementation

The design was implemented in Prolog. One example plan based on the hospital model was implemented to reflect the dynamic generation of the workflow based on the user invoking the planner. The example was where different initial forms for accessing the patient were used based on if the user creating the workflow was a staff nurse or a receptionist.

4.2.3.2 Form Entry Group Permissions

The Problem

Within the workflow there needs to be a way of specifying which groups can access a form. For example in the hospital model we would only want users who were a member of the doctors group to be able to access the prescribe drugs form.

The Design

The solution to the problem was to introduce group axioms into the workflow specification.

Within the compound action that represents a workflow node it is possible to have a precondition which finds a group from the group axioms. A variable can be bound representing a group and introduced into a `happens` predicate.

```prolog
// Figure 18: Binding Groups in Prolog

groupLevel(0, GROUP),
happens(formEntry(formName, GROUP), T).
```

So the final plan will now have a reference to which group can access a form.
Groups are not reference through name but through a security level. A number of groups are assigned to each security level. The highest level of security starts at Level 0. This is used to represent the super user status. From 1 upwards each level represents a lower level of security. The advantage of such a solution is that there is a level of abstraction between groups and the plans meaning that a group name can change without requiring a change in the plan.

Multiple groups are handled through conditional planning. If there are more than one group assigned to a security level then the planner when backtracking will find another plan which indicates a different way of solving the Group axiom. In other words other groups not already chosen that are at that same security level would be selected.

The Implementation

The design was implemented in Prolog. It was converted to the Meta language required by the Event Calculus planner. Five different groups excluding the super user were implemented indicating the different roles seen in the Hospital model:

- Receptionist
- StaffNurse
- Doctor
- Pharmacist
- Nurse

4.2.3.3 Form Input Entry Constraints (JavaScript)

The Problem

Within a HTML form system JavaScript is often used to constrain how the user fills in a form. Usually this takes the form of either preventing submission of a form due to invalid fields or disabling form elements until other form elements have been completed. Within our workflow model we want to be able to adopt modelling of the user inputting form data. This will enable support of JavaScript based constraining. This part of the workflow specification is not mandatory in all workflow units and does not have to consist of all form elements in a form.

The Design

The entry action is used to achieve two levels of form input constraining.

1. Form element value constraining

The entry action provides a way of specifying constraints on the values entered in a form element. The ordering of entry actions indicates dependences between the past entry actions. For example:
Since approval occurs after drugsCorrection, it is dependent on drugsCorrection passing the presence test. In this way a sequence of constraints on the values entered into a form element can be built up indicating a way of entering data into the form.

Through dynamically binding the value of the entry action we can reflect different ways of passing the form value tests. The planner would backtrack generating plans representing each of the different ways of binding the value.

How these values for the entry action are bound relates to domain specific constraints. For an example we look at the entry actions for a form relating to the hospital admissions process.

```prolog
wardInformation(childrensWard, Age) :-
  age( child, Age ).
wardInformation(adultWard, Age) :-
  age( adult, Age ).
age( child, lessThan(18) ).
age( adult, greaterThanOrEqual(18) ).
```

Figure 21: Hospital specific constraints

Here the workflow specific wardInformation predicate is resolved against predicates that represent facts in the hospital model. This generates two different ways of binding the entry actions, one when a child’s age is entered indicating that the ward must be a children’s ward and another where an adults age is entered indicating an adult ward must be entered.
2. Form input ordering

Since *entry* actions in the Event Calculus have a temporal ordering we can use this ordering to imply the order that the user enters the form elements. This is dependent on the formElementName provided by the *entry* action but can disregard the other predicate values.

The user may not always want this constraint but is instead using the *entry* actions to reflect form element value constraining. Due to this the *entry* actions to reflect input ordering have to be activated. This is achieved by adding disabled attributes to all the createFormElement predicates corresponding to the form elements mentioned in the *entry* action except the first action. The disabled attribute means that the form element cannot be used. The first element is not disabled since the user needs a point to start inputting at. If it was disabled the user would never be able to start the form entering!

To following figure helps illustrate this solution:

We can take this constraint as saying that the roadName form element cannot be entered until the houseNumber element has been filled in.

To activate this constraint the road element is set as disabled but houseNumber is not.

The client-side JavaScript based planning engine (seen in Section 5.5) would use the *entry* plan to activate the form elements as the user meets the conditions of having filled in previous form elements. Not setting the form elements as disabled results in the planner activating form elements that are already active which thus has no effect.

**The Implementation**

The form input *entry* actions where implemented as part of the hospital model. These were written in Prolog in the format of the Meta interpreter.
4.2.3.4 Conditional Edge Progression Actions

The Problem

We have already seen conditional actions and how they are used in *entry* actions to generate different ways of the user entering the form data. We also need to support conditional actions reflecting the different ways that edge progression can occur leading to new workflow units. For example in checking whether drugs prescribed to a patient are correct or not the edge progression could either be where the drugs checked are ok or when they are not.

The Design

The design uses the idea of providing multiple *edgeProgression* actions. The following figure is an example from the hospital model that will help illustrate the solution.

```prolog
happens(workflow(workflowEdge,User), T1,T3):-
    happens(workflowNode(drugAuthorise,User),T1,T2),
    before(T1,T2),
    happens(workflowEdge(drugAuthorise,User),T3),
    before(T2,T3).
```

**Figure 24: The drug authorise Workflow unit**

Here the action we are concerned with is *workflowEdge*. This is the compound action call that reduces to the different ways the workflow will progress to a new workflow unit.

By providing two different compound actions for *workflowEdge* we can indicate that there are two different ways of achieving progression.

```prolog
happens(workflowEdge(drugAuthorise,User),T1, T1):-
    holds_at( validated(form(checkdrugs, 
                        formElement( correctDrugs, match, false)) ) , T1 ).

happens(workflowEdge(drugAuthorise,User),T1, T1):-
    holds_at( validated(form(checkdrugs, 
                        formElement( correctDrugs, match, true)) ) , T1 ).
```

**Figure 25: Multiple workflow edges**

The *holds at*’s fluent in both cases is initiated by the *edgeProgression* action.

This provides the edges but does not enforce linking to the next workflow unit. For example it is possible for the false correctDrugs edge progression to occur and for the drug success workflow unit to follow when this should only occur when the edge progression is false. This was overcome by adding preconditions to the success drugs and failed drugs workflow units using the *occurred* predicate. The *occurred*
predicates design and structure are detailed in Section 4.3.3. This checks that the workflow units are only chosen if the relevant action (true or false edgeProgression) has occurred in the past.

Having multiple workflowEdge actions causes an interesting problem. Within the workflow unit seen in Figure 24 we see that a single time point is used for the happens predicate of the workflowEdge. The problem being the compound action for one workflowEdge may have multiple edge progression constraints while the other option may only contain a single edge progression constraint. The first requires a single time point while the other since it has multiple actions requires a start time point and an end time point.

In Figure 24 it is not possible for the workflowEdge happens predicate to bind with both conditions, it either has to use one time point and match with Figure 26 or use two time points and match with Figure 27. i.e.

happens(workflowEdge(drugAuthorise, User), T3),

or

happens(workflowEdge(drugAuthorise, User), T3, T4),

Not being able to match both workflowEdges means that we would have to produce two workflow units one for the single case and one for the start and stop time point. This loses the elegance of leaving the work to the planner. The user has to explicitly indicate the two different paths. The problem was overcome by chaining edge progression constraints. The edgeProgression action that initiates the holds_at’s fluent is constrained by another holds_at condition which is initiated by another edgeProgression action. And so on for as many edgeProgression constraints as required.
The Implementation

This solution was implemented in Prolog within the Meta language for the hospital model.

4.2.3.5 Plan Iteration

The Problem

The workflow needs to represent the concept of loops. In our hospital model we see a doctor checking a pharmacist's drug prescription corrections. If the prescription corrections are wrong then we need to loop back to the workflow unit where the doctor made the prescription. This sequence of workflow units would loop until the doctor accepted the pharmacists correction or no corrections where given by the pharmacist. Looping needs to support repetition of a sequence of workflow units rather than an individual unit. This is a difficult problem in Event Calculus planning.

The looping plan on its own is problematic since it never achieves the goal node. In the hospital model there are multiple plans and each represents the different paths to the goal. The failure plan does not contain the option for success just failure since the success case is dealt with in another plan.
Therefore the workflow cannot represent an iterative plan on its own as it never achieves the goal.

The iWFMS uses the generated plans to guide execution. The user in reality can loop as many times are they wish. While the planner has to have a fixed limit on the number of times it evaluates a loop in order to prevent looping infinitely. Due to this some processing above the plan level will always be required since the planner can never truly represent what the user could do.

**The Considered Solutions**

1. **Looping is implicit in the plan**
   The planner does not process the loop but instead is resolved outside the planner. The planner contains an action which identifies the point to loop to but the action when encountered by the planner is simply added to plan rather than trying to resolve the loop. Then either through Prolog or the iWFMS the loop point can be removed and replaced with the actual loop. We will refer to this process as loop resolving.

   **Advantages**
   - The overhead of detecting loops and ensuring they do not occur infinitely in plan generation is not necessary.

   **Disadvantages**
   - A less natural representation in the workflow specifications.

   **Prescribe drugs**
   **Check drugs**
   **Failure**
   **Prescribe drugs**
   **Check drugs**
   **Success**
   **Goal Node**

   **Figure 29: High level workflow plans**

2. **Recursive looping explicit in the plan.**
   Handling the looping through recursion in the compound actions. The loop is represented as a compound action which has a reference to itself in the actions that make up the compound action. A base case compound action is given which does not have this loop. The recursive case would contain the relevant workflow units that have to be repeated. The planner would be limited to only using the recursive case once to prevent infinite looping.

   **Advantages**
   - The plan generated by the Event Calculus planner requires a degree of processing before it can be used for execution.

   **Disadvantages**
   - A less natural representation in the workflow specifications.
4. LOGIC PROGRAMMING DOMAIN

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leads to a more natural plan specification.</td>
<td>There is an overhead of detecting loops and ensuring they do not occur infinitely.</td>
</tr>
<tr>
<td></td>
<td>The plan generated by the Event Calculus planner requires a degree of processing before it can be used for execution.</td>
</tr>
</tbody>
</table>

3. One stage Looping is explicit in the plan
Since the recursion is going to be limited to a single loop the failure path could be mapped as part of the workflow specification at the directed graph level rather than using recursion.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>No loop detection overhead but the plan generated is exactly the same as solution 2.</td>
<td>Slightly verbose solution resulting in more complex workflow specifications which may make them harder to reason about and develop.</td>
</tr>
<tr>
<td>Simple to implement in the plan as it is nothing new to the already established workflow design.</td>
<td>In complex plans the path of the iteration at the directed graph abstraction level may be complex.</td>
</tr>
<tr>
<td></td>
<td>The plan generated by the Event Calculus planner requires a degree of processing before it can be used for execution.</td>
</tr>
</tbody>
</table>

The Selected Solution

I initially decided to adopt the one stage looping is explicit in the plan solution. The reasoning behind selecting the one stage looping solution over the recursive solution lies with the priorities of the iWFMS.

The chosen solution achieves exactly the same plan as the recursive solution. The difference between the two solutions comes down to the balance of workflow specification simplicity over performance. Since the workflow specification is visualised as being managed through a java based application the extra specification detail can be handled through this application. On the other hand the performance of the planner is vital in the iWFMS since the user has to wait for the plan generation. Hence this real-time constraint takes priority over the workflow specification simplicity and thus was why the solution was selected.

After analysing the problem and trying to develop a design I discovered that the solution selected and solution 2 are in fact impossible. The problem lies with the processing required to truly represent the loop. Both solution 2 and 3 produce the same result so they both require the same processing.

In the plans generated by solution 2 and 3 the loop is only represented as a singular loop and not explicitly as a loop (it only happens once). The iWFMS plan execution process would follow the plan only allowing a single loop while the user in reality may loop many times. Therefore a degree of processing is required to detect any repetition of actions and reduce to an explicit loop.
Note as already mentioned, Figure 31 is impossible to represent as a plan (as seen in Figure 29: High level workflow plans).

There is a fundamental problem with this solution. This is best illustrated with a simple workflow example:

Figure 32 is a plan which would be generated by the Event Calculus planner using the two solutions discussed for handling loops. It has a single loop case presented which is the action $c$. The first $c$ action fails and is repeated where it succeeds and stops looping.
4. LOGIC PROGRAMMING DOMAIN

Figure 33 is the plan in Figure 32 after the loop resolving process has been performed in order to enable the user to loop infinitely rather than once.

The problem lies in that the two graphs are not Bisimilar. We can regard the two workflow graphs as process definitions in CSS [6]. Bisimulation is used to show that two processes behave in the same way. We can prove that the two graphs are not Bisimilar.

**Notation**

<table>
<thead>
<tr>
<th>$Q \rightarrow_{a} P$</th>
<th>$a$ indicates an action. $\rightarrow_{a}$ indicates a transition from process Q to process P is possible by performing action $a$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha.Q \rightarrow_{a} Q$</td>
<td>Prefix operator (.)</td>
</tr>
<tr>
<td>$\equiv_{def}$</td>
<td>Links a process identifier to a process</td>
</tr>
<tr>
<td>$E + F \rightarrow_{a} Q$</td>
<td>Summation. Either E or F but not both can perform a $a$ transition to get to process Q.</td>
</tr>
</tbody>
</table>

Figure 32 can be written as the following CSS process.

$$P =_{def} w.c.c.d.0$$

Figure 33 can be written as the following CSS process.

$$Q =_{def} w.Q'$$

$$Q' =_{def} c.d.0 + c.Q'$$
4. LOGIC PROGRAMMING DOMAIN

Proof:

Assume some B Bisimulation exists such that

\[ B = \{ (P, Q), (c.c.d.0, Q'), (c.d.0, Q'), (d.0, d.0), (d.0, Q') \} \]

<table>
<thead>
<tr>
<th>P \rightarrow_w c.c.d.0</th>
<th>Q \rightarrow_w Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.c.d.0 \rightarrow_c c.d.0</td>
<td>Q' \rightarrow_c Q'</td>
</tr>
<tr>
<td>c.d.0 \rightarrow_c d.0</td>
<td>Q' \rightarrow_c Q'</td>
</tr>
<tr>
<td>d.0 \rightarrow_d 0</td>
<td>Q' \rightarrow_c Q'</td>
</tr>
<tr>
<td>d.0 \rightarrow_d 0</td>
<td>d.0 \rightarrow_c ??</td>
</tr>
</tbody>
</table>

This cannot do a c transition.
Hence we have a contradiction,
Bisimulation B does not exist. P and Q
are not bisimilar

Hence the new representation of Figure 33 does not reflect Figure 32’s workflow
specification. The two specifications do not behave in the same way, hence one can
do something which the other cannot. The user may have explicitly wanted the
workflow to follow the sequence of Figure 32 and not act as a loop. It is impossible to
tell when the user means for a loop to be specified or when a singular loop is used
which should not be reduced to an infinite loop. As we have seen loop resolving
cannot be performed regardless as the two graphs produced do not behave in the same
way.

Due to this I adopted the first solution of making looping implicit in the plan.

The Design

Loops are specified in the edge progression part of the workflow unit, indicating
where the loop wants to loop to. The loop point can be specified by indicating the
form name in the plan to look back to. Loops may only ever point to a form entry
point. Pointing at any other point would lead to inconsistent plans as it would be
possible to enter a workflow unit half way through.

The full details of how the loop resolving is performed are given in Section 4.3.5.

In the workflow specification at the directed graph level (the highest level of
abstraction) the workflow unit that is going to loop is specified as achieving the goal
even though it has not really achieved it. There are a number of reasons why this was
done:
4. LOGIC PROGRAMMING DOMAIN

1. It prevents continued planning to get to the goal which will never be used since the plan will loop.
2. It means that only one plan is generated for the loop. This means only one plans needs the loop resolving process. This avoids adding performance overheads to the planning process.
3. The searching for a loop point is very efficient since we need to only check the time points that are directly ordered before the goal.

The other solution would have been to have in every plan an implicit loop. This way it would not be necessary to force the planner to achieve the goal in the looping case. The major problem with this solution is the need to undertake loop resolving with every single plan generated.

The Implementation

The design was translated to the Meta language of the Event Calculus planner. Looping was implemented in the hospital model at 2 different points. One when the doctor rejects the pharmacist’s changes and the other where the nurse refuses the controlled drugs.

