

**COMPUTER-INTERPRETABLE GUIDELINES USING GLIF WITH WINDOWS
WORKFLOW FOUNDATION**

by

Ryan Minor

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science (MSc) in Computational Science

The School of Graduate Studies
Laurentian University
Sudbury, Ontario, Canada

(c) Ryan Minor, 2014

THESIS DEFENCE COMMITTEE/COMITÉ DE SOUTENANCE DE THÈSE

Laurentian University/Université Laurentienne School of Graduate Studies/École des études supérieures

Title of Thesis Titre de la thèse	COMPUTER IMPLEMENTABLE GUIDELINES USING GLIF WITH WINDOWS WORKFLOW FOUNDATION		
Name of Candidate Nom du candidat	Minor, Ryan		
Degree Diplôme	Master of Science		
Department/Program Département/Programme	Computational Sciences	Date of Defence Date de la soutenance	June 30, 2014

APPROVED/APPROUVÉ

Thesis Examiners/Examineurs de thèse:

Dr. Waldemar W. Koczkodaj
(Supervisor/Directeur de thèse)

Dr. Ryszard Janicki
(Committee member/Membre du comité)

Dr. Raymond G. Hendel
(Committee member/Membre du comité)

Dr. Andrzej Grzybowski
(External Examiner/Examineur externe)

Approved for the School of Graduate Studies
Approuvé pour l'École des études supérieures
Dr. David Lesbarrères
M. David Lesbarrères
Director, School of Graduate Studies
Directeur, École des études supérieures

ACCESSIBILITY CLAUSE AND PERMISSION TO USE

I, **Ryan Minor**, hereby grant to Laurentian University and/or its agents the non-exclusive license to archive and make accessible my thesis, dissertation, or project report in whole or in part in all forms of media, now or for the duration of my copyright ownership. I retain all other ownership rights to the copyright of the thesis, dissertation or project report. I also reserve the right to use in future works (such as articles or books) all or part of this thesis, dissertation, or project report. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that this copy is being made available in this form by the authority of the copyright owner solely for the purpose of private study and research and may not be copied or reproduced except as permitted by the copyright laws without written authority from the copyright owner.

ABSTRACT

**COMPUTER-INTERPRETABLE GUIDELINES USING GLIF WITH WINDOWS
WORKFLOW FOUNDATION**

Ryan Minor

Masters in Computational Science

Laurentian University

2014

Modern medicine is increasingly using evidence based medicine (EBM). EBM has become an integral part of medical training and ultimately on practice. Davis et al. [6] describe the “clinical care gap” where actual day-to-day clinical practice differs from EBC, leading to poor outcomes.

This thesis researches the GLIF specification and implements the foundation for a GLIF based guideline system using Windows Workflow Foundation 4.0. There exists no public domain computer implementable guideline system. The guideline system developed allows a guideline implementer to create a guideline visually using certain medical related tasks, and to test and debug them before implementation.

Chapter 5 of this thesis shows how to implement a guideline called *Group A Streptococcal Disease Surveillance Protocol for Ontario Hospitals* which is of fundamental importance for Ontario hospitals.

The workflow approach allows developers to create custom tasks should the need arise. The Workflow Foundation provides a powerful set of base classes to implement clinical guidelines.

Contents

1	Introduction	1
1.1	Introduction	1
1.1.1	Motivation for this Thesis	1
2	Computer-Interpretable Guidelines (CIGs)	3
2.1	What is Evidence-Based Medicine?	3
2.2	From Text Guidelines to Computer-Interpretable Guidelines	3
2.2.1	Why are Guidelines Needed?	3
2.2.2	The Development of Clinical Guidelines	4
2.2.3	From Text to Computer-Interpretable Guidelines	5
2.2.4	Anatomy of a CIG	5
2.2.5	Electronic Medical Records	6
2.3	Modeling Clinical Guidelines	7
2.3.1	Introduction	7
2.3.2	Arden syntax	7
2.3.3	Task-Network Models	10
2.4	Implementations of Computer Clinical Decision Support	11
2.4.1	The Unified Medical Language System (UMLS)	11
2.4.2	HL7 and the Reference Information Model (RIM)	13
2.5	Comparing Computer-Interpretable Guideline Models	14
3	Guideline Interchange Format (GLIF)	16
3.1	The Three Layers of Abstraction	18
3.2	Major Classes	18
3.2.1	Guideline Class	18
3.2.2	Steps	18
3.2.3	Action Steps	21
3.2.4	Branch and Synchronization Steps	21