4.2.3.6 Ordering Resolution of Before and Happens Predicates

The Problem

In the example plans provided by Shanahan [33] temporal orderings before(T1,T2) are used before the action that will occur at time point T2. An example from one plan is illustrated:

```prolog
happens(formCreation(form1,User), T1,T2):-
  happens( createFormelement(b1, textbox, [size=2]),T1),
  before(T1,T2),
  happens( createFormelement(b11, textbox, [size=2] ),T2),
```

Figure 34: Shanahan’s ordering of before predicates

Ordering of the before predicate in the preconditions causes duplicates in complex plans. If the planner finds itself in a state where it already has all the actions in the example in the plan and it encounters the composite action again there is a problem.

Encountering `happens(formCreation(form1),T1,T2)` for a second time the planner will decompose the compound action into the three different preconditions. The first precondition is:

```prolog
happens( createFormelement(b1, textbox, [size=2]),T1),
```

The Event Calculus planner checks if this action is already in the plan which it is. The Event Calculus planner avoids adding duplicate actions. It then process
4. LOGIC PROGRAMMING DOMAIN

4.3 ASLDICN Event Calculus Planner

The Event Calculus planner uses a Meta language to describe the Event Calculus axioms. Meta languages are explained in Section 2.5.1. The planner given \( \Sigma \) (domain dependent sentences), optionally \( \Delta_0 \) (initial state) and \( \Gamma \) (goals) produces \( \Delta \) (narrative/plan). All the predicates of the Event Calculus seen in Figure 2, are supported.

Support for compound actions is built into the planner. The planner also supports recursion. Recursion is achieved through the use of compound actions, one compound action representing the base case and another representing the recursive case. E.g.

```
happens(formCreation(form1,User), T1,T2):-
    happens( createFormelement(b1, textbox, [size=2]),T1),
    happens( createFormelement(b11, textbox, [size=2] ),T2),
    before(T1,T2)
```

**Figure 35: The correct ordering of before predicates in compound action**

All the *before* predicates in the hospital workflow specification where changed to support this new layout.
4. LOGIC PROGRAMMING DOMAIN

4.3.1 ASLDICN Event Calculus Planner Structure

The Problem

The Event Calculus planner used for the project was Dr. Murray Shanahan’s [33] ASLDICN (Abductive SLD with Integrity constraints and proof by Negation) planner with compound action support. This planner is an adaptation from one published in one of Dr. Shanahan’s research papers [28]. Although this paper defines the underlying structure of the planner it does not completely document this version of the planner. Hence the problem was to initially understand how the planner worked.

The Design

The call structure of the planner was mapped to diagrams to help understand how the program worked.

%Base case
happens(go_to_room(R,R),T,T).

%Recursive case
happens(go_to_room(R1,R3),T1,T3):-
   connects(D,R1,R2), towards(R2,R3,R1),
   happens(go_to_room(R2,R3),T2,T3), %Recursive call
   before(T1,T2), happens(go_through(D),T1),
   not(clipped(T1,in(R2),T2)).

Figure 36: Recursion in the event calculus
Abdemo Top represents the start of the planning process. The planner repeats two main phases, the iterative deepening and refinement process.

Figure 37: Overview of ASLDICN planner with compound actions support
The depth in the iterative deepening process represents the length of the current plan. Abdemo represents the actual resolution of the Event Calculus axioms (which are detailed in Section 2.2). The special case included in the Abdemo's rules that is not seen in EC is the resolution of an expand predicate. The expand predicate is used to indicate that there is a compound action in the plan that needs to be further refined.
The refinement process deals with the actual flagging of the compound actions for refinement (by wrapping them in an expand predicate). In plan specifications the actions that are executable (i.e. not compound) have to be defined. The refinement process uses this information to identify if an action in the plan is not executable in which case the refinement process flags the compound action for expansion when it enters the next iterative deepening process. The flagged compound action is added to the goal list. If all actions are executable then there is no need to refine since the plan represents the solution plan.

4.3.2 Converting to a Conditional Planner

The Problem

The original planner only supports the generation of a single plan. We decided in the outset of the project to use conditional planning (discussed in Section 3.1.5). We want the planner to generate multiple plans representing the different ways of reaching the goal. The problem is how to convert the planner to generate all possible plans. Importantly ensuring that this does not cause infinite looping and no redundant plan solutions are generated.

The Considered Solutions

I produced two main experimental versions of the Event Calculus planner during development. Both solutions involved more advanced debugging information than the original program and also had support for breaking out of the Meta language and accessing Prolog. This allowed Prolog traces to be implanted in a workflow specification to allow debugging at specific points in the planning process. As the planners were developed they were continual tested on a range of test case plans and on the original plans given by Murray Shanahan [33].
1. **Iterative deepening solution (version 1)**

The iterative deepening was kept from the original planner. Preventing infinite looping and limiting the axioms backtracked to was achieved through a dynamic failure predicate.

For infinite looping the idea was to assert a failure predicate indicating the reason why the next iterative deepening step was going to be performed. If the reason was not due to exceeding the depth limit we know that a further iteration will not achieve anything new. This prevents infinite looping. Infinite looping is undesirable since the planner will be used in real time as part of the web based iWFMS. Looping infinitely would leave the web page hanging.

At each iterative step the failure predicate is checked to see if we are infinitely looping. When a solution is generated all these failure predicates are removed. Then when backtracking the program will keep failing the infinite looping test until it finds a branch which has positive progress. It fails the infinite looping test due to the lack of any failure predicates. When I say positive progress I mean that an action is added to the residual which causes a failure predicate to be asserted. Then the planner can continue planning forwards at it passes the infinite looping test.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The planner supported recursion. Only through iterative deepening can the planner handle recursion.</td>
<td>The planner was computationally intensive due to the nature of iterative deepening.</td>
</tr>
<tr>
<td></td>
<td>There is an overhead of having to store dynamic predicates indicating the reason for failure.</td>
</tr>
</tbody>
</table>

2. **Depth first solution (version 2)**

This solution was based on removing all the code that enabled the iterative deepening and replacing it with a single depth first solution. Prolog cuts where used to prune the search tree and ensure only the desired solutions where returned. In locating were these cuts should go substantial time was spent tracing through the program and analysing the code. Also dead rules were used to enable multiple solutions. Dead rules are what I refer to as rules which always fail.

```prolog
prologRule(Var) :- fail.
```

These are used were you do not want to prune the search tree but rather force further backtracking.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The planner is computational more efficient.</td>
<td>Recursion cannot be handled since the planner would loop infinitely due to the nature of depth first search.</td>
</tr>
<tr>
<td>No need to detect infinite looping.</td>
<td></td>
</tr>
</tbody>
</table>
The Selected Solution

Version 2 was the final version of the planner. Although limiting recursion it significantly improved the performance time of the planner. Iterative deepening guarantees the optimal solution but with conditional planning we are not concern with the optimal solution but all possible solutions that hold. Thus depth first search is more suitable.

The Design

The new planners design is the similar to that seen in Figure 37: Overview of ASLDICN planner with compound actions support.

![Figure 40: Depth first Event Calculus planner](image)

The main difference with this version and the original planner is that the iterative deepening process is no longer present. The planner finds all actions for a specific abstract level, it then goes about refining them and finding all solutions at the refined abstraction layer. This is repeated until the planner can no longer refine the actions in the plan indicating there are no more compound actions.

The Implementation

The planner was initially converted to run under Sicstus Prolog. Then a total of 5 cuts were added and 1 cut was removed from the planner. All changes to the planner were documented clearly indicating that they were an addition/removal by myself. Also the changes were documented with when they were made and why.
4. LOGIC PROGRAMMING DOMAIN

4.3.3  Advanced Planner and Event Calculus Features

A number of further extensions were added to the planner including extensions to the Event Calculus. These features identified themselves as being useful through the prototyping of the Event Calculus workflow specifications.

4.3.3.1 Impossible Predicate

The Problem

It is desirable to assert in certain situations in a workflow specification that it is impossible for an action to occur. An example situation in workflow may be that a workflow unit is impossible if a particular fluent does not hold.

The Design

The idea of an impossible predicate was taken from a research paper by Rob Miller [17].

\[
\text{impossible}(\text{Action}, \text{Timepoint})
\]

An action is impossible at a particular time point.

The \textit{impossible} predicate can be introduced into the Event Calculus planner by altering the clipped and declipped axioms EC1 and EC2 encountered in Section 2.2.

\[
\begin{align*}
\text{Clipped}(t1, f, t4) & \iff \exists a, t2, t3 \ [ \text{Happens}(a, t2, t3) \land t1 < t3 \land t2 < t4 \land \\
& \quad [\text{Terminates}(a, f, t2) \lor \text{Releases}(a, f, t2) \lor \text{Impossible}(a, t2)]] \\
\text{Declipped}(t1, f, t4) & \iff \exists a, t2, t3 \ [ \text{Happens}(a, t2, t3) \land t1 < t3 \land t2 < t4 \land \\
& \quad [\text{Initiates}(a, f, t2) \lor \text{Releases}(a, f, t2) \lor \text{Impossible}(a, t2)]]
\end{align*}
\]

\textbf{Figure 41: Clipped and Declipped with impossible support}

The Implementation

The design was implemented in Prolog. Although it was initially felt that the \textit{impossible} predicate was required in the hospital specification, as the specification versions developed it become apparent that the impossible predicate was not quite right for what was trying to be achieved in the hospital model. Instead the \textit{occurred} and \textit{notOccurred} predicates which we see next were used. This does not remove from the fact that the \textit{impossible} predicate has a part to play in workflow specifications, just not the hospital model.

4.3.3.2 Occurred and NotOccurred

The Problem
In workflow it is desirable to be able to have conditions on actions that do not add to the residual. The problem is that the preconditions of actions are regarded as being something in the future that the planner must achieve and thus they add to the residual. We need preconditions that do not add to the residual in workflow frequently to ensure workflow units are only select if certain paths have been followed previously. For example:

The drugCheckFailure workflow unit should never occur if the planner uses the edgeProgression action which indicates drug success. Thus we need some form of condition in the drugCheckFailure workflow unit to indicate such a condition.

The planner encounters such a condition as in Figure 42 and we want it to evaluate it as indicating the drugs check must not have been successful. What in reality happens is the planner treats this as a goal and achieves the edgeProgression of success, doing the exact opposite of what we wish the condition to do!

The Design

The solution to the problem was to introduce occurred and notOccurred predicates into the planner. Evaluation of such actions has a special resolution process which checks the residual rather than adding to it.

The idea of the occurred predicate was taken from a research paper by Rob Miller [17].

occurred( Action, Timepoint )

An action has occurred at a specific point in the past. This can be a generic check by leaving the time point unbound.

notOccurred( Action, Timepoint )

An action did not occur at a specific point in the past. Note we cannot use a negation in Prolog with the occurred predicate to achieve the notOccurred predicate. This is due to the occurred and notOccurred predicate having to occur in the Meta language used by the Event Calculus planner.

The Implementation

The occurred predicate and notOccurred predicate were implemented in Prolog in the Event Calculus planner. The predicate occurs in the Abdemo process with deals with resolving of all the Event Calculus predicates.
4.3.4 Mapping Partially Ordered Plans into Total Ordered Plans

The Problem

The Event Calculus planner supports the generation of partial ordered plans. Even with what would appear to be a total ordered plan specification we come across partial orderings. This is primary based on the way the \textit{holds_at} predicate is resolved (The \textit{holds_at} predicate in the Event Calculus is detailed in Section 2.2).

The \textit{holds_at} predicate provides a convenient way of decoupling the need to satisfy a fluent and the action that initiates the fluent. Hence many actions reflecting different paths in the workflow can initiate a fluent. Therefore \textit{holds_at} predicates are very useful in our specifications of workflow.

Based on the resolution of the \textit{holds_at} predicate if something holds at time point T3 it actually occurs before T3. This introduces a new time point into our plan in which the action is carried out. There is a temporal ordering indicating that the new time point is before T3. This introduces partial ordering into our plans.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Fig43.png}
\caption{Example of a partially ordered plan with redundant nodes}
\end{figure}

In Figure 43 we see the effect of the \textit{holds_at} predicate in plan generation. T33 represents when the action occurs and T31 represents the time point introduced by the \textit{holds_at} predicate. The arrow between T33 and T31 is representative of the temporal ordering \textit{before}(T33, T31).

Note that the plan is partial ordered. The problem is how to reduce this plan into a total-order plan. The goal of the project was to only support total ordered plans.

The Design

Firstly in detecting a partial ordering we check to see if there is any time point which is linked to by two different time points. Then we can achieve a mapping to a total ordering by:
1. Cutting the redundant time point.
This is achieved by removing the before predicates from the temporal orderings that link the left and right tree with the redundant node. In Figure 43 this would be the before(T30, T31) and before(T33, T31) orderings.

![Figure 44: Removing the redundant time point](image)

2. Joining the left tree with the right tree
From removing the redundant node we are left with left and right trees which are not connected. The last time point of the left tree is attached to the last time point of the right tree. This is achieved by adding a before constraint to the temporal ordering. In Figure 44 this would be the before(T30, T33) ordering.

![Figure 45: Joining the left and right tree](image)

3. Linking the right tree to the children of the redundant node
Now we have linked the left and right trees we need to reattach the resulting tree to the child node of the redundant node that was removed. The first node of the right tree is attached to the children of the redundant node by adding a before constrain to the orderings. In Figure 45 this would be the before(T33, T18) ordering.

![Figure 45: Joining the left and right tree](image)
Very importantly the process then has to start checking for partial orderings from the first node of the left tree (In Figure 46 T30). This is required since the right tree may contain further partial orderings.

Now in reducing to a total-ordered plan we are eliminating certain plans which are valid! In Figure 43 T33 could occur at anytime before T31. For example T33 could occur then T24, T30 and T31. Within the plan there is no dependency upon the T33 and the rest of the time points. In the final solution of Figure 46 we are implying that T33 must occur only once T30 has occurred. This is possible due to the nature of the workflow specifications. The specifications have been given always in mind that a total-ordering will be used. In viewing the specification as being total-ordered we expect that within a compound action everything has to happen within that compound action before any other actions can be undertaken. In fact through the Event Calculus holds_at predicate and the partial order Event Calculus planner the actual plan states that this is only one interpretation of the plan generated. This indicates a discrepancy in the planner being partial ordered and us as seeing everything as a total ordering.

Even though we are using conditional planning we are only generating a specific subset of the actual valid total-ordered plans.

**The Implementation**

The design was implemented in Prolog as a separate module from the planner. This was tested before adding to the Event Calculus planner. The Planner before it returns the plan calls the loop reduction process. If there is ever a failure to map the plan to a total-ordered plan then that particular plan is failed and the planner backtracks trying to generate other solutions.

### 4.3.5 Reducing Loops

**The Problem**

Implicit loops in the plan need to be converted to explicit loops. This is required to enable the iWFMS to be able to execute the plan following the possible infinite
looping of the user. Due to the multiple languages used in the iWFMS this fix could be applied at a number of levels.

The Considered solutions

1. Implement in PHP at the graph object construction level
   Reduce the loops manipulating the graph object construction which is used to represent the plans and orderings in PHP. The graph consists of objects representing nodes and edges. The nodes contain happens predicate objects and the edges are temporal orderings.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The high level of abstraction makes the reasoning and designing of the algorithm easier.</td>
<td>Involves the manipulation of many objects which carries an associated overhead.</td>
</tr>
<tr>
<td>Strong support in methods and functions provided by the core library and the graph object.</td>
<td></td>
</tr>
</tbody>
</table>

2. Implement in PHP at the predicate object level
   Implementing the change before the predicate objects used to represent the plan actions and orderings are converted to the graph object representation.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smaller number of objects to be manipulated reducing performance overheads.</td>
<td>Still carries an overhead having to deal with the manipulation of objects.</td>
</tr>
</tbody>
</table>

3. Implement in Prolog.
   Implement the change in the Prolog planner so the solutions produced carry the explicit loop.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excellent built in libraries for list processing.</td>
<td>Lacks the methods and attributes associated with the predicate object. Though arguably it does not need them as a predicate has a built in meaning in Prolog unlike PHP which needs the predicate object to be built to gain such meaning.</td>
</tr>
<tr>
<td>Will work with the iWFMS without the need for any changes.</td>
<td></td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to implement the loop reduction in Prolog. The performance of the loop reduction is likely to be faster in Prolog since it is optimised for dealing with lists and
does not have the need to deal with the overhead of objects. It will be important to ensure the Prolog solution does not involve substantial backtracking since this can add a serious overhead to the performance.