3.2.5	Three Value Criteria	22
3.3	Authoring using Protege	22
3.4	Expression Languages	22
4	Implementing GLIF in C# .NET	23
4.1	Introduction to .NET Framework	25
4.2	Windows Workflow Foundation	26
4.3	Workflow Modeling Styles	27
4.4	Activities	27
4.4.1	Creating a Custom Activity	28
4.5	The Flowchart Style	29
4.5.1	FlowStep	29
4.5.2	FlowSwitch<>	29
4.5.3	FlowDecision Activity	29
4.6	Workflow Services	30
4.7	XAML Format	30
4.8	Implementation of GLIF3.4 in WF	30
4.8.1	Implementing the Host	31
4.8.2	Host-Workflow Communication	31
4.9	Evaluating Expressions	34
4.10	Designing GLIF Activities	34
4.10.1	The BaseActivity Class	34
4.10.2	Implementing PatientStateStep	35
4.10.3	Implementing ActionStep	35
4.10.4	DecisionStep	38
4.10.5	Branch and Synchronization Steps	41
4.10.6	Parsing RDF	42
5	Authoring a Guideline in Workflow Foundation	44
5.1	The Guideline	44
5.1.1	Subguideline - Treatment of GAS	45
5.1.2	Creating the Subguideline	46
5.1.3	Implementing the Main Guideline	49
5.1.4	Testing the Guideline	53
6	Conclusion	55

CONTENTS

vi

A BaseActivity Source Code

59

B PatientStateStep Source Code

62

C ActionStep Source Code

65

List of Figures

2.1	Guideline System Diagram	6
2.2	Arden Syntax	9
2.3	UMLS Metathesaurus	12
3.1	Cough Guideline	17
4.1	Integrated Development Environment (IDE)	24
4.2	Major Assemblies	25
4.3	CodeActivity sample	28
4.4	Workflow Services	30
4.5	IHostMessaging	31
4.6	IHostDataRequest	32
4.7	Event Handlers	32
4.8	Instance Variables of BaseActivity	33
4.9	Rules Engine Calculator	35
4.10	PatientStateStep	36
4.11	ActionStep	36
4.12	Task Diagram	37
4.13	Find Flowchart Reference	40
4.14	Deleting Temporary Variables	40
4.15	Setting Temporary Variables	41
4.16	Parsing Stages	43
5.1	Decision Step - Is HWC Pregnant or Lactating?	46
5.2	Medical Action Task - Prescribe Cephalexin	47
5.3	Adding an OutArgument	48
5.4	Treatment of GAS Subguideline	48
5.5	Adding a Subguideline	49
5.6	Matching Guideline Variables with Subguideline Variables	50

5.7 PatientStateStep for treated = true	51
5.8 MessageTask - Suspend HCW for 24 Hours	51
5.9 Full Main Guideline	52
5.10 Query if Patient is Pregnant or Lactating	53
5.11 MedicallyOrientedActionStep - Prescribe Cephalexin	54
5.12 Suspend Health Care Worker for 24 Hours	54

Chapter 1

Introduction

1.1 Introduction

Empirical research suggests that health care is not as safe as it should be[14]. A study in Canada has shown an alarming number of unintended injuries or complications resulting in death, disability or prolonged hospital stays arising from health care management[4].

One reason is that decision making in modern medical practice is based on increasingly complex medical knowledge and clinical evidence, making it difficult to provide the best possible care in a busy healthcare setting [22]. Clinical practice guidelines using evidence-based medicine have been developed to deal with widespread variations in clinical practice and also soaring health care costs[24].

The usefulness of clinical guidelines on medical practice has been demonstrated in several studies [3]. Clinical guidelines have been gaining increasing acceptance in medicine to support diagnosis and treatment.

There is some impetus for computerized clinical guidelines. The report of the U.S. Institute of Medicine - *To Err is Human* - recommended that health care organizations, individually and in collaboration, commit to using information technology to manage their knowledge bases and processes of care, and manage the case process through reminder, decision support, and guidance grounded in evidence-based knowledge[14].