The Design

In reducing the loop three different operations have to be performed. The logic and flow of data of the process are illustrated in the following figures.
This process is repeated for every loop identified.

**The Implementation**

The solution was implemented in Prolog and attached to the Event Calculus planner. Before the plan is returned in the planner a call is made to the loop reduction process passing the temporal orderings and actions. The result returned represents a new set of orderings with the loop reduction performed.

### 4.3.6 Knowledge Base Integration

**The Problem**

The planner needs to know information about the user that is logged into the system, specifically the groups they are a member of. This is required since this effects the way that the plans are generated when the user invokes the Event Calculus planner.

**The Considered solutions**

1. Introduce a knowledge base within Prolog
   Sicstus Prolog provides an API to function with the Berkeley DB Database. This database could be used to store the groups of the user when they log into the system.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduces the overhead of having to pass all the groups a user is a member of before planning is undertaken.</td>
<td>There is an overhead of updating the database when the user logs into the system.</td>
</tr>
</tbody>
</table>
Introducing a knowledge base provides for a wide range of extensions within the project. For example it would be possible to allow Prolog CGI to maintain state information in a similar way to PHP.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple to implement since it can use the underlying core PHP libraries of the iWFMS.</td>
<td>An overhead is introduced into the most performance critical part of the system, the addition of new workflow items. The planner has to fetch the information from the database when planning.</td>
</tr>
<tr>
<td>It makes testing the planner in isolation difficult due to the dependency on the interaction with iWFMS PHP elements.</td>
<td>An overhead is introduced into the most performance critical part of the system, the addition of new workflow items.</td>
</tr>
</tbody>
</table>

2. Use the mechanism of transferring Data to Prolog from PHP. When the Event Calculus planner is invoked by the user to add a new workflow item to the system, the relevant information would have to be fetched from the PHP system database. This would then be saved as a temporary file and passed to Prolog. We discuss in more depth in Section 6.4 how files are used to transfer data to Prolog.

The Selected Solution

I decided to adopt the knowledge base solution. Both solutions provide an additional overhead into the addition of workflow by having to look up information in the database. The second solution however has an overhead of having to convert the information into a Prolog format and writing it to a file. The database processing is more complicated in the second solution due to the difficulty in storing array data in the database. In PHP we have to use a serialise function which allows the storing of arrays in database fields. Therefore whenever the groups are fetched from the database by PHP this data needs to be unserialised. This all adds up to a larger overhead than the first solution. The overhead added by the first solution to the login point is minimal as this page is already incredibly fast. A small overhead will not make the login process too slow.

The Design

Two main classes were created in Prolog using the object orientated feature of Sicstus. The first provided a generic interaction layer with the Berkley database. The next class provided specific functions in looking up users and the groups associated to them.

The Implementation

The design was implemented in Prolog

4.4 HTML Type Checking

The Problem
Typing is needed to ensure that all the `createFormElement` actions in the workflow specification have the correct syntax. This is required to ensure that the generated plans when being executed do not have any undesirable effects due to the bad specification of form elements. Since the form elements are represented as Prolog predicates one loses the usual flexibility to quickly run HTML code and check for errors. The typing helps overcome this problem. Typing is generally loose in HTML but it is highly desirable to ensure that it is stricter within the iWFMS. Within websites the focus of HTML is on making the publishing of a site as easy as possible for users ensuring at least something is displayed if there are errors. Within the iWFMS the focus of the HTML is to generate forms which have well founded formats and correct functionality. Incorrect HTML for forms breaks this correctness and could lead to unexpected results being displayed to the user. For the purpose of clarity in the following examples we shall ignore the `createFormElement` actions and look at the typing in terms of the HTML that would be generated from the `createFormElement` action.

Typing requires checking that attributes are valid in form elements.

e.g. ( <select width="100"> </select> )

Width is an allowed attribute in the select tag

Also checking that the children of a parent HTML tag are allowed. When we say children we mean tags enclosed within a parent tag.

e.g. <select>
    <option> </option>
    <optiongrp> </optiongrp>
</select>

The select tag has two children, option and optiongrp.

The Considered Solutions

1. **Online verifier**
   Convert the Prolog `createFormElement` actions into HTML and pass to an online HTML verifier.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The tool for verifying the correctness of the HTML is already available.</td>
<td>The process is time intensive since a webpage has to be posted to and then the results need to be processed and returned to the user.</td>
</tr>
<tr>
<td>Since the tool is online we can seamlessly deal with updates to the HTML specifications.</td>
<td>Converting the Prolog predicates into HTML takes the same time as is required to deliver the HTML form to the user.</td>
</tr>
</tbody>
</table>

2. **W3C HTML DTD file**
Fetch typing information from a W3C HTML DTD file and use this to generate typing rules in Prolog. These are then incorporated within the planner to check form element specifications.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The typing rules can be generated separately from the planning process. Therefore additional overheads to the planning process are reduced.</td>
<td>Complex DTD processing is required in order to generate typing rules.</td>
</tr>
<tr>
<td>Typing occurs in the planner so an additional overhead in the process is introduced.</td>
<td></td>
</tr>
</tbody>
</table>

3. Separate the workflow specification and HTML
Represent the HTML Forms in separate HTML files and link to the file within the workflow specification rather than including the information directly in the plan.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since the HTML is in a separate file it can be quickly run to check typing. Although this is possible it is limited by the fact that files would not represent complete HTML documents, but rather HTML forms. Additional HTML would have to be added.</td>
<td>Complicates the workflow specification since many references are required within it to form element’s names. Removing the HTML form predicate means that the name of the element is tangled in the HTML.</td>
</tr>
<tr>
<td>Implementing JavaScript becomes very difficult since there is no abstract representation of the form elements. This means there is no separation from HTML syntax and there is no logical separation between the form elements. The HTML code would have to be broken apart using complex regular expressions to add the relevant JavaScript to the form elements.</td>
<td></td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to adopt the W3C HTML DTD file solution. All but the third solution provide an overhead to the planning process. The selected solution provides the smaller overhead. It also does not require complicating or limiting the workflow specification unlike the third solution.

The Design

The source of the typing system is the W3C HTML DTD documents. These provide information on more than just HTML form elements but information on all HTML tags. There are specially three different typing checks provided by the DTD document that we are interested in:
1. The different HTML tag attributes allowed
   E.g. ( <table width="100" > ) width is allowed in the table tag

2. Any types associated with values of attributes
   e.g. ( <table width="100" > ) width must be an integer

3. The valid children allowed for different HTML tags.

An example of how information about 1 and 2 are represented is given in the following figure.

```
<!ATTLIST td
%attrs;
  rowspan  %Number;   "1"
  colspan   %Number;   "1"
>
```

*Figure 49: W3C typing for the td tag*

We can extract from this information the attributes that the <td> tag is allowed and typing information on the values they can be assigned (in this case Number).

The valid children are represented in a syntax similar to relative expressions. An example expression for the <table> tag is given in the following figure.

```
(caption?, thead?, tfoot?, (col*|colgroup*), (tbody+|tr+))
```

*Figure 50: W3C typing information on the valid children for the table tag*

This tag indicates the children of the table tag can consist of the following:

1. 0 or 1 <caption> tag
2. 0 or 1 <thread> tag,
3. 0 or 1 <tfoot> tag,
4. 0 or more <col> tags or zero or more <colgroup> tags
5. At least 1 <tbody> tag or at least 1 <tr> tag

This information is converted into a Prolog rule based file. This brings the typing rules into a format which can be easily interacted with in the Prolog type checker. A Prolog type checker is used as it needs to be incorporated into the Prolog Event Calculus planner. The two different predicates generated from the W3C DTD are the following:

- Attribute typing - Mapping attributes to tags and including constraints on the types of the values assigned to the attributes where specified in the W3C HTML DTD. These rules add constraints to the Definite clause-grammar rules used within the typing system which we see later in Section 4.4.1. An example is given for the HTML tag ‘select’ with the attribute tag ‘dir’.
attributeType(select, dir, AttributeValue) :- AttributeValue == ltr ; AttributeValue == rtd.

- Children typing - Mapping Tags to their type relative expression which reflects the children that are valid for this tag. These rules are simple Prolog Facts associating the string type relative expression with the name of the Tag. An example is given for the select tag.

typing(select, "(optiongrp|option)+").

The Type checking is used inside the planner before evaluating the actions of the formCreation compound action. A call to the type system exists in the workflow specification in the form of a precondition for every createFormElement action. The precondition either contains children information or attribute information of the createFormElement action it appears before. The typing precondition breaks through the Meta interpreter by being wrapped in a ‘prolog’ clause ensuring that when it is resolved by the planner it is evaluated in Prolog. This means the type system can called without having to deal with the Meta language the Event Calculus planner requires. The result of the typing system failing is that an error message is returned to the user and planning stops.

The Type checker performs two different types of checks utilising the dynamic typing rules. Checking valid attributes and value pairs and checking valid children elements of a HTML tags.

Figure 51: HTML typing architecture data flow
Further details on how valid children elements are checked and how attributes are check is discussed in section 4.4.1 and section 4.4.2.

The Implementation

The type rule generation was implemented in PHP. The DTD processing was achieved using the core DTD library produced in PHP. Converting to Prolog utilised the object grammar design for Prolog seen in Figure 69: UML diagram of the Prolog grammar. The rest of the type system was implemented in Prolog. In the following sections we look in more detail at this.

4.4.1 Valid Attribute Value Pairs.

The Problem

How using the dynamic Prolog HTML typing rules and the call from the planner the attributes of the HTML form elements are checked.

The Design

The Planner calls the typing system with a precondition that contains the attributes for a HTML form element. The Type system extracts all these attributes.

Attributes can take the form of:

Attribute=”value” or Attribute.

The attributes and values are tokenised through Definite clause grammar rules. Specifically tokenising Attribute=value or Attribute.

The Definite clause grammar rules have constraints associated with them. These constraints try to find a Prolog fact that matches with the relevant form element and the specific attribute and value which are being tokenised. These constraint rules are the attribute typing rules that are pre-generated from the W3C HTML DTD document.

If the input is successfully tokenised then we know that the HTML form element is type correct. Failure at any point in the tokenising indicates type failure. If a constraint rule is not found this implies that the specified attribute does not exist for the relevant HTML form element, so typing fails.

Typing of the actual values assigned to the attributes is achieved through using Prolog’s built in predicates which allow the checking of types. For example the number(X) predicate checks to see if the X is a number.

The Implementation

The design was implemented in Prolog.
4. LOGIC PROGRAMMING DOMAIN

4.4.2 Valid Children Elements of a HTML Tag.

The Problem

How the dynamic Prolog HTML typing rules and the call from the planner are used to check if the children elements are valid.

The Design

Type checking of children of a parent HTML tag is achieved through performing a special relative expression match given a list of children and the type expression of the parent. The type checker when called from the planner is provided with a form element name and a list of children that that form element has associated with it in the workflow specification. The type checker converts the list of children into a string. Then it can find the type regular expression for this particular form element if any by matching to the relevant children typing fact.

The formalisation of the W3C HTML DTD regular expression syntax had to be produce in order to process the type regular expressions. BNF was used as this leads to a very natural conversion to Definite clause-grammar rules in Prolog which are used later as part of the typing system.
The Backus-Naur form specification was converted into a Definite clause-grammar in Prolog. Definite clause-grammars provide a convenient way of tokenising a string input placing constraints on the different token strings. I developed a method of using the Definite clause-grammar rules to build up a Prolog based predicate representation of the type regular expression. The Definite clause-grammar consumes the string tokens and builds the Prolog predicate constructs.

Example:

Before tokenising

(col*|row*)

After tokenising

or(*(&(c,(&(o,1)))),*(&(r,(&(o,w)))))

Figure 52: BNF specification of the type expressions
Converting the type regular expression strings into Prolog predicates allows easier and more complex processing within Prolog. It would have been possible to simply keep the string format but this would have meant checking character codes as defined by Sicstus Prolog which would make the code unreadable and considerably time consuming to produce.

Once the type regular expression is in predicate form, typing is performed. This process is analogous to trying to find a solution path in the abstract syntax tree that matches the children string of form element being checked.

Prolog rules are given for the logical operators (OR, &) and the arity constructs(+, ?, *). Character by character comparisons are made between the children string and the predicate form type relative expression. The logical or arity rule definitions constrain how the match may occur and how many times. Matches are consumed and the process continues trying to match with the rest.

- **Or**
  Can be naturally represented by two different Prolog rules indicating the two different ways the regular expression can be selected
- **Conjunction**
  A single Prolog rule indicating both cases have to hold

We look at the main arity constructs involved:

- **Zero or more**
  This can be achieved through finding one or more matches or no matches at all.
- **One or more**
This can be achieved by finding a single match and then finding zero or more matches
- Zero or one
  This can be achieved by finding a single match or no match at all.

If a remainder exists after the typing process has finished then the children string does not obey the typing of the predicate type relative expression. A partial match may have been found which implies partial type ordering conformance.

e.g. ?(a) matching "aa", matches the "a" but leaves an "a".

The Implementation

The design was implemented in Prolog.
5 Server-side Domain

5.1 Overview

This section looks at the server-side components required to facilitate the running of the iWFMS from the plans generated in the logic programming domain. Also the server-side components required to provide overall management functionality. The main components covered consist of:

- **XML specifications**
  Looking at a flexible way of updating data from HTML forms.

- **Core index page management**
  The page that shows all the active workflow items and their state, in terms of the form elements that have been completed for that workflow item. A workflow item is conceptual how the user sees the workflow in the iWFMS. A workflow item will consist of all the plans generated by the Event Calculus planner for a specific workflow specification. The core index page also provides access to all the functionality provided in the iWFMS though a navigation tool bar.

- **Planning and plan execution**
  How the workflow plans are used to govern execution in the iWFMS.

- **Core libraries**
  Fundamental libraries which are used throughout the server-side domain.

5.2 XML Specifications

5.2.1 Updating Databases with Form Posted Data

_The Problem_

When a HTML form is posted by the user in the iWFMS the data needs to be stored in a particular database or perhaps a group of databases. It is desirable for these databases to be separate from the iWFMS system database to prevent tangling of the different data. Also the target databases may already exist in other locations and thus could not be part of the iWFMS system database. The problem is deciding how to specify where the form data is going to be directed.

_The Considered solutions_

1. XML specifications to indicate where the data is to be directed
XML specifications would have to be provided for every single form linking the form elements to databases, tables and fields which are the destination for the data. A XML schema would be used to specify the structure of the XML files.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many tools are available for the creation of DTD and XML files. This will ease the development of the XML files.</td>
<td>A large number of XML specifications would have to be produced</td>
</tr>
<tr>
<td>Strong API’s in PHP which handle the processing of XML documents</td>
<td>XML is not very human readable making hand editing difficult</td>
</tr>
</tbody>
</table>

2. Explicit in the workflow specification where the data should go. The workflow specification contains the references to the databases, tables and fields that the data should be directed to.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>No major changes to the architecture of the iWFMS. The updating simply involves the executing of the plan.</td>
<td>Entangles the plan with database management.</td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to adopt the XML specification solution. The primary reason for this decision was based on the fact that the destination for the posted data may change and this change does not promote any core changes to the workflow specification. Thus it is desirable to abstract the workflow specification and the database information so the destination for the posted data can be changed without having to edit the workflow specification.

The Design

For simplicity a number of assumptions where made:

1. All the form elements names are the same as the database table’s field name.
2. The XML file is named after the form its federating the data for.

The XML schema for the specification was design to provide maximum flexibility. The schema supports any XML file that needs to describe data being imported or data being exported. This flexibility allows the reuse of this schema in the Workflow unit state specification problem seen later in Section 5.3.1.
The XML file following the schema supports multiple databases, multiple tables and multiple fields.

**The Implementation**

The XML file and schema were implemented using XMLspy 2004, a visual XML editor.

### 5.2.2 Converting XML Specifications to PHP

**The Problem**

Transforming the XML specifications into a format which can be flexibly used in PHP. Ideally abstracting from the fact that the data came from XML.

**The Design**

Abstracting from the XML is achieved by breaking it into objects which represent the XML information. Since the XML is conveying data about databases, database tables and database fields the abstraction was based on these objects.
5. SERVER-SIDE DOMAIN

The core processing of the XML was handled by the core XML library. This returned a stack containing the XML. The XML was then transformed into the designed object structure using PHP.

5.3 Core iWFMS Index Page

The core index page is the page which provides access to all the functionality of the iWFMS. It provides the point where all the workflow items already added to the system can be accessed. The core index page needs to provide a number of functions.

1. Display all active workflow items.
2. Display the state that has been achieved by those workflow items
3. Checking whether the user can access a workflow item at a particular stage

5.3.1 Workflow Unit State Specification

The Problem

Each workflow item depending on the stage it is at in the execution will have associated with it certain data which has been collected from the form posts. This data reflects the state of the workflow item. It should be noted that this data could be distributed over multiple databases. It is highly desirable to allow users to gain some visual conception of the workflows state.

The state can be independent of the values stored for a workflow item and instead focus on a presence check. The data cannot be shown since only certain group members may be authorised to see such data. Putting the data on the core index page means everyone who could log in could see the information.
The problem is how to specify the databases, tables and fields that are going to be tested for the presence of content.

The Considered Solutions

1. Fixed PHP/HTML
   The core index page is hardcode in PHP and HTML. The databases, tables and fields required are coded into the PHP processing.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The page is generated quickly.</td>
<td>Each new iWFMS domain would require a specific index page to be coded.</td>
</tr>
<tr>
<td></td>
<td>To make changes the person must know HTML/PHP.</td>
</tr>
</tbody>
</table>

2. XML specification
   The core index page is produced from an XML document. The XML document contains the database, tables, fields and display values which are used in the core index page to identify the state of workflow items. PHP code would process the XML and fetch the data from the specified locations. The data is displayed dynamically each time the page loads.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows the use of XML based tools to design and implement.</td>
<td>There is a Performance overhead of having to process the XML file.</td>
</tr>
<tr>
<td>Interoperability with any application that may be created to manage the core index page.</td>
<td>Every time the page loads the XML file has to be processed introducing a repeated overhead.</td>
</tr>
<tr>
<td>Separates the PHP and HTML processing from the database source information. This makes it simpler to change the XML database specification and the PHP/HTML.</td>
<td></td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to adopt the XML specification solution. The level of separation between PHP/HTML and the database source information is highly desirable. As the system develops the user is likely to want to add new items to the core index page. The database sources would need to change while the PHP and HTML can be completely abstracted from. Anyone who understands XML could make the change. While in the first solution the change would require someone with PHP, HTML and Database knowledge. Also the XML specifications of the workflow system are seen in the future as being managed by an application. With a fixed PHP/HTML solution it would be difficult to manage the core index page since the editor would have to rip apart PHP and HTML. It would be fair simpler to manage given an XML specification. The performance issues are a minor problem but in this case the advantages of avoiding code tangling out weighs the small performance problem.
5. SERVER-SIDE DOMAIN

The Design

The schema designed in Figure 54 is reused in this problem. It has exactly the same fields as needed in this problem but rather than indicating the data’s destination it is used to indicate the data’s source.

The Implementation

The XML files were implemented using XMLspy 2004.

5.3.2 Generating HTML From XML Specifications

The Problem

The XML specification provides the source of the information. Once the XML specifications have been turned into a PHP object based structure it is necessary to fetch the information from the relevant databases/tables/fields as specified in the XML. Once this information has been fetched it has to be displayed as HTML. It is very important that the HTML and the PHP/Database processing are separate. The goal of separating HTML and PHP is something that is held throughout this project. Separation provides less entanglement of design and logic.

The Design

The HTML generation is separated from the PHP/Database processing by using a coreWorkflowActionPacket class as a container for the data. This class is used to gather all the information for the core index page from the relevant databases. The Object is then passed to the HTML processing which utilises it to produce the HTML. In this way there is no HTML processing in the database/PHP processing. The HTML can abstract from the source of the data allowing other utilities in the iWFMS to utilise the same HTML output as long as the data fits in the coreWorkflowActionPacket object.

The Implementation

The design was implementation in PHP.

5.3.3 Group Based Form Access

The Problem

To access a form for a workflow item the user has to be a member of certain groups. These groups are specified in the formEntry action of the plan. This data has to be fetched from the plan and used to restrict which user can access the form.

The Design
A workflow item may have a number of plans associated to it. Each may reflect the different groups that can enter a form. Due to this every group from every plan has to be fetched. There then has to be a check on whether the current user is a member of this set of groups.

The Implementation

PHP was used to dynamically create JavaScript which alerted the user with an error message if they did not have permission to access the form. If they do have access then no JavaScript message appears.

5.3.4 GUI

The Problem

We have discussed the components of the iWFMS core index page. These need to be incorporated into a HTML page and a layout needs to be designed.

The Design

The GUI was based around tables in HTML. This allows the segmentation of the different parts of the GUI making styling simpler.

All styling within the GUI was achieved using Cascading Style Sheets (CSS). These are used to associate styling to classes. The class are then referred to in the HTML tags. This promotes separation between the HTML code and the design. The colours and layout of the entire system can all be changed from the CSS file.
Figure 57: Core index page interface of the iWFMS
The Implementation

The design was implemented using PHP, HTML and CSS.

5.4 Server-Side Planner

The server-side planner is responsible for generating the HTML and JavaScript for the user. We will refer to this part as client preparation planning. The planner also deals with checking if a user has met the conditions to progress to a new workflow unit when they post a HTML form. We will refer to this as server condition planning.

5.4.1 Plan Representation

The Problem

We need a way of defining the workflow plans in PHP. These plans will be used by the plan execution architecture to run the iWFMS form system. The plans enter PHP from the Prolog planner without any structure. We will discuss how the Prolog plans are converted into the high level PHP workflow plans in Section 6.3. These unstructured plans need to be represented in PHP with an abstract structure which provides the methods, attributes and abstractions that allow effective plan execution. Leaving the plans unstructured and executing them would mean planning would have to be overly complex, dealing with issues such as having to navigate the unstructured plans and retrieving the relevant data at a low level of abstraction.

The Design

The analogy of the directed graphs was used in the Event Calculus workflow specifications (Section 4.2.1). I decided to adopt the same abstraction for the PHP plans. Directed graphs provide a very effective representation for Event Calculus plans. The Event Calculus plans consist of temporal orderings and actions that occur at time points. The nodes can be seen as representing time points while the edges represent the temporal ordering constraints. Thus directed graphs model everything that is required of Event Calculus plans.

The designed class model is based around nodes, edges and a graph plan which contains the relevant edges and nodes.
5. SERVER-SIDE DOMAIN

The name of the node class represents the time point. The data of a node represents the *happens* action that occurs at that time point.

The GraphPlan class records the start node and the current position. These both reference the name of a node. The start node is required to ensure the planner knows where to start in the directed graph. The current position provides the planner with knowledge about where to continue planning from. Also the history of a plan is stored indicating the different time points that have already occurred.

*The Implementation*

The design was implemented in PHP.

### 5.4.2 Plan Processing

*The Problem*

The Event Calculus planner can generate a number of different plans. When following workflow plans in PHP we have the problem of how to coordinate the execution of each of the plans. Each of the plans will contain slightly different actions at different time points as each plan represents a different way of achieving the goal. All the plans will have different time points at which the actions occur, primarily due to the way the Prolog Event Calculus planner backtracks when generating multiple plans. This means even through plan actions may be the same in different plans they could have different time points. Due to this the way of managing the plans needs to be independent of the time points. The problem is how to utilise multiple plans in plan execution.
The Considered Solutions

1. Reduce all the plans into a single directed graph

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows simpler execution of the plan by the iWFMS since only one plan would have to be considered.</td>
<td>The algorithm to merge the graphs is particularly intensive. It involves having to compare one node with every other node in the all the other plans.</td>
</tr>
<tr>
<td>Saves on storage costs since one plan is likely to be smaller than several plans with similar actions.</td>
<td></td>
</tr>
</tbody>
</table>

2. Navigate all the plans throwing away those that no longer apply
This would keep track of all of the required plans as it progressed through execution. While tracing through all the plans if different nodes are encountered in the plans the planner knows it has a conditional point. Depending on the choice that’s followed the plans that do not follow that choice can be discarded.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>No processing is required during the plan generation.</td>
<td>There is an overhead in the database for storing all the plans.</td>
</tr>
<tr>
<td>The number of plans decreases rapidly due to discarding plans that no longer reflect the path of the user.</td>
<td>At the start of the plans there would be an overhead since all plans generated would have to be examined.</td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to adopt the second solution. This decision was primarily made due to the importance of performance in the plan generation for the iWFMS. This is the point were the user will have to wait for Prolog to generate plans. It is important that the overheads added to the point are minimised to ensure the user will have a fast response time. Also the second solution spreads the performance overhead over the whole execution lifetime, rather than solution one which does it all at one time point.

The Design

The design is based on the concept of an action packet.

Figure 59: Specification of Action Packets
The action packet consists of an action and a list of graphs each representing a plan. The action represents the next action of the associated graph plans. If more than one graph plan exists in an action packet it means that the plans all have the same next action.

Upon a request to fetch the next action from all the current plans the required graph plans are fetched from the database. These plans represent all the active plans at a particular moment in the life of a workflow item. As each graph plan object records its current position the next action can be found by seeing which actions follows the current position. Action packets are generated for each graph containing the next action and the associated graph plan. Then action packets with the same next action are merged.
If all Action packets are merged into a single action packet then there is no decision to make. If however as seen in the Figure 60 we encounter different action packets then a choice has to be made about which action packet is going to be followed. We keep the action packet belonging to the chosen action and discard the other action packet. This is because the discarded action packet's plan/s no longer represent the path that the execution of the plan has taken.

The Implementation

The design was implemented in PHP.

5.4.3 Plan Execution Architecture

The Problem

Producing an execution framework that uses the produced plans to guide the running of the iWFMS HTML form system.

The Design
I decided to try and base the architecture on the abstract formalisation of a multi-agent introduced in Section 2.3.1. The server-side planner can be seen as being analogous to a robot following a plan and interacting with its environment. Using this model will help highlight the key points of the planner, how it interacts with its environment and its internal logic.

**The Environment**
The environment represents the different states the HTML forms are at. The environment is acted upon but it is not sensed. The environment is not sensed since it is never available to the server-side planner. The planner located on the server is unaware of what is happening on the client-side when the form is being processed. The planner can act on the environment by the way it constructs the HTML forms which are delivered to the client. When the user posts the HTML form the planner becomes aware through messages what the final state of the environment is before the post. The states the environment can be in can be constrained by JavaScript. This helps ensure the order in which different form elements have to be completed. We look at the JavaScript planning in more detail in Section 5.5.

**Messages**
Messages can be regarded as analogous to the Posting of data from a HTML form. The Message space is the POST structure which is used to carry the users posted data in PHP. The index to the messages is based on the name of the form element desired. The value of the message represents the value entered by the user in the form.

**The See Process**
The see process is used to process users form posted information. When a user submits a form they change the state of the environment and post data which is
detected by the see component. The see function indexes into the message space and retrieves the desired message values. The messages fetched are dictated by the plan.

*Action Process*
This is used to decide what action to undertake. In some cases there is only one action to undertake while in others multiple actions are possible. The main decision points the action process encounters are when multiple groups are encountered, when multiple edge progressions are possible and when multiple paths are possible for form inputting. The action process decision can be based on:

- The see process
  Used to decide which edge progression action to follow and which group to follow.

- The memory
  Used when planning is progressing without updating the database.

- Reduced plans
  Used when planning has just begun and the active plans have to be fetched from the database.

The action process may also choose to update the plans.

*Reduce plans Process*
This process reduces the plans to action packets, returning only those plans which contain distinct next actions.

*Plans*
These are represented as directed graphs that contain the workflow. Initially there may be many plans but over the process of the workflow item the relevant plans may decrease. Within the plans there are two types of action:

- Constraint actions (*edgeProgression* and *formEntry*)
  These help the planner make decisions in collaboration with the see process about which action to select when conditional actions are encountered and thus which plan to follow. These actions are not executed but used as constraints.

- Normal actions (*formSubmission*, *entry*, *createFormElement* and *databaseFetch*)
  These actions are executed by the planner. For example the *createFormElemen* action is executed in order to generate the relevant HTML.

*Memory*
This stores the current action packets until they are used to update the plans.

*The Goals*
The goals can be seen as trying to reach the goal node which is the last action in all plans bar plans that represent the looping cases. These never achieve the goal since they loop.
The execution of actions
Constraint actions are not executed. Normal actions when executed generate HTML forms and JavaScript for the user.

The Implementation

The architecture was implemented in PHP.

5.4.4 Handling Interruptions in Plan Progression.

The Problem

The planner may be at a point in the plan where it is waiting for a HTML form post. The user could simply close the browser leaving the planner at a point where it will never receive the HTML form submission!

The Considered solutions

1. Plan up to submission but do not save any plan progress
Not updating the database with the plans indicating the progress to the submission event. This means if the user destroys the browser session the plan is still in a consistent state. When the user next comes to edit the workflow item its current position would be at the start of the workflow unit.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>A consistent state for the user is achieved if the browser is interrupted without requiring any backtracking.</td>
<td>When the user posts the HTML form the planner would be activated. It would fetch the plans from the database whose current position would be at the beginning of the workflow stage. It therefore needs to repeat the planning process to progress through all the actions until it reaches submission. Therefore there is redundancy, repeating the planning process.</td>
</tr>
</tbody>
</table>

2. Plan up to submission and save progress
Once the planning reaches the submission point in the plan it has completed everything required for the client-side HTML form to be delivered. Therefore update the current position of the plans in the database.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server-side planning is not required to repeat previous planning steps.</td>
<td>Checking on the core index page would be required for every workflow item to ensure the plans where at the start of a workflow unit (formEntry). The user may have closed the browser leaving the plan</td>
</tr>
</tbody>
</table>
mid way through a workflow task.  
Backtracking would have to be performed for those workflow items that were not in the expected position.

**The Selected Solution**

I decided to use first solution. The primary reason for choosing this solution was the smaller overhead. A single plan has to be re-planned in order to get to the submission point. In the second solution every workflow item on the core index page would have to have their plan’s fetched from the database irrelevant of whether or not they were in the expected current position. Then on top of that it may be the case that multiple plans have to be backtracked to get them to the correct current position. This is a considerable overhead.

**The Design**

The planner when planning after the submission of a form uses the same functionality as that used to generate the HTML forms and JavaScript for the user. The difference being that it does this in a passive mode, not actually performing any actions. This is due to the fact that it is re-planning to get to the point in the plans that represent actions it can act upon (specifically `edgeProgression` actions).

**The Implementation**

The design was implemented in PHP.

5.4.5 Client Preparation Planning and Server Condition Planning

**The Problem**

In preparing the HTML form for the client-side the plan has to be followed to gather the actions which create the form elements and JavaScript controls. A design decision already made in Section 5.4.4 is that the database is not updated with this planning information while preparing data for the client. It is only updated after the form is posted from the client. Consequently when the client posts the contents of a form to the server it has no idea which paths have been previously taken by the planner. An example situation is that the path followed through the plan for building the client HTML form is dependent upon the group the user that accessed the form was a member of. Workflow may continue on a different course based on that group so the server-side planning when it repeats the planning needs to identify the path that was followed when preparing the HTML form for the user.

**The Considered Solutions**

1. Embed the data that was used to make decisions in client preparation planning in the clients HTML form.
The data could be contained in hidden form elements which are never seen by the user. When the user posts the form this data is passed to the server. Thus the server has available the relevant data to choose the correct plan path.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The server condition planning algorithm can be identical to the client preparation planning algorithm. Both have exactly the same data available.</td>
<td>The data has to be encrypted in the hidden form elements. Otherwise the user could subvert the planning system by saving an offline copy and changing the hidden data.</td>
</tr>
</tbody>
</table>

2. Save the data in the PHP session

Saving the relevant data in the session will mean it will be accessible by the server condition planning component since the session is similar to a global collection of variables.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The server condition planning algorithm can be identical to the client preparation planning algorithm. Both have exactly the same data available.</td>
<td>The session variables have to be explicitly removed from the session.</td>
</tr>
<tr>
<td>No encryption is required since the session data is only accessible at the server.</td>
<td>There is a small overhead for the web server in maintaining individual sessions for every user.</td>
</tr>
</tbody>
</table>

The Selected Solution

I decided to adopt the first solution of embedding the data that was used to generate the plan path in the clients form. I decided to use this solution since there is no need to destroy the information as in the session solution. The data posted in the HTML form only exists at the time of being posted. This leads to a more elegant solution. The overhead for the sessions is very small but perhaps in the future development of the project more intensive data may be needed. It comes down to the balance of which is more intensive encryption or session overhead. I feel the session overhead is larger since it is dependent on the number of users which if large implies a multiplied overhead.