1.1.1 Motivation for this Thesis

Limited work has been done by academics in the area of computerized clinical decision support. In recent years, researchers have developed computer-interpretable guideline frameworks to represent clinical guidelines. Unfortunately, execution engines based on

these frameworks are proprietary and cannot easily be extended by other researchers.

The major goal of this thesis is to provide an accessible execution engine for computer-interpretable guidelines to the research community for further development. It is hoped that this engine will serve as the foundation for a guideline system to:

- educate students of medicine by allowing them to follow a clinical guideline and access useful information at the various stages in the guideline (e.g. why is the guideline asking this question?);
- assist personnel at medical facilities to follow proper medical protocols prescribed by administration or government; and
- provide assistance to physicians at the point of care when dealing with complex cases.

In this thesis project, a computer-interpretable guideline engine based on the Guideline Interchange Format (GLIF) is developed. GLIF is a format for representing clinical knowledge conceptually.

GLIF was selected because the framework was developed by leading researchers in the field of medical informatics from institutions including Harvard, Stanford and Columbia universities. It also incorporates successful features of other computer guideline models [18] and does not presuppose that an institution stores data in any particular format. A major challenge of implementing computer-interpretable guidelines (CIGs) is integration with an actual electronic medical record(EMR). A CIG based on GLIF can be exported to other hospitals and only the EMR interface would need to be built.

New programming languages and libraries are available to implement clinical decision support. The execution engine developed is based on Microsoft C# .NET Windows Workflow Foundation (WF). WF is part of the .NET framework and is bundled with recent versions of the Windows operating system and can be downloaded free.

In section 2 of this thesis, existing work on computer-interpretable guideline systems is discussed. In section 3, the GLIF specification is discussed in detail. Section 4 describes the work done to create an execution engine based on GLIF specification using Windows Workflow Foundation. In section 5, a guideline is implemented using the execution engine developed in section 4. Finally, section 6 concludes this thesis.

Chapter 2

Computer-Interpretable Guidelines (CIGs)

2.1 What is Evidence-Based Medicine?

Evidence-based medicine (EBM) is defined as "the conscientious, explicit, and judicious use of current best evidence in making decisions about the care of individual patients"[21].

EBM has become an integral part of medical training and ultimately on practice. Physicians-in-training are taught skepticism of anecdotal experience and to accept consensus recommendations of medical experts, expecting that such recommendations would be supplanted if contrary evidence is found through study[1].

The benefits of EBM have been cited many times, but little research has discussed its limitations[1]. According to Tonelli[1], EBM devalues the individuality of patients as well as the complex nature of sound clinical judgment, and research underpinning a medical conclusion may not apply to each individual circumstance.

2.2 From Text Guidelines to Computer-Interpretable Guidelines

2.2.1 Why are Guidelines Needed?

The Institute of Medicine defines clinical practice guidelines as “statements that include recommendations intended to optimize patient care that are informed by a systematic review of evidence and an assessment of the benefits and harms of alternative care options”

[20].

Davis et al. [6] describe the "clinical care gap" as actual day-to-day clinical practice differing from evidence-based knowledge leading to poor outcomes. The authors cite numerous examples including suboptimal management of rheumatoid arthritis in British Columbia where actual patient care was not consistent with the clinical practice guidelines which recommended early and aggressive antiheumatic drugs in certain instances. According to the authors, one strategy to enhance quality of care is to "improve the process by which knowledge – more specifically, clinical research findings and evidence-based practices – are incorporated into routine practice."

Woolf [26] surveyed the potential advantages and limitations of clinical guidelines. Potential advantages include improved consistency of care and health care outcomes for patients and improved patient knowledge of available procedures and risks. Potential limitations relate primarily to adequacy of the scientific evidence supporting clinical guidelines and biases of guideline developers, governments and insurers.

There is evidence that guidelines can assist practitioners when making medical diagnoses. A well-known study by Essex[8] documented the value of problem-oriented flow charts to make diagnoses at outpatient clinics in Tanzania. At the time of the study, most medical care in developing countries was provided by paramedical staff who undergo three years of training but who are not doctors. To solve a particular type of problem, researchers developed a flow chart for these paramedical workers to follow. These flow charts were used on 2,030 occasions to make diagnoses. On 94% of these occasions, doctors agreed with the diagnosis. Of the wrong diagnoses, 58 diagnoses would have resulted in the patient being given the same treatment or being referred, 36 in the patient being given unnecessary treatment or being referred and 32 in the patient having substantially different management. The author concluded that the problem-oriented methodology had acceptable levels of accuracy, repeatability and rapidity, and may have resulted in improved quality of medical care.