The Design

The specially encrypted data passed to the server condition planning process in the form is treated in the same way as the form data posted from the user. The server condition planning process when it encounters a conditional action checks the posted data. As far as the planner is concerned the data could have been sent by the user.

The Implementation

The solution was implemented in PHP. The hidden form elements where attached to the Prolog generated form and their values encrypted using the ‘mcrypt_ecb’ function. This allows the specification of the encryption algorithm to use and the key.
5.4.6 Segmentation between the Treatment of Conditional Actions

The Problem

Conditional actions need to be handled differently when encountered in the client preparation planning and the server condition planning.

Client-side conditionals are when different plans indicating different paths that could be followed are encountered while performing client preparation planning. This would be the preparing of HTML Forms/JavaScript for the user at the client-side. For example two different paths may exist each indicating a different way that a user can fill in a form. This would be used to create JavaScript to constrain the user. This is explored in more detail in the Client-Side Planner, Section 5.5.

With client-side conditional actions we cannot pick one plan to follow. The information that decides which plan path is followed is not available until the HTML form is at the client-side. It is therefore impossible for client preparation planning located at the server, while preparing the HTML forms/JavaScript to foresee what the user is going to do, since it has not happened yet!

Server-side conditionals indicate the possible transitions from a workflow unit to a new workflow unit. These will always have the information available to decide upon which plan path to follow if any.

Therefore with client-side conditionals we have to act upon and follow multiple plans since we don’t know which ones will be used. This is an unusually planning situation, in the past examples we have only seen execution of single paths. There is the problem of how to achieve multiple plan execution and how this affects the rest of the planning process.

The Design

The solution adopted when encountering client-side conditionals is to follow all the plans until they converge to a single plan and hence a single action. The plans representing the different client-side conditional actions must all converge upon the same action since the client-side conditionals all occur in the same form. They all have to have the same form submission action.
5. SERVER-SIDE DOMAIN

The process starts when multiple action packets are encountered while preparing the HTML forms/JavaScript. Multiple action packets indicate conditional actions. The plans associated to the action packets all have to be processed. The way that these plans are processed may depend on what types of conditional actions have been encountered in the action packets. Once all the plans have been processed the next action packets are fetched. If we find more than two we know the plans have not converged and we continue progressing along all the plans. When we fetch the next action packets and find only one, the plans have converged and we can stop processing all the plans.

The Implementation

The design was implemented in PHP and incorporated into the plan execution logic. Exactly how the solution fits into the plan execution logic can be seen in Figure 65: Plan execution logic.

5.4.7 Saving Workflow States for Iterations

The Problem

A plan may reach a loop point looping back to a previous time point. As the planner has been progressing to this point it has been discarding plans that no longer apply due to decisions made by the user. The problem being that when the loop moves back to the previous time point it has fewer plans that when this point was first encountered. Hence some of the options available at that point when it was first encountered will not be possible now.

For example:
Note that Figure 63 is a simplified representation. Precisely it would represent two plans one for the succeed path and one for the fail path.

When the check drugs node is encountered in Figure 63 and the fail drugs path is chosen the other plan reflecting the succeed drugs check path is discarded. The reason plans are discarded is that they no longer reflect the path of execution. When prescribe drugs is encountered again through the loop as seen in Figure 64 we no longer have the choice to follow the succeed drugs check path.

The solution has to be performed during workflow execution since we cannot determine the paths that will be followed by the user beforehand. It could be performed before runtime by an exhaustive evaluation but this carries an undesirable overhead.

The Considered Solutions

1. Store execution plans at decision points
   The active plans could be stored in the system database. This would require recording at decision points the different plans reflecting the different choices. Thus when the looping case reached a decision point after looping it could locate the different plans in the system database. The need to store the active plans at decision points would only need to start when a loop re-entry point was encountered. The loop re-entry point at a node can be detected by seeing if there is an edge pointing to that node that is not on the history time points. The plans record their past time points. The past time points have to be looked at since past nodes point to current nodes. An example of a re-entry point for a loop was seen in Figure 64, 'Prescribe drugs' is the re-entry point for the loop.

   It may seem that the overhead of storing the state could be reduce by only storing the discarded plans at decisions points rather than all the plans. This is not a good idea
since this requires a continuous process of storing discarded decisions points with every loop. In each loop a different set of paths might be followed implying a different set of discarded plans.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The plan paths that use the node that loops can detect when they have encountered the start of the loop. It encounters a node which has an edge going back to a node in its history. Thus it knows it can stop recording active plans at decision points as it has passed the loop.</td>
<td>Once a loop re-entry point is encountered plans that do not lead through the node that loops will have to record every single decision point from then onwards. There is no way for the planner to know that the loop node will not be encountered. Therefore as far as it is concerned it may encounter the node that loops at the next step.</td>
</tr>
</tbody>
</table>

2. Store the active plans at points where loops re-enter
Check at each node encountered during planning seeing if it is a loop re-entry point. At such points store all the active plans in the system database. When the loop encounters the re-entry point it loads the active plans from the database.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is no need to continuously record the different plans for different decision points.</td>
<td>There is an overhead of having to check for every node if it is a loop re-entry point.</td>
</tr>
</tbody>
</table>

The Selected Solution
I decided to adopt the second solution, storing the state at loop re-entry points. The decision comes down to the balance of overheads. The storing at the decision points solution has a smaller overhead in terms of checking nodes to see if they are loop re-entry points. This checking only needs to be done up until a loop re-entry point is encountered. On the other hand the storing at re-entry point solution requires performing this check on every node. Ultimately, having to store the state at every single decision point after encountering a loop re-entry point is a substantial overhead. This requires database updating which is going to be far more intensive than checking if a node is a loop re-entry point. The second solution thus has the smaller overall overhead. A smaller overhead means a more responsive system for the user.

The Design
As defined in the design of the looping in plans in Section 4.2.3.5 the loop re-entry points will only ever be formEntry actions. All workflow plans are at this point on the iWFMS core index page. The formEntry action is checked at this point to produce the functionality for group access. So now in addition the check for the loop re-entry point can be implemented here.

The Implementation
The design was implemented in PHP.
5.4.8 Plan Execution

The Problem

We have seen how a number of the problems encountered during planning are dealt with. Finally we need to take the architecture, problem solutions and plan representations to devise how the logic for the plan execution takes place.

The Design

The plan execution logic is given an action to execute the plan up to. It continues to fetch action packets and perform the current action until it reaches the goal or finds the action to plan up to or it encounters a conditional action. A conditional action is marked by encountering multiple action packets. These decision points need to be acted upon in order to decide upon which action packet to follow so the process can continue trying to reach the desired action. We can detect what type of decision has to be made when we reach a conditional action by the name of the predicate encountered.
• Conditionals in *FormEntry* occur when the way a HTML form is displayed or the workflow path followed is different based on which group the user who entered the form is a member of.

• *edgeProgression* conditionals indicate multiple paths that can be taken evolving to new workflow stages. Note that if no plan holds meaning no possible *edgeProgression* action holds with what the user has posted in the HTML form, planning stops since the workflow should not progress to a new workflow task.

• *entry* conditionals represent client-side conditional actions. These define different ways the user can fill in the data in a form. This is using the solution already seen in Figure 62: Following multiple plans. The processing of multiple plans and generation of JavaScript plans is detailed in the Client-Side Planner, Section 5.5.

---

**The Implementation**

The design was implemented in PHP.

### 5.5 Client-Side Planner

Some degree of planning at the client-side enables us to guide and constrain what the user can input in order to successfully submit a form. This helps us prevent invalid states from being submitted and ensures that the user follows good practices in filling in forms.

#### 5.5.1 Plan Representation

**The Problem**

Parts of the plans generated from the Event Calculus planner can reflect different ways the user can fill in a form. These parts are the *entry* actions design in Section 4.2.2.4. When the planner is preparing HTML forms for the user it needs to convert these parts of the plans to a JavaScript plan representation. These plans would then be used by a plan engine based in JavaScript. There can be multiple plans for *entry* actions reflecting different ways that the user can enter information in a form in order to achieve a valid form.

**The Design**

JavaScript plans can be represented as arrays in JavaScript. A number of arrays may exist indicating multiple plans. The actions of the JavaScript plan are conditions on form elements. These conditions are generated from the *entry* actions. An example plan is given in the following figure.
In these figures we see a plan consisting of two actions. One checks that the age element of the form has a value that is greater than 18. If this condition holds then the following condition is used to check that the ward form element is set to adult. Therefore the plan reflects that if the person’s age is greater than 18 they must be assigned to an adult ward.

The Implementation

The implementation was achieved by writing the plans initially out in static JavaScript. This was then used as a base for the dynamic generation of the JavaScript from the server-side plans. PHP was used for the dynamic generation of the JavaScript.

5.5.2 Plan Execution Architecture

The Problem

The architecture for the client plan executor is different from the server plan executor since we now have direct access to the environment. Also this architecture has to be built upon a different platform since it is run at the client rather than the server. The problem is designing how the plan executor will run under this different situation.

The Design
5. SERVER-SIDE DOMAIN

The Environment
The environment represents the different states the form elements and the user can be in.

The See process
This function maps the form elements to their associated values.

The Message Store
The message store is used to represent events generated by the user. Events could be the result of clicking on a form element or clicking away from a form element. Such events generate messages which the plan executor responds to. These events are the points where the planner is activated.

The Plans
The plans represent a sequence of constraints within the JavaScript syntax on form elements. The actions of the plans are analogous to the constraint actions seen in the server-side planner (Section 5.4.3). This type of action is not executed but used to help decide upon which plan path to follow.

The Action process
This process deals with selecting actions based on the see process which provides information about the form element’s values. There may only be a single plan and thus no decision to make. If there are multiple plans the decision to follow an action is only made for the first action of a plan. Which initial action to decide upon is decided by testing each of the first actions of the plans until one holds from the information provided by the see process. This plan is then used to further test the next constraint of the plan and so on. If multiple first actions hold each of the plans is tested to see if any of them reach the goal.

The Goals
The goals of the plans can be regarded as ensuring that all the conditions in the plan hold based on the data in the HTML form.

The execution of actions
Within the planner the actions are not executed. Since we are constraining the user all we need to do is find a valid plan path. The planner affects the environment by alerting the user if it cannot find any valid plan path.

The Implementation

The architecture was initially implemented in static JavaScript. This was then ported to dynamic generation through PHP.

5.5.3 Plan Execution

The Problem

The plan executor needs to be able to handle two different situations:

- **Form submission**
  When the user inputs information into a form and then submits the form the planner will have to ensure that a valid plan path is reflected by the values that are held in the form. The form submission should be prevented if such a plan cannot be found.

- **Real time user input**
  a. **Guidance through a plan**
     As the user inputs information the planner needs to be aware of the current position in multiple plans and what effect this has on the current form element being filled in. Alerting the user if they have failed to achieve any plan as a result of the data they entered into the form.

  b. **Temporal ordering restrictions**
     The planner needs to be able to handle temporal restrictions on the form elements. Indicating in which order form elements have to be completed. Therefore we have to look at the impact that a real time input action will have on the rest of the form. For example one form element being completed may have the effect that another disabled form element can be activated for input.

The Design

Form submission

All plans are searched to try and locate one for which all the conditions hold. If such a plan can be found then the form is allowed to post. Otherwise the form is not posted and the user is displayed an error. Finding no plan indicates they have not followed one of the available ways of inputting data into the form.

Real time user input

Guidance through a plan
A user clicks away from a form element, which generates an event (JavaScript OnBlur event).

Each action in the plans has an associated form element. For example in Figure 66 we see the form elements age and ward are associated to the first and second constraint respectively.

The planner tries to find a plan that holds up to the form element that caused this event. If such a plan cannot be found the user must be alerted. They are not following a valid plan path and thus must be alerted since they will not be allowed to submit the form if a valid plan path has not been followed. If a valid plan can be found then there is no need to interact with the user since they are following a valid path.

**Temporal ordering restrictions**
Event Calculus plans have an explicit concept of time. Hence we can take the order of the entry predicates to indicate the order the elements must be filled in. Using this we can generate a timeline JavaScript array indicating which order the form elements must occur in. Capturing the event where the user clicks on a form element we can respond by enabling the next form element in the timeline.

**The Implementation**

The design was initially implemented with static JavaScript plans. Then once all functionality was working the system was incorporated into working with dynamically generated plans produced through PHP.

### 5.6 Core Libraries

A large number of libraries were produced in order to achieve the smooth running of the iWFMS.

#### 5.6.1 Messaging

All debug, error messaging and performance messaging passes through set PHP functions. This way the format of all debugging and error messages is at a unified point. It also makes it very easy to turn debugging and error messages off simply by disabling the message function.

#### 5.6.2 Database Management

Deals with the updating, deleting, querying and inserting for databases. The PHP PEAR [22] database library model was integrated into the database management in order to provide an abstraction layer between the physical database. This provides database interoperability.

#### 5.6.3 Session Management
Deals with the management of the session recording when a user is logged into the system.

### 5.6.4 XML Processing

The core processing of the XML utilises the xml_parser object in PHP. Functions were produced to deal with the character data, start tags and end tags. These were then used to initiate the xml_parser which is passed a string containing the XML. The XML string is read in using the file reading libraries developed as part of the iWFMS core libraries.
6 Logic and Server-Side Meeting

6.1 Overview

This section looks at the how the logic programming domain and the server-side domain interact. The server-side domain needs to convert Prolog plans generated by the logic programming domain into a more manageable form. The server-side also needs to be able to generate HTML typing rules for the logic programming domain.

6.2 Representing Prolog in PHP

The Problem

The plans generated by the Prolog planner when they reach PHP are nothing more than strings. The meaning of Prolog’s predicates/lists/atoms etc is lost in the passing. Processing Prolog at the string level would be incredibly messy involving many complex relative expressions. There needs to be an abstraction which regains the meaning of Prolog’s terms in PHP removing the need for messy string processing. In the HTML type system Prolog type rules have to be dynamically created from DTD documents. In the Prolog CGI based form creation unit which generates HTML forms for the user the plan actions indicating form creation have to be passed from PHP to Prolog. Therefore PHP needs to be able to produce Prolog and convert Strings into Prolog representations.

The Design

I decided to base the solution on the interpreter design pattern. This design pattern provides a method for representing a languages grammar and an interpreter which interprets sentences in the language. Classes are used to represent each grammar rule. Through adapting this design pattern we can represent the grammar of Prolog and provide a means of interpreting Prolog in PHP.

One of the reasons the interpreter design pattern is so appealing is that it makes changing and extending the grammar easy. We can represent some of the more abstract constructs such as happens predicates and temporal orderings (before predicates) as seen in the Event Calculus through extending the grammar.

One of the warnings of the using the interpreter design pattern is that it should only be used when the grammar is simple. This is primarily due to the vast number of objects that can be required to express a complex grammar. In Prolog we do not have a hugely complex grammar with a small group of base types and a small number of higher level constructs.
We can reduce the granularity of the Prolog grammar to avoid overheads associated with large numbers of objects. For example the Atom construct in Prolog is a sequence of characters and the atom construct can be represented as a single class. It is not necessary to further break down the grammar of the atom so that it consists of characters and each character is an object.

Most design patterns incorporate some high level abstract class which is used to process the subclass generically. PHP is loosely typed so variables can represent objects of any type. Thus the need for a high level abstract class is not as apparent since the actual typing of the variable containing the object is not important. Instead the use of such classes is more for sharing common methods and behaviours. In the design we hence do not have an abstract class Prolog.