2.2.2 The Development of Clinical Guidelines

Tu et al. [25] propose six dimensions for modeling clinical guidelines:

- Provider behaviours that a guideline influences - does the guideline author wish to set goals or constraints, choose an alternative course of action among competing options, sequence a set of actions, or interpret data?

- Temporal dimensions of actions and data - is an action specified in the guideline a “one-shot” decision, episodic interventions, or continuous monitoring?
- Abstractions - does the guideline require data abstraction such as a person’s cholesterol level (low, normal or high)?
- Degree of uncertainty - how certain are the recommendations and data collection requirements?
- Point of view - is the “vantage point” of the guideline that of a doctor or nurse?
- Normal case and exceptions - does the guideline describe exceptional cases?

In 1992, the Institute of Medicine published a report on authoring guidelines called *Guidelines for Clinical Practice: From Development to Use* which suggests a collaborative approach with experts from numerous domains when developing clinical guidelines [9].

2.2.3 From Text to Computer-Interpretable Guidelines

Computer-interpretable guidelines (CIGs) require machine readable representations of the medical knowledge contained in the clinical guidelines.

Guidelines are written in text and need to be translated into a computer representation. Patel et al. [16] demonstrate that such translation is not a trivial task. Experts interpret the guidelines based on specialized knowledge from education and experience, creating a *situation model*. They also organize knowledge into a hierarchy with higher level concepts on top and gradually moving to lower level concepts. Non-experts organize knowledge based on the organization imposed by the guideline text. They may also miss “obvious” steps in the decision process that would be clear to experts. Computer implementations require that all steps be explicit. The authors suggest that experts work with computer scientists to create computer-interpretable guidelines.

Research has shown that the following features of clinical decision support systems are critical for improving clinical practice: decision support as part of physician workflow, the providing of useful recommendations rather than just assessments, and the availability of the support at the time and location of decision making[13].

2.2.4 Anatomy of a CIG

A typical clinical guideline system is shown in Figure 2.1.

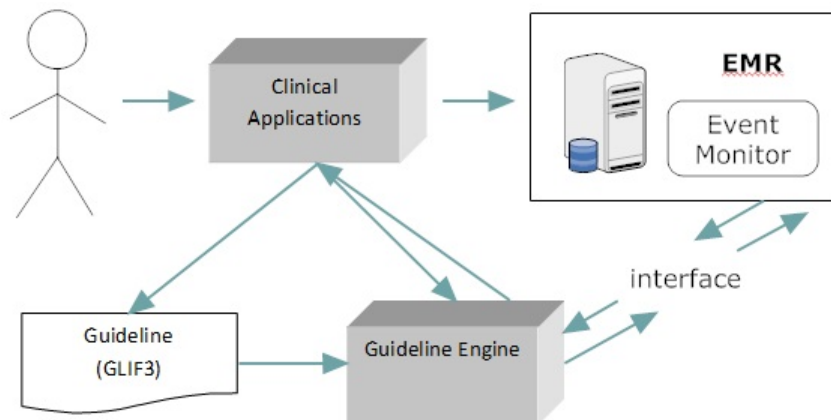


Figure 2.1: Guideline System Diagram

The user interacts with the system through *clinical applications*. The clinical applications call upon the *guideline engine* to execute the *guideline* which is specified in a representation framework such as GLIF. The execution engine uses data from the health care provider’s databases, but would generally use an *interface* because of differences in terminology and labeling between the guideline developer and the health care provider. The interface would translate the data from the health care provider’s databases into usable data for the guideline system.

2.2.5 Electronic Medical Records

The National Alliance for Health Information Technology defined an electronic medical record as follows[15]:

An electronic record of health-related information on an individual that can be created, gathered, managed and consulted by authorized clinicians and staff within one healthcare organization.

In practice, guidelines are not well deployed at the point of care, mainly due to a mismatch between decision support modules and electronic medical records (EMR) [23]. A guideline system must be able to request external data such as patient data in an electronic medical record (EMR). A specific drug, for example, might have numerous identifiers across hospitals. A guideline using a particular hospital’s identifier would likely require modification in order to be used at a different institution.

To improve portability of guidelines, researchers have developed controlled vocabularies to refer to the same items of data, and representation models for communicating

information amongst different stakeholders such as hospitals and insurance companies. An example of a controlled vocabulary is the Unified Medical Language System (UMLS) and an example of a representation model is the Reference Information Model (RIM).

Both UMLS and RIM are described later in this chapter.

2.3 Modeling Clinical Guidelines

2.3.1 Introduction

Decision rules are often represented in one of two formats: procedures and production rules [11]. A procedure is similar to a subroutine of a programming language and consists of references to data and logical statements that manipulate the data, as well as control structures that direct the flow of decision making [11]. A production rule is similar to an *if...then* statement in a programming language. An execution engine decides which rule to execute and proceeds to do so. An example of a production rule system is the language CLIPS.

Procedures have the disadvantage of mixing control and clinical knowledge, requiring that the developer be familiar with the programming language [11]. In addition, procedures may be difficult to maintain and require recompiling and redistribution when changes are needed.

Production rules may be developed by clinical guideline developers without programming knowledge. However, each production rule is independent, making it difficult for the developer to predict the order of rule execution and the interaction with other rules [11]. Some production systems, such as CLIPS, allow the programmer to specify a priority level for each production rule. The use of priority levels can make the rules difficult to read and debug.

2.3.2 Arden syntax

To overcome the disadvantages of procedures and production rules, researchers created a standard for representing knowledge in a shareable format. In 1991 the *Arden syntax* was published by the American Society for Testing and Materials (ASTM). The Arden syntax combines procedures and production rules and was later integrated into Health Level 7 (HL7).

Arden syntax code is organized into self-contained files called *Medical Logic Modules* (MLMs). Each Medical Logic Module represents a single clinical decision and is

organized into three categories: maintenance, library and knowledge. Each category is divided into subcategories called *slots*. The maintenance category contains information about the author, the institution and MLM version. The library category contains background information about the purpose of the MLM. The knowledge category contains the algorithm for decision making. The *evoke* slot defines when the MLM is to be executed. When the *evoke* condition is met, the MLM may start immediately, after a specified waiting period, or periodically. The *logic* slot contains the logic and procedures for decision making based on input data and the *action* slot writes the result to screen, sends an e-mail or calls on other MLMs.

Figure 2.2 shows sample code from HL7¹. The sample illustrates a *knowledge* category of a medical logical module for calculating body mass index (BMI). The *data* slot refers to three data items: size, weight and birth. The *logic* slot begins by calculating *BMI* and *age*. The slot then sets a variable called *classification* based on the results of user specified logic. Finally, the *action* slot outputs the body mass index (BMI) and classification.

¹HL7®Material included in this thesis is for non-commercial use and is copyrighted by Health Level Seven®International. All Rights Reserved. Use of HL7 copyrighted materials for commercial purposes is limited to HL7 Members and is governed by HL7 International's Intellectual Property Policy. HL7 and Health Level Seven are registered trademarks of Health Level Seven International Reg. U.S. Patent Trademark Office.


```
knowledge:
type: data_driven;;
data:
// Arguments
(size, weight, birth) := argument;
priority: ;;
evoke: ;;
logic:
// calculation of BMI
let bmi be weight / (size** 2); // BMI
// calculation of AGE
age := currenttime - birth; // AGE
// classification
if the age is less than 19 years then classification :- null;
// This classification is only valid for patients older than 19
elseif the bmi is less than 18.5 then classification:- localized 'under';
// classified as underweight
elseif the bmi is less than 25 then classification :=null; // BMI normal range
else let the classification be localized 'over';
// classified as overweight
endif;
bmi := bmi formatted with localized 'mag'; // construct the localized message
conclude classification is present ; // execute action slot

;;
action:
write bmi || classification || "."; // return result
return classification;
urgency: ;;
```

Figure 2.2: Arden Syntax

Note that the Arden syntax is static. It takes data, applies logic and draws a conclusion based on the data. The Arden syntax is part of the Health Level 7 standard.

A weakness of the Arden syntax is that it does not provide full support for conceptualizing a multi-step guideline that unfolds over time[18]. The Arden syntax could not express a guideline that requires, for example, medication, observation and then testing. Peleg, Tu et al. [18] identified a more versatile subset of approaches to computerized clinical guidelines based on a series of component tasks that are executed over time known as *task-network models*.