The following constructs where accounted for in the design:

**Concrete types**
- Predicates
- Lists
- Atoms
- Character Strings
- Quoted strings
- Variables
- Expressions (x =:= y)

**Complex types**
- Rules
- Definite clause-grammars
The design was implemented in PHP. One of the interesting problems that arose in the implementation was due to PHP not supporting method overloading. The Prolog predicate object required two constructors. One to support the case where a string is used to create a predicate object and another where a name and an array of values are given. The lack of overloading was overcome by creating a constructor function which used the number of variables passed to the constructor to dynamically call another function. The dynamic function name is the constructor name but with the
number of parameters concatenated to the end. Thus, distinction based on the number of parameters is possible.

```php
//Constructor
function prologPredicate(){
    //Create a dynamic function name
    $methodName="prologPredicate".func_num_args();
    //Get the arguments passed to this function
    $arg = func_get_args();
    //Call the new function with arguments
    call_user_func_array(array(&$this, $methodName), $args);
}
```

**Figure 70: How to overcome constructor overloading limitations in PHP**

It is not possible to do the same for overloading types since PHP is loosely typed.

### 6.2.1 Parsing Prolog into PHP

#### The Problem

The interpreter design pattern does not explain how to approach the parsing. There is still the issue of how we convert a string representation of a Prolog construct into the Prolog object hierarchy.

#### The Considered Solutions

1. Relative expression matching

   Use relative expressions to try and break apart the string. For example using relative expressions to find all the values of a predicate.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concise solution that requires only a couple of lines of code.</td>
<td>Relative expressions can be notoriously hard to read and difficult to reason about.</td>
</tr>
<tr>
<td>PHP uses the Perl relative expression library which is regarded as one of the fastest implementations of relative expressions.</td>
<td>Limited ability to deal with recursive string patterns when matching strings.</td>
</tr>
</tbody>
</table>

2. Custom parser

   Create a parser from scratch that processes character by character the input string, breaking up the string.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The parser can have its performance tailored to the domain it is going to operate in (in our case Event Calculus)</td>
<td>There is an associated overhead of having to perform character per character parsing.</td>
</tr>
</tbody>
</table>
The Selected Solution

The solution selected was the custom parser. The custom parser is required due to the complicated nature of the possible sub-expressions in Lists and predicates. It is desirable to only break up the current level of variables for the construct being parsed. For example:

“happens(action(run(x,y,[1,2])),T1,T2)”

On the first parse of this string we want to break up the different values of the happens predicate but we do not want to further break the actual values up yet. The values returned should be:

“action(predicate(x,y,[1,2]))”, “T1” and “T2”

Then each of these values can be parsed in turn and so on until the object structure is produced. This allows a well-structured recursive algorithm to be written which simplifies the parser design. This sort of parsing is not possible with regular expressions due to the nature of the recursive patterns that can occur within the different predicate values.

Although a custom parser will be used, this does not mean that relative expressions have been completely ruled out. They will still play a part in preparing the strings before they enter the parser.

The Design

The parser will be given before predicate strings or happens predicate strings. These are the predicates that make up the plan returned by the Event Calculus planner. Due to this we can assume that the parser is specifically taking predicate strings and returning predicate objects. Within the predicates there may be a number of complex data types. For example one predicate value may consist of a number of predicates, lists, strings or atoms and they in turn may also have various Prolog terms. This requires a level of recursive parsing until a value is in an object form.
Since all Prolog data types implement the toString() method conversion back to Prolog is achieved by invoking this method on any Prolog object.

The Implementation

The design was implemented in PHP.

6.3 Moving Prolog Plans to the Server-Side Domain

The Problem

The plans generated by the Prolog Event Calculus planner are interpreted in PHP as nothing more than strings. We have seen how the parser helps convert Prolog string predicates into Prolog objects. The parser needs to be incorporated into the process of converting the string plans produced by the planner into the directed graph object.
model previously seen in Figure 58: Representation of plans in PHP. The directed graph object model produced is used by the iWFMS to execute the plan.

The Design

The object model of the Prolog grammar seen in Figure 69 was extended to support the specific constructs used in Event Calculus plans.

The reason the grammar was extended was to simplify the process of converting to the graph plan object structure. The before and happens predicate classes provide specific functions that ease the processing and fetching of the different values of the predicates. This is possible since we know the exact format of these predicates. The prologPlanOrderingCollection class provides a collection of before predicate objects and the prologPlanCollection provides a collection of happens predicate objects.

The string plan is broken into two strings, one with the temporal orderings the other with the happens action strings. These are then processed by the parser generating a collection of happens predicate objects and a collection of before predicate objects.
The Prolog plan ordering collection object is used to generate all the edge objects of the graph and all the nodes. Each time point in the before predicate indicates a node and the two time points that occur in the before predicate indicates an edge between those two nodes. The Prolog action collection is used to find the actions which occur at specific time points and attach the happens predicate object as data to the relevant node. Finally the directed graph is stored in the database.

The Implementation

The design was implemented in PHP. Initially the process was built to support a single before predicate and a single happens predicate which were static (not generated dynamically by the Event Calculus planner). Once it was established this case worked real static plans where used. Finally the process was passed dynamic plans from the Event Calculus planner.

6.4 Prolog Functioning as CGI

The Problem
When workflow is added to the iWFMS a user needs to be able to invoke from a webpage the creation of the workflow plans. This requires a level of interaction between the webpage and Prolog. The Event Calculus planner needs to run and return the plans generated. With Prolog form generation which we see in Section 6.5 there is a need to post a plan to Prolog again requiring interaction with Prolog and the posting of a webpage. When the user logs into the iWFMS Prolog needs to be activated in order to record in the knowledge base the groups the user is a member of. All these interactions require runtime running of Prolog. It is not possible to perform the interaction offline beforehand thus avoiding the need to invoke Prolog. The interactions can be handled by utilising Prolog as CGI. Technically any programming language can function as CGI. The problem is how to effectively run Prolog as CGI when it was not initially designed for such a purpose.

**The Design**

Prolog can be run as CGI by using a PHP wrapper script which invokes the Prolog engine from within PHP. Prolog can be invoked indicating Prolog files to load and goals to initially achieve once loaded.

![Diagram of Prolog function as CGI](image)

When a request is made to the server and Prolog needs to run as CGI, PHP can call the wrapper script activating a Prolog process.

```bash
$cgiOutput = `sicstus --goal $goal. -l "$cgiPrologScriptToLoad"`;
```

The wrapper script indicates to Prolog the CGI Prolog script to load and the goal to immediately try and achieve once the script is loaded. The goal represents the name of the CGI script to run.
Once the Prolog process has returned PHP regains control and can continue. CGI output is recorded from the data directed to the server terminal, so by writing results within Prolog we can return the required data back to PHP. This solves the problem of how to invoke Prolog as CGI but it does not provide Prolog with any data that may have been posted to the server. Such data is needed to indicate plan actions for creating a form or which Event Calculus workflow specification to use.

This data has to be passed to Prolog through a temporary file created by PHP.

```bash
$cgiOutput = `sicstus --goal ini('TemporayDataFileName'),$goal. -l "$cgiPrologScriptToLoad"`;
```

Figure 77: PHP wrapper script for launching Prolog with file transfer

When the wrapper script is called an extra initial goal is given which ensures that Prolog loads the temporary file and the second goal then runs the relevant Prolog CGI script. Only one file can be loaded from invoking Prolog so the temporary file has to be loaded by this new initial goal. Once Prolog has returned PHP can clean up the temporary file since it is no longer needed.

No matter what happens during the running of Prolog CGI, it must always halt Prolog once finished or if an error occurs. If this does not happen then the webpage will hanging as PHP never regains control since Prolog never returns.

The Implementation

The required Prolog CGI scripts where written in Prolog. All CGI scripts are wrapped in a special Prolog predicate which ensures that even if an error occurs the Prolog process halts.

6.5 Prolog Form Generation

The Problem

The HTML forms for the iWFMS have to be generated from action predicates in the plan. The problem is how to process the generation of these forms.

The Considered Solutions

1. Prolog
   Utilising Prolog running as CGI to generate the HTML Forms

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The predicates have a built in meaning in Prolog and are easy to process.</td>
<td>There is an overhead associated with having to use Prolog as CGI since a temporary file has to be created to pass data to Prolog.</td>
</tr>
</tbody>
</table>
The support of Definite clause-grammars is available which allows the generation of strings based on predicates.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs with no significant performance overhead.</td>
<td>The solution involves building up strings from the predicate values to create form elements. Dynamic string generation code can be quite complex and time consuming to produce.</td>
</tr>
<tr>
<td>The Prolog object grammar means PHP can process predicates just as effectively as Prolog.</td>
<td></td>
</tr>
</tbody>
</table>

**The Selected Solution**

I decided to adopt the Prolog solution. The primary reason for this choice was the ease with which Prolog could be used to generate forms from predicates compared to the complexity of having to deal with dynamic string building in PHP.

**The Design**

The solution was based on an adaptation to the PiLLoW Prolog Sicstus library. This is a product of the PiLLoW project [24]. The project is based on the investigation of Prolog generating HTML forms. This library takes a different approach to the generation of HTML forms than is the focus of this project. The PiLLoW library looks at generating very simple HTML forms from the simplest possible Prolog predicates. Within this project the goal is to enable the user to specify complex predicates which generate form elements as customisable as the user desires. This enables the user to achieve any type of form element they want with any parameters while the PiLLoW library does not allow this.

**The Implementation**

An extension module was attached to the Pillow library which provided rules for handling the `createFormElement` predicates given in the Event Calculus workflow specification.

### 6.6 DTD to Prolog Definite Clause Grammar Rules

**The Problem**
The W3C DTD file containing information about valid HTML has to be converted dynamically into a set of Prolog Definite clause grammar rules that are used by the type system.

The Design

The DTD file is parsed into an array structure using the PEAR [22] DTD library. The DTD values are then parsed into Prolog Definite clause grammar rule objects. This class is defined as part of the Prolog grammar object model of Figure 69. These Definite clause grammar rules are written to a special file in Prolog format using the file writing core libraries. This file is included by the Prolog type system when it runs.

The Implementation

The design was implemented in PHP.
7 Extensions

7.1 Overview

This section looks at a number of extra features that were not in the original set of goals for this project.

7.2 Visualisation of Workflow

The Problem

From talking with Dr. Murray Shanahan, the developer of the original Event Calculus planner and from my own experiences I know that the task of checking Event Calculus plans are valid is a frustrating and hugely time consuming process. The process involves:

1. Generating the plans from Prolog.
2. Copying them into a text editor.
3. Breaking up the individual plans.
4. Drawing on paper all the time points that occur in the happens predicate.
5. Drawing all the edges represent by the before constraints.

Even with a small number of plans this process is difficult. Within complex models such as the hospital which can contain around 19 plans each with around 42 time points, checking for valid plans can take weeks.

We already have the tools to take Prolog plans and convert them into graph objects within PHP. Hence we can regard the problem as how to visualise the graph objects rather than Prolog plan strings.

The Design

An open source tool called GraphViz [1] is available from AT&T research labs that produces graphs in Scalable Vector Graphics from a specification language called dot. Scalable Vector Graphics (SVG) [36] is a language which uses XML to define graphics. Since SVG uses vectors it is possible to zoom into any level without any distortion to the graphics.

By adding new methods to the graph plan class, node class and edge class seen in Figure 58 it is possible to convert a graph plan object into a dot specification file. This dot specification is passed to the GraphViz tool which generates the SVG representing the workflow of that plan.
Two different visualisations are possible from the graph plan object.

1. Visualise the time points
   Shows all the time points for a plan

2. Visualise the *happens* actions
   Shows all the *happens* actions for a plan

These visualisations can be used to view all the time points within all the plans or all the *happens* actions for all the plans. For the *happens* actions the time points are all removed. This ensures that actions that are identical bar the time point are displayed as a single node in the SVG graph.

In the following figure we see an example of the *happens* action visualisation.
Note that the current node in the plan is highlighted in purple. The goal node is also highlighted.

Figure 79: Visualisation of all the Happens actions in all of the plans in the hospital model
In the following figure we look at an example of the time points visualisation.
The time points visualisation provides interesting information about the plan generation process. Different time points occur when the Event Calculus planner has had to backtrack. Therefore by looking at the plans that share the same time points and where they diverge to different time points we can see the points that the planner backtracked to.

The Implementation

The solution was implemented in PHP. The header function in PHP is used to indicate to the user’s browser that the content is a SVG. To use the SVG function a browser plug-in needs to be installed.
7.3 Automatic Documentation Generation

*The Problem*

Due to having to stop work on the project for 1 month because of exams it was vital to provide detailed documentation in the code. Such commenting means the time taken to familiarise oneself with the project is small. It also means that people will find it easier to continue adapting and working on the project.

*The Design*

There is an open source tool called PHPdocumenter [21]. By using a specific format of comments throughout the code this tool can parse the entire code of a site and generate complete documentation for the code.

*The Implementation*

The commenting for PHP used this format. The documentation generated can be viewed at:

http://www.doc.ic.ac.uk/~jw99/iwfms/doc/
8 Testing

8.1 Overview

This section provides information on the testing strategy’s used while the project was in development and once the project was complete. Interesting bugs detected during the testing process are highlighted.

8.2 Unit Testing

8.2.1 PHP Unit Testing

The PHP Simple Test [34] tool was used for testing PHP scripts. This is a unit-testing framework for PHP which is based on the JUNIT framework for Java. Despite still being a beta version the tool proved to be very robust. Tests were written for all the complex object structures of the project and the core libraries as they were written. Extensive tests were written for the Prolog data structures as they contain some of the most complex object hierarchies in the system. These tests create objects, manipulate and perform operations over them and then using the functionality of Simple Test check certain assertions hold. This sort of testing is particular important in PHP since the use of loose typing means the compiler does not return type errors. Due to this functions have to be doubly robust to ensure they handle cases when the parameters passed do not represent the expected types. During testing a bug was locate in PHP. The problem related to large objects being side effected when passed as parameters. By default every parameter passed in PHP should be by value. This bug was passed on to the PHP development team who acknowledge the problem.

All the unit tests can be locate in the following location: http://www.doc.ic.ac.uk/~jw99/iwfms/tests/

8.2.2 Prolog Unit Testing

Before writing any complex Prolog a number of test cases where produced. This was particularly important since most of the Prolog scripts work as part of the Event Calculus planner. Due to this it is not possible to test them as isolated units on real examples without producing static tests. The tests tended to start with simple trivial examples and work towards realistic examples. The tracing feature in Prolog was used to ensure the correct path was followed through the tests. If any unusual bugs occurred further test cases where added. The Prolog unit tests can be found attached to the relevant Prolog code.
8.3 Prolog Event Calculus Plan Generation Testing

Since the planner is such an integral part of the iWFMS intensive testing was performed throughout its development. Testing involved ensuring the new versions of the Event Calculus planner matched plans generated before modifications. This involved the 2 plans initially published along side Shanahan’s Prolog planner [33]. These plans make use of the two main extensions provided by Shanahan, compound actions and state constraints.

- **Shopping Model**
  Model looking at shops in specific locations selling specific items

- **Plant Model**
  Model using state constraints. A boiler system dealing with pressure values.

The mail model provided by Shanahan was not used since it requires recursion.

As new advanced features where added to the planner and the workflow specification, test plan specifications were created. Since a prototyping design methodology was used for the workflow specification development these prototypes were used as test cases for the planner. The prototypes and test plan specifications were used to ensure further iterations of the planner still produced the expected plans for the advanced features.

Since the new version of the planner introduces multiple solutions while the previous planner only gives a single solution it was important to test that the plans generated where exactly those that were necessary. Any duplicates implied increasing overheads unnecessarily. The hospital model and a number of variations were used to generate plans. Then using the advanced visualisation feature added as an extension to the iWFMS all the plans were visualised at once in order to count the number of plans at each `happens` action. This number was check to ensure it reflected the different possible paths through the workflow.

8.4 Hospital Scenario Testing

These tests looked at ensuring that the hospital workflow specification was executed correctly by the iWFMS. Since I was unfairly biased with my knowledge of the iWFMS I gained the assistance of two colleagues and observed them as they tested the system.

They were given the task of running a number of complete cycles introducing a new patient who is prescribed controlled drugs. Each of the relevant groups involved in the system (doctor, receptionist, nurse and pharmacists) had to be simulated by logging in with their name and password. Three main paths have to be followed in order to check each of the decision points where the workflow could progress to new stages.

1. No mistakes are made by the doctor and the nurse accepts the controlled drugs.
2. The doctor makes a mistake which is reported back by the pharmacist.
3. The nurse rejects the controlled drugs.

To account for unexpected circumstances the testers where given the chance to also follow any path they wanted.

8.4.1 HTML Forms System

As the testers went through the cycles they were instructed to noted any exceptions they found in the forms. Also looking at ensuring the JavaScript constraints prevented invalid forms from being posted and allowed valid forms to be posted. This testing uncovered an interesting problem with constraining the order the user enters the form elements in. The JavaScript code responsible for this ordering is based on capturing the onClick event of the user clicking the mouse on the form element. In one of the tests the tester used the tab button to change the selected form element. Since this does not use the onClick event it meant the user could not enter the next form element even through they have filled in the previous element. This was solved by adding an onFocus event as well as the onClick event.

Finally they checked that the core index page showed the correct information and all links lead to the correct forms.

8.4.2 Database integrity

The database integrity was tested using the hospital model. It involved ensuring that the database of the iWFMS and the database used to store the form data produced by the iWFMS remained in an integral form. This involved examining the state of the databases before a form was submitted and after a form was submitted.

8.5 Integration Testing

Integration testing looked at making a change to the hospital workflow specification model and ensuring that the change was reflected in the iWFMS. Three different specific types of change to the specifications were examined. These were selected to reflect the likely changes a user might need to make during the lifetime of the workflow specification.

1. Reordering the occurrence of workflow units
2. Adding a new user group to the system
3. Changing the JavaScript constraints on a form

For each change complete cycles were performed of a patient entering the system through to the nurse checking the controlled drugs.
8.6 Performance Testing

Performance testing was broken up into tests on the core Event Calculus planner and the additional Prolog units added to the planner such as loop reduction and mapping to total ordered plans. The option to profile code in Sicstus Prolog was used in order to gain detailed information for each of the clauses in the program about the number of calls, the amount of backtracking, the number of deep failures and the execution time. This information was used to try and tweak the performance overheads wherever possible in the program. Further discussion of these results is given the evaluation.
9 Evaluation

9.1 Overview

This section provides a critical evaluation of the project. Two main sections are covered, quantitative evaluation and qualitative evaluation. The quantitative evaluation looks at the performance of the core elements of the iWFMS and the overall performance with all integrated parts. The qualitative part looks at how successful the workflow formalisation was for representing the hospital specification and how successfully the iWFMS framework is for running the specification. Finally, we look at comparisons with the project and the state of the art.

9.2 Quantitative Evaluation

9.2.1 Performance of Prolog Form Generation

I looked at the performance of Prolog functioning as CGI generating form elements against PHP generating the form elements. The performance time effects the time a user would have to spend waiting for a form page to load. Since the iWFMS is fundamentally a form based system this reflects an important performance point. The test for Prolog was performed by creating a simple workflow specification with a single form. The number of form elements where gradually increased within the form. The test for PHP was performed with a simple PHP script which outputted form elements with the number of elements gradually increasing. The timing functions within PHP were used to time both processes. For each case three results were taken and the mean was used to help compensate for small variations.
From the results we can see that there is a clear distinction between the Prolog CGI and PHP performance times. PHP is by far the faster solution. If the system requires the maximum performance time it should use PHP generation rather than Prolog CGI form generation as used in this project. However, we can also observe that Prolog CGI even with very complex forms manages to stay under the 1-second time barrier. Therefore, if time is not critical Prolog can function as a form generator responding to the user within a highly acceptable time span. The results also indicate that the iWFMS can handle extremely complex forms without substantially delays. The hospital model which represents a realistic form system has the maximum form size of five elements.

9.2.2 Comparison of the Event Calculus Planner and Extensions.

I examined how the overall Event Calculus planner’s performance was affected by the extensions added. These extensions consist of loop reduction, sorting the plans, mapping the plans to total-ordered plans and the HTML type checking. The evaluation was undertaken in order to see if the extensions had added any serious overheads to the planning process. Testing was achieved by using the profiling code feature of Sicstus Prolog.
We can see quite clearly from the results that the extensions add virtually no overhead to the planner. In addition, as the number of plans grow the extension’s performance overhead grows very slowly in comparison with the planner. We can safely say that the extensions can successfully function for a very large number of plans without serious performance consequences.

### 9.2.3 Comparison of the Extensions to the Event Calculus Planner

Within the extensions themselves I examined how each contributes to the performance overhead. Although we have seen the overhead as being small in comparison to the planner this does not mean that it does not add some level of performance overhead. Even if small it is important to evaluate if there are any areas which could have the performance improved upon.
From the results we can see that the decision to implement loop reduction with minimal performance overhead has paid off. This contributes the smallest amount of any overhead. The reason that the type system is such a large proportion of the overhead is because it is used for every form element within the workflow specification. As the number of plans increase the typing overhead grows very rapidly. The large majority of typing tests being performed for each plan are likely to be identical. This is because even through different plans follow different paths, the same forms are likely to appear in a number of the plans. This highlights a performance weakness of the typing system within this project. Although the performance overhead is not very large in comparison to the Event Calculus planner it still represents an area where perhaps the performance should be further investigated.

### 9.2.4 Prolog and PHP Performance

I examined the performance time of the plan generation, the time taken to parse the plans by the iWFMS into the abstract object structure and the time to store the plans in the database. All these performance times reflect the time the user would have to wait when invoking the creation of new workflow plans when workflow caching is not used. The test workflow specification used was the hospital model produced as part of the project. This represents a realistic model of workflow with conditional choice points and complete forms. The number of groups that can enter the initial form was increased by one with each test. Each increase results in a set of new plans.
for every possible plan path reflecting the new group. To increase the accuracy of the timing three results were taken for each test and the mean used.

The default timeout period in a web server of 30 seconds reflects quite well the maximum time that a user would want to spend waiting for workflow to be added. This means that the system would be able to generate a maximum of 60 plans. Considering the hospital model and the different variants of the model which represented real workflow models ranged from 5 plans to 19 plans 60 seems a realistic size to reflect moderately complex workflow. It does mean that the workflow specifications may not be able to represent highly complex applications unless the users are willing to wait longer than 30 seconds or the workflow caching feature is used. With workflow caching the plans are generated once and cached so everyone else uses the cached version rather than repeating the planning.

From the results we can see that the main overhead of the workflow creation process lies with the Prolog plan generation. Although the overhead is the largest it still manages to keep under the 30 second time barrier when generating 80 plans. Although this could perhaps be further optimised it indicates the really unavoidable overhead of using a Prolog planner. It also indicates the fact that although conditional planning is effective for representing moderately complex workflow models the increase in the number of plans with highly complex models would mean that the real time addition of the workflow would be too time consuming. This does not rule out complex workflow models though, since the facility to cache workflow provided by the project would mean that such workflow models could still be achieved. Without caching it is unrealistic for runtime plan generation.

Another point highlighted by the results is the unsuitability for specifications with very large number of groups which all can perform the same task for real time plan
9. EVALUATION

As we can see just by increasing the different groups that can access the same form we double the number of plans. Again this highlights the need for the caching feature provided by the project.

The choice of converting the Prolog plans into a complex directed graph object structure has brought about the second largest overhead. This reflects a performance disadvantage which is nearly as substantial as the Prolog planner. The offshoot being that once the workflow has entered the system the performance cost of executing the workflow is reduced due to the directed graph object structure.

9.3 Qualitative Evaluation

The project has covered two main areas, how the workflow is specified in the Event Calculus and how it is used to guide the execution of the system. In this section we look at the hospital model (seen in Figure 11: UML attribute model of hospital medication process) from the start of writing the specifications through to the completion of the workflow item in the iWFMS to help evaluate the project as a whole. Where ever possible diagrams will be used over showing the workflow specification and we shall abstract from the Meta language the specification is contained within in order to increase the clarity of the evaluation.

9.3.1 Event Calculus Specifications

9.3.1.1 High Level Specification

The first step in modelling the workflow of the hospital is to specify the high level specification. This relates to the directed graph level of the specification.

It is possible to set down the structure of the plans without having any knowledge of what will be contained in the form elements. This follows very naturally with the nature of designing specifications, starting from the highest level of abstraction. In the following figure, we look at an example from the hospital to illustrate these strengths.

![Figure 86: An extract of the directed graph level specification for the hospital model](image)
This shows that at the directed graph level, we do not have to worry about the contents of the nodes/workflow units, we can just view them as drug authorise and drug success. We also are unaware of the actual form element values and edge progression constraints required to progress to drug success.

It is also possible to use the Event Calculus planner and the high level specification to generate plans at the highest level without the lower levels existing. This allows the user to be sure that they have correctly modelled the form system by looking at the plans generated. The level of abstraction also means that workflow units can be reused very easily without having to duplicate the actions contained within the workflow units. The high level compound actions can be seen as being like procedures calls.

There is however a difficulty which complicates the reuse of workflow units in certain instances. In the following figure, we look at part of two high-level specifications paths within the hospital model.

![Figure 87: Two specification paths at the Directed graph level for the hospital model](image)

This figure shows an example where we want to reuse the drug success workflow unit. The problem arises due to the preconditions that this drug success workflow unit has. The follow figure highlights the precondition.

```prolog
notOccurred(edgeProgression(_, formElement(approvePharmacyChanges, match, false ))), _)
```

![Figure 88: Drug success workflow unit](image)

This pre-condition states that this workflow unit should not occur if drug failure has occurred. This is fine for the first specification path but in the second specification path we find that we want drug success to occur and drug failure has occurred in the
past! This means that we cannot reuse exactly the same workflow unit, but need a new workflow unit which does not have this pre-condition.

This is a disadvantage to the high level specification as we cannot intuitively re-use the same drug success workflow unit at the directed graph level of the hierarchy. This is not a massive disadvantage since reuse is still possible at the level below the directed graph. Looking at Figure 90, we can see how it calls the same compound actions as Figure 89, sharing the same low level details.

The problem occurs due to pre-conditions having to occur in the workflow unit. They cannot occur in the high level specification level since the pre-condition is a constraint based on a low level action. This will not be detectable in the plan until we have reduced the higher level to the lower level.

At the directed graph level the loop cases have to be identified for the hospital model. The defining of loop cases has its good and bad points. It is not intuitive to state that the point that loops is going to achieve the goal. It is really going to do the exact opposite and not achieve the goal by looping. However, since the specification stops at the loop point we do not to have to specify a continuing plan for the loop case after the looping point which is intuitive with how a loop should work.

9.3.1.2 Role Definitions

Roles and groups have to be entered for the hospital model specification. These reflect the different groups in the hospital domain.
9.3.1.3 Workflow Unit Specification

The next step after producing the directed graph level and roles is to provide the contents of the workflow unit. It is up to the user to correctly specify the actions within the workflow unit.

`formEntry, createFormElement, formSubmission` and `edgeProgression` are the mandatory actions.

This has the problem that the user could generate a plan where a workflow unit does not have the mandatory elements. The user would have to wait until the point where the incomplete workflow unit was encountered before being made aware of the error by the iWFMS.

This is not really a problem that needs to be overcome by the specification. What it highlights is the need for a visual editor to the Event Calculus workflow specification. This visual editor could then enforce that the user always created valid workflow units.

9.3.1.4 Form Creation Specification

The next stage in the hospital model is to specify all the forms required in the system.

The form creation specification provides the very strong advantage of enabling the user to customisable a form element in any way physically possible under current HTML standards. This increases the portability of the specification since it can ensure the form style and functionality match with what a particular group of users in a specific domain would expect.

In addition, the form creation specification allows all the main HTML form elements to be used. In the following figure, we can see within the hospital model an example of the varied HTML form elements and how they have been customised.

![Figure 91: iWFMS displaying a HTML form for the hospital model](image)

An interesting point in the form creation is that we are completely abstracting from the HTML. The user does not have to produce any HTML in order to create form
elements. They provide the attributes and required values. For example, the user does not have to know the HTML format of a radio button which is:

```html
<input type="radio" name="example" value="false">
```

All they enter is:

```prolog
happens(createFormElement(radio, "name", example, 'false', [], T2)
```

This will make it easier for anyone to use the specification. Abstraction from HTML is very attractive since it separates specification from the details of the syntax. This means different syntax layers could be produced for formats other than HTML. A hard coded HTML form system would not be able to achieve this level of HTML abstraction or support other syntaxes with such ease.

A limitation of the form creation specification is that it does not support the rather unusual elements that can occur in forms. For example such functions as Flash or Shockwave (used for interactive visualisations). Although these functions are rare it would improve the portability of the system to support them.

### 9.3.1.5 HTML Typing

While creating the forms for the hospital specification we have to add checks that ensure the form element specification is correct. This makes it easier to ensure that the HTML forms do not have any unexpected effects due to incorrect specification.

When HTML typing fails no plans are generated and an error message is returned to the user indicating which specific attribute in the form element specification was invalid. This helps save a large amount of time detecting and correcting errors. We can look at an example of a typing error message:

**ERROR: HTML Typing is incorrect for attribute:size  value:'letters’**

Here a size attribute has been assign a string when it should only have numbers.

A problem with the typing system highlighted in this example is that it does not explain why the attribute is in error. We are not told size must be assigned a number but rather that there is just a type error. This is particularly difficult to achieve since the type rules are being dynamically created. It is difficult to also dynamically create messages why typing failed.

Another problem with the typing error message is that it does not tell you in which file and at which line the error occurred at. This would be a very useful feature but I suspect due to the complexity of knowing file names and line numbers in Prolog it would be impossible.

A problem with the typing system is that it relies upon the user entering the correct typing axiom in the specifications. They can completely bypass the typing system by
not adding these axioms. Ideally these axioms need to be automatically generated rather than being left to the user. This again highlights a problem not with the specification as such but the need for a visual editor to the workflow specifications that generates these typing rules enforcing that they always appear.

9.3.1.6 JavaScript Control

JavaScript controls have to be added to the required forms in the hospital model. This specially relates to the Assess patient HTML form in the hospital model.

A nice feature of the JavaScript controls is that they use rules about the domain which have no reference to JavaScript or form elements at any level. The specification in the hospital model indicates information about how children only go to the child ward, while adults only go to adult ward. This highlights a strength in the specification in that generic rules about the domain can be used to dynamically change the value that a happens action takes.

Within the JavaScript control all the basic functions that would be required to constrain a form system are available. It also does not require knowledge of any JavaScript to use! The user does not have to worry about the creation of intricate and time consuming JavaScript code. It is all generated automatically from the JavaScript control specification. This sort of abstraction would not be possible in a hard coded HTML form system. This is a powerful feature but by providing an abstraction from the JavaScript we introduce limitations.

The limitation with the JavaScript controls lies with its inability to provide the same functionality found in hand written JavaScript. By creating a generic framework for JavaScript we are abstracting away from the JavaScript and in doing so we loose the power to write complex JavaScript. An example of complex JavaScript could be a form system where the selection of a select box changes the form elements that are visible in the form. This ultimately limits the complexity of the constraining for the benefit of allowing anyone to use the specification.

9.3.1.7 Edge Progression

The edgeProgression holds_at conditions have to be added to the relevant workflow units for the hospital model. Then actions have to be added which initiate the edgeProgression.

Within these edgeProgression actions it is possible to specify a wide range of different comparison conditions. The support of relative expressions in the edgeProgression proves to be a powerful feature which allows us to check the type and format of form element values. All the comparison options cover those that are required by the hospital model. The specification also makes it possible to express multiple edge progression actions indicating that multiple conditions are required in order to progress to a new workflow unit.
A limitation of the edgeProgression actions is their inability to deal with comparisons where some degree of processing is required beforehand. For example perhaps the form input from the user has to be processed, parsed and produced into some format before it can be used in some comparison function. Such functionality would allow the iWFMS to be applied to more complex applications.

The edgeProgressions have to be defined for the looping cases in the hospital model. There is nothing restricting the user from looping to any point in the plan. Within the specification loops should only every loop to the start of a workflow unit. This constraint would be something that could be handled with an editor to the workflow specifications.

9.3.1.8 General Specification Points

By placing a group permission axiom which uses the role axioms at any point in the specification the user can control exactly how the workflow is generated based on the user generating it. This allows a high level of customisation which helps the specification meet the complex demands of real world models. If the caching feature is used this feature is limited. This is because only one person generates the workflow plans and everyone else uses this cached version.

We have not mentioned the temporal orderings while going through the specification. These have to occur for nearly every action within the specification. Having to enter the temporal orderings is tedious and time-consuming. Since the plans are total ordered it is not intuitive for the user to have to put the before predicates into the plans. This highlights that the before predicates should be abstracted away from through some visual editor to the Event Calculus specifications.