This important subset is discussed in the next subsection.

2.3.3 Task-Network Models

Task network models (TNM) typically are designed to represent complex, multistep clinical guidelines and for describing temporal and other relationships between component steps[18]. TMNs can model alternative pathways to control flow through a guideline and provide tools to visually represent the guideline or steps and the organization of tasks within them.

In this section, two task network models are discussed: Asbru and GLIF.

Asbru

Asbru was developed by the Vienna University of Technology and Stanford Medical Informatics (known as the “Asgaard project”) and is an intention-based plan representation language to represent clinical guidelines and protocols as time-oriented skeletal plans. The skeletal plans provide an outline of a clinical guideline and leave the implementation to the run-time engine.

Skeletal plans are plan schemata at various levels of detail that capture the essence of a procedure, but leave room for execution-time flexibility in the achievement of particular goals[10]. The application of guidelines involves an interpretation by the care provider of the skeletal plans that have been designed by the guideline’s author.

The Asbru approach distinguishes between design time and execution time of a clinical guideline. During design time, the guideline author prescribes (i) the intended intermediate and overall pattern of patient states, (ii) an intended plan and (iii) specific actions. The intended intermediate and overall pattern of patient state could be, for example, to maintain morning blood glucose to stay within a certain range. The intended plan could be to prescribe a certain class of drugs twice a day, and the specific action could be to administer a particular drug in the morning and in the evening.

During execution-time, the care-provider performs actions, which are recorded, observed and abstracted over time into an abstracted plan.

Guideline Interchange Format

Intermed Collaboratory was created in 1994 as a collaboration among Stanford University, Brigham and Women’s Hospital and Columbia University with the principal goal

of developing tools and resources for disseminating clinical guidelines across medical disciplines and settings. A major contribution of Intermed was the development of the Guideline Interchange Format (known as “GLIF”) which is a computer-based format to distribute guidelines across different institutions and systems.

GLIF represents a clinical guideline as a network of steps resembling a flowchart. Steps can implement branching logic or execute actions.

In 1998, Intermed published its first version of GLIF called GLIF 2.0.² At the time of writing, the latest release of GLIF is version 3.5.

The GLIF model is implemented in this thesis. A detailed description of GLIF is reserved for Chapter 3.

2.4 Implementations of Computer Clinical Decision Support

Implementing computer guidelines in a health care setting is an expensive process. In addition to the difficulties associated with creating the guidelines themselves, there exist difficulties with the institutions collecting, using and transmitting the data. Hospitals, for example, may use numerous terms to communicate the same information. Greenes [11] described a situation where decision logic was exported from one hospital to others. The recipient hospitals wanted to identify patients on I.V. antibiotics that were capable of oral intake, but represented “oral intake” in a textual format and in numerous different ways: *orally, take orally, by mouth, swallow, etc.* A computer-interpretable guideline at one health care institution may not work at another due to such differences. This is a severe impediment to implementing computer guidelines. Also, in the past, there was no consistent framework for communicating medical information. Fortunately, the Unified Medical Language System (UMLS) was created to provide a consistent framework for referring to medical information in a language neutral manner and the HL7 Reference Information Model (RIM) was created to provide a consistent means of organizing and communicating this information.

2.4.1 The Unified Medical Language System (UMLS)

UMLS was created to help build software programs that need to use the language of biomedicine and health. For example, electronic medical records store data about patient

²Versions prior to GLIF 2.0 were not released to the public.

medications. Medication can be represented using many different specialized vocabularies. Developers of software that use data from electronic medical records may need to make significant changes to the software in order to target a different electronic medical record system. UMLS was designed to link these disparate vocabularies into a common representation. UMLS is maintained by the U.S. National Library of Medicine³.