A problem that hampers the ease of hand writing the specification lies with the reliance upon name matches. This can mean that if there is a discrepancy with the names in multiple files no plan will be generated. No error message is given and the user has no idea why the plan was not generated. This problem was frequently encountered in creating the hospital specification. This again could be solved through some visual editor to the specifications.

9.3.2 iWFMS Evaluation

Finally once the hospital specification is complete it is selected in the iWFMS and used to generate all the plans for a workflow item. The ability to choose which workflow specification you want to use means that the system can have multiple specifications all active within the one system. Therefore, the same iWFMS could be used to run multiple form systems. Once added the workflow item appears in the core index page.
Through the XML specifications it is possible for the data posted in the forms to be directed to a number of databases distributed in different locations. This is a very useful function as in most environments there is never one big database.

When a form is submitted without meeting the order dictated by the JavaScript ordering constraints an error message appears telling the user exactly which element they need to fill in. Error messages are also given for JavaScript errors relating to constraints (i.e. age<18 && ward==children). The JavaScript planner gives an English error message by converting the JavaScript constraints to English. This however has its limitations. The following figure looks at an error message generated by the hospital specification when a child’s age is entered and they have been assigned to an adult ward.

![JavaScript error message](image1.png)

Although this provides a reasonable attempt at an error message it is not as good as what could be achieved with a hard coded HTML form system. The iWFMS is constrained in the natural language that can be used in-order to function generically from JavaScript constraints. Ideally what the user would want to see is an error message of the form:

‘You have assigned a child patient to a ward which is only for adults.’

### 9.3.3 Handling Change

The whole reason a workflow specification language is used is in-order to extract the workflow from the actual system meaning changes can be made easily. The iWFMS must also handle change gracefully to maximise its adaptability to evolve as the real world does.
9. EVALUATION

- The high-level abstraction levels in the workflow specification make it very easy to alter the order of the workflow units without having to worry about their details.

- The use of XML specifications means that database fetching and the destination of the data posted in forms can be changed without having to alter the workflow specifications. These XML files have to be changed by hand which can be a fiddly task with the bigger XML files.

- Changing workflow specifications means that all the workflow items already in the system will still be using the old workflow specification. This is important since the old workflow items may not be compatible with the new workflow specifications.

- The HTML typing system can handle new HTML specifications from the W3C standards organisation allowing the system to always maintain the latest standards.

- The groups and roles specified in the workflow specification can be changed or added to without having to make any other changes to the workflow specification.

- When changing the number of elements within a compound action in the workflow specification it is often confusing to have to remember to change the time points in the head of the compound action i.e.

  \[
  \text{happens}(\text{formCreation}(\text{failedDrugsApproval}, \text{User}), \ T_1, T_2):=\ 
  \text{valid\_html\_form}(\ \text{input}, [\text{type}=\text{radio}]),\ 
  \text{happens}(\text{createFormElement}( \text{radio}, \ "\text{Approve Pharmacy Changes}\),\ 
  \text{approvePharmacyChanges}, \ 'true', [], T_1),\ 
  \text{valid\_html\_form}(\ \text{input}, [\text{type}=\text{radio}]),\ 
  \text{happens}(\text{createFormElement}( \text{radio}, \ "\text{Reject Pharmacy Changes}\),\ 
  \text{approvePharmacyChanges}, \ 'false', [], T_2),\ 
  \text{before}(T_1, T_2) .\]

  Figure 94: Compound action with two time points

  For every new action added T2 in the head of the compound action has to be changed to a new time point.

  - If changes introduce errors, no plans are generated and the user is not informed why. This leads to a frustrating process of having to workout what is incorrect in the specification. It would be useful if the Event Calculus planner could capture the reason why the specification failed.

9.3.4 Overall Evaluation
This section of the evaluation compares the achievements of this project with the state of the art.

The PiLLoW library form generation
Within the PiLLOW [24] library form generation from Prolog is seen as expressing the Prolog predicates that are used to generate forms in the simplest and most concise way. The resulting HTML form elements generated contain the minimal attributes. This project has contributed a new angle to the PiLLOW library which focuses on allowing the user to specify in a high level of detail exactly how they want the form element to appear. This increased flexibility opens the doors to new applications for the PiLLOW library. The disadvantage this has over the PiLLOW library is the need to perform a degree of HTML typing to ensure the form element does not have unexpected behaviour due to incorrect specifications.

Prolog as CGI
Prolog’s appearance as CGI has so far in the current state of the art been limited to form generation. This project takes what Prolog is good at and uses it within CGI rather than the PiLLOW project which has tried to take Prolog and shape it into what the server-side languages are good at, form generation. The problem this project has faced and the main reason probably why currently Prolog is not used as CGI is the performance overhead. Even within the caching solution used in this project the underlying suitability for Prolog as CGI is only in those situations where the overhead delay is acceptable to the user.

Interaction support between PHP and Prolog.
The PiLLOW library saw Prolog acting on its own as CGI. This project takes Prolog and supplements it with PHP which can help provide the functionality that Prolog lacks. The project has produced one of the first interaction layers that allows PHP and Prolog to interact. This brings Prolog a step closer to being adopted more frequently as CGI. The project produced a fully expressive model but due to time restrictions the parser was only geared towards predicates. With a small tweak to the parser the PHP code would be ready to be used to handle all Prolog. The real disadvantage of the model developed over the PiLLOW vision of Prolog functioning on its own is the overhead of converting between the two languages.

Event Calculus as a workflow representation
The Event Calculus has been applied to a number of domains and this project has expanded this to include the as yet untapped domain of workflow. The Event Calculus was not expressive enough in its basic form to express workflow as it had to use a number of extensions.

Event Calculus conditional planner
There appear to only be a few Event Calculus planners that have been implemented and are openly available. This project has taken Murray Shanahan’s planner and helped contribute a conditional Event Calculus planner to the planning community.

iWFMS compared with state of the art WFMSs.
The disadvantage the iWFMS developed has over the state of the art in WFMSs is the performance overhead of the real time plan generation, although the workflow caching function does help minimise this overhead. In addition, most WFMS support
the use of partial order specifications while the produced iWFMS only supports total ordered plans. The use of partial order plans was not a goal within this project but there is no reason why the specification could not be altered to support it.

A limitation of the iWFMS developed is that there is no graphical interface to edit the workflow specifications. This was not a goal of the project but is something that needs to be done in order to bring the iWFMS into a state where it can really compete with WFMSs.

WFMS workflow specifications can be written in XML implying interoperability. By using a logic based formalisation the ease of interoperability is not achieved. However, it would be possible to write a high level XML specification which could be converted to the logic specification or have an editor to the workflow specifications which could export them as XML.

The advantage the iWFMS developed has is the conciseness in which the workflow can be specified, leaving the planner to work out the details. WFMS specifications in contrast tend to be overly verbose. The iWFMS is also more effectively orientated towards change than current WFMS. This is through the abstraction layers in the specifications and the use of XML throughout the system. The ability to use domain rules to bind dynamically the actions that occur in happens predicates is a feature which is not possible under conventional WFMS. This feature clearly distinguishes the iWFMS produced in this project from current WFMS. It is a clear example of how AI planning has been used to overcome the problem of handling change with greater effectiveness. The user can change the domain rules without having to even consider the implications in the rest of the workflow specification.
10 Conclusion

I achieved most of the goals I initially set myself. The goal I did not completely achieve was the complete specification of the hospital model, certain stages were ignored. This however does not detract from the overall quality of the project since the stages abstracted from were just examples that had already been dealt with in the workflow specification.

The project has managed to produce a fully functional iWFMS and a complete workflow specification language based on the Event Calculus. The workflow specification language provides powerful levels of abstraction meaning anyone could use the specifications without having to know or write any HTML or JavaScript. This does come at the cost of limiting the more advanced features that can be achieved by hand written JavaScript. Through the workflow specification all the basic functionality required to describe workflow and HTML form systems is provided. A HTML type system which can always be up to date helps overcome the dangers of unexpected effects in forms through bad form element specification. The specification and framework were developed to be expressive enough to model the hospital model. Investigation and implementation of plan execution frameworks in both server-side environments and client-side environments was completed successfully. New foundations in using Prolog as a CGI based language were introduced and a new framework for PHP and Prolog to interact was developed and is ready for the PHP community to use. An invaluable tool for anyone using the Event Calculus was produced allowing the graphical visualisation of plans. As well as producing a functional system issues in modelling workflow and server/client interactions where introduced and overcome. The workflow specification language is generic enough to be applied to new domains with small alterations.

The iWFMS developed in this project handles change better than a WFMS. It also allows some advanced features such as creating different workflow based on the user invoking creation which are not seen in conventional WFMS. It does this while sharing workflow so a change can be made for all users in one place. The whole iWFMS has been geared for change which it achieved very successfully; it however does this at a cost. The limiting factor in the iWFMS is that of performance. A more abstract and flexible workflow specification is possible using such features as hierarchical planning and domain rules but the planner has to do the work of converting these specifications into the plans. Through caching this planner overhead can be reduced but it still has to occur at least once. Therefore, in performance critical applications a WFM is more suitable. If on the other hand a small delay or the caching solution is acceptable the iWFMS should always be used over a WFMS. Performance issues are the underlying next challenge for iWFMSs.

In the current state the workflow specifications have to be written by hand which can be frustrating. The need for an application that provides an interface to the workflow specifications is essential in order to complete the iWFMS tool given in this project.

A number of further conclusions have been highlighted by the work in this project and are summarised below:
Facilitating planning at the high abstraction level and the low abstraction level within the Event Calculus is performance intensive due to the tangling of dependencies between the two levels. This seriously limits the ability to perform re-planning within complex Event Calculus specifications.

The basic Event Calculus on its own is not sufficient to express workflow. It needs further extensions in order to provide an effective workflow specification language.

Looping is an impossible problem at the planner level in Event Calculus. The planner cannot ever represent exactly what the user can actually achieve. They can loop infinitely, the planner cannot. It is possible to produce a fixed number of loops in the planner. The problem with this is that the plans produced need to be converted to a representation that truly reflects infinite looping. This is not possible since we cannot tell in the plans if there is a loop or the user just wants to repeat a task a fixed number of times. A degree of processing is required in order to introduce loops into the Event Calculus specification above the level of the planner.

Conditional planning is effective but has an associated performance load due to the number of plans generated.

PHP’s loose typing provides interesting adaptations of design patterns. Super classes used to allow uniformed handling of subclass are not always needed due to PHP’s ability to already handle the subclasses in a uniformed way through loose typing.

Prolog has a place as CGI, helping supplement server-side languages rather than replacing them. Prolog should be used for what it is good at, not trying to simulate the behaviour of what server-side languages are already good at. In using Prolog CGI there is however a small performance cost which means time critical applications are not suited to using Prolog CGI.

10.1 Further Extensions

There are a number of further extensions that are possible for the project.

1. GUI interface to the Event Calculus specifications
From evaluating the workflow specification language it is clear that they need some form of editor application. This would take the Event Calculus specifications as the underlying syntax but present a visual editor abstracting from the specifications to the user. Since the workflow specification has different levels of abstraction these could be transformed into different visualisation levels for the user. The editor would also have to ensure a number of constraints are maintained in order to ensure that only valid specifications are generated. This would most likely require some form of parser
that could validate the Event Calculus specifications. Such a visual editor would complete the iWFMS system.

2. Adding support for JavaScript and PHP pre-processing scripts before checking conditions
This would involve linking into the workflow specification an action which linked to scripts. These scripts would be used to take the input from the user and perform certain language specific transforms and finally return a value. These scripts would then run before making any comparisons. This would allow more complex data processing such as parsing input data into a specific format.

3. Support for advanced HTML form elements
To introduce Flash and Shockwave elements into the HTML forms. These are interactive plug-ins that can be used within a webpage. The workflow specification would have to be extended to support the complex information required to specify such elements. The challenge here would be the automatic generation of Flash and Shockwave from the plan. For Flash this could be achieved by using a tool called PHP Turbine 7 [23]. This tool allows the generation of Flash through PHP scripts and potentially XML documents, databases, video and images. There would also have to be a planner implemented in Flash/Shockwave in a similar way to the JavaScript planner. This would constrain the user within the Flash/Shockwave. There would have to be some mechanism for the JavaScript planner and the Flash/Shockwave planner to interact. This is required since one form element may have to completed before the Flash/Shockwave element could be used. This could be achieved at the worst through temporary files as both planners have access to the file system. More ideally this would be performed through the browser document object which all elements within a webpage should have access to.

4. Support for Partial ordered plans
The system only supports the generation of total order plans. An interesting and challenging further avenue is to adopt the framework to support partial ordered plans. It would be necessary to remember multiple current points in the plan. This is required since multiple users could be simultaneously at different points within the plan. The workflow specification would need the introduction of constraints based on the progress of other users. Supporting OR, XOR and ANDs. For example if one user has done x OR another use has done y then progress to the next workflow unit. The edge progression action could be refined to add this extra functionality. The PHP planner could also be adapted to support these new conditions.
11 Bibliography

[1] AT&T Labs-Research Graphviz Project


http://www.doc.ic.ac.uk/~kjc/


[6] Philippa Gardner’s Lecture course on Models of Concurrent Computation
http://www.doc.ic.ac.uk/~pg/

[7] Imperial College E-Science Networked Infrastructure (ICENI)
http://www.lesc.ic.ac.uk/iceni/iceni_architecture.html


http://www.coli.uni-sb.de/~kris/prolog-course/html/node90.html

[13] Frank Kriwaczek’s Lecture course on The Practice of Logic Programming


[20] K.L.Myers and P.M.Berry, At the Boundary of Workflow and AI In AAAI-99 Workshop on Agent based systems in the business context, Orlando, USA, 1999


[22] PHP Extension and Application Repository (PEAR) http://pear.php.net/


Supporting Dynamic Workflow Modification,
Proceedings of the workshop on Software Architectures for Business Process
Management at CAiSE*99, The 11th Conference on Advanced Information

[28] M.P.Shanahan, An Abductive Event Calculus Planner,

[29] M.P.Shanahan, The Ramification Problem in the Event Calculus,
Proceedeeings IJCAI 99, pages 140-146.

[30] M.P.Shanahan, The Event Calculus explained,
Artificial Intelligence Today, ed. M.J.Wooldridge and M.Veloso, Springer
Lecture Notes in Artificial Intelligence no. 1600, Springer (1999), pages 409-
430.

[31] M.P.Shanahan, Solving the Frame Problem,

[32] M.P.Shanahan, Prediction Is Deduction but Explanation Is Abduction,
Proceedings IJCAI 89, pages 1055-1060.

[33] M.P.Shanahan, Abductive Event Calculus Planners
http://casbah.ee.ic.ac.uk/%7Empsha/planners.html

[34] SimpleTest/PHPUnit Homepage
http://www.lastcraft.com/simple_test.php

[35] Francesca Toni’s Lecture course on Advanced Issues in AI
http://www.doc.ic.ac.uk/~ft/

[36] W3C Scalable Vector Graphics (SVG) 1.1 Specification
http://www.w3.org/TR/SVG/

[37] Yi Wang’s Research presentation on ‘Workflow Description Language and
Workflow Patterns’
http://www.es.ucf.edu/~dcm/Fall02Class/ResearchPresentations/Yi%20Wan
g.ppt

[38] P.Yolum and M.P.Singh. Reasoning About Commitments in the Event
Annals of Mathematics and Artificial Intelligence (AMAI), Special Issue on
12 Appendix

12.1 Source Code

12.1.1 Obtaining the Source Code

All code for the project is located at:

http://www.doc.ic.ac.uk/~jw99/iwfms/doc/

All PHP code can be seen through the on-line documentation:

http://www.doc.ic.ac.uk/~jw99/iwfms/cgi-bin/

All Prolog code is located within:

http://www.doc.ic.ac.uk/~jw99/iwfms/cgi-bin/

All Event Calculus models are located within:

http://www.doc.ic.ac.uk/~jw99/iwfms/specifications/workflowModels

12.1.2 Dependencies

The iWFMS is dependent upon:

- Any web server with can support PHP
- Database server.
- Any database server which is supported in PEAR. The configuration file within the source code has to be set to indicate which server you are using. This file is located in:

  http://www.doc.ic.ac.uk/~jw99/iWFMS/includes/configuration/database.php

- PHP 4.3.7 or higher
- PEAR 1.69 or higher
- PHP Simple Test 1.0 beta 6 or higher
- Sicstus Prolog 3.11.10
- Berkeley Database 4.2.52 or higher