Concept (CUI)	Terms (LUIs)	Strings (SUIs)	Atoms (AUIs) * RRF Only
C0004238 Atrial Fibrillation (preferred) Atrial Fibrillations Auricular Fibrillation Auricular Fibrillations	L0004238 Atrial Fibrillation (preferred) Atrial Fibrillations	S0016668 Atrial Fibrillation (preferred)	A0027665 Atrial Fibrillation (from MSH)
			A0027667 Atrial Fibrillation (from PSY)
		S0016669 (plural variant) Atrial Fibrillations	A0027668 Atrial Fibrillations (from MSH)
	L0004327 (synonym) Auricular Fibrillation Auricular Fibrillations	S0016899 Auricular Fibrillation (preferred)	A0027930 Auricular Fibrillation (from PSY)
		S0016900 (plural variant) Auricular Fibrillations	A0027932 Auricular Fibrillations (from MSH)

Figure 2.3: UMLS Metathesaurus

The “Metathesaurus” is what links the numerous disparate medical vocabularies into a common system. It is organized by *concept*, with the purpose of linking similar concepts from the numerous vocabularies in use into a single concept that can be referenced by a computer program. Each concept has a unique code called a Concept Unique Identifier (CUI). Each CUI is linked to at least one unique atom identifier (AUI), single string identifier (SUI) and lexical unique identifier (LUI). The AUI refers to the instance of the concept in each source vocabulary. A single string identifier (SUI) is created whenever

³see <http://www.nlm.nih.gov/research/umls/>

the same string appears in multiple vocabularies. Finally, a lexical unique identifier joins strings that are lexical variants of one another.

Figure 2.3 illustrates concept unique identifier C0004238 concerning Atrial Fibrillation⁴. Note that there are two source vocabularies referenced: MSH refers to Medical Subject Headings which is maintained by the U.S. National Library of Medicine⁵ and PSY which refers to the Thesaurus of Psychological Index Terms and is maintained by the American Psychological Association⁶. Both MSH and PSY use the same term “Atrial Fibrillation” so a single string identifier is created. Finally, other parts of the MSH and PSY refer to “Auricular Fibrillation” and its plural variant. These concepts are joined in a lexical unique identifier because the singular and plural terms are not lexicologically distinguishable. Finally, Atrial Fibrillation is joined with Auricular Fibrillation to form a single concept.

Developers of computer systems for hospitals, for example, can use these concept unique identifiers to communicate with insurance companies, or to update electronic medical records. These concepts are critical to developers of computer-interpretable guidelines to ensure that consistent understandable information is communicated.

2.4.2 HL7 and the Reference Information Model (RIM)

Health Level Seven (HL7) was founded in 1987 and is a non-profit organization that creates standards for the exchange, integration, sharing and retrieval of electronic health information to support clinical practice and health services.

The impetus for a widespread reference model for communicating medical data came in 1986 at the MEDINFO Conference where an influential group of end users and system vendors determined that communication of medical data between heterogeneous health-care information systems is necessary to accelerate the diffusion of automated systems technology in health care [12].

The RIM is a standard for modeling health information, consisting of generic classes from which more specific health classes are derived. For example, the RIM contains a generic class called Act from which two subclasses - Observation and Procedure - are derived.

⁴UMLS®Reference Manual [Internet]. Bethesda (MD): National Library of Medicine (US); 2009 Sep. 2, Metathesaurus. Reproduced with permission. Available from: <http://www.ncbi.nlm.nih.gov/books/NBK9684/>

⁵<http://www.nlm.nih.gov/mesh/meshhome.html>

⁶<http://www.apa.org/pubs/databases/training/thesaurus.aspx>

The major RIM classes are:

- *Act* which represents the actions that are executed;
- *Participation* which expresses the context for an act in terms of who performed it, for whom it was done and where it was done;
- *Entity* which represents the physical things and beings of interest in health care;
- *Role* which establishes the roles that entities play in the health care acts;
- *ActRelationship* which represents the binding of one act to another; and
- *RoleLink* which represents relationships between individual roles.

At the time of writing, the latest version of the RIM is 3.0.

2.5 Comparing Computer-Interpretable Guideline Models

Peleg et al. [18] compared six computer interpretable guideline models - Asbru, EON, GLIF, Guide, Prodigy and PROforma - based on eight dimensions the authors believe capture the structure of a computer interpretable guideline framework.

Each of the computer interpretable guideline frameworks studied does each of the following to some degree:

- represents a guideline as a consisting of numerous tasks and relationships between the tasks;
- specifies the guideline's intention or goal;
- models an action task such as a medical action (e.g. prescribe a medication) in some manner;
- allows for decision-making by providing constructs such as decision steps or switch constructs;
- provides expression/criterion languages to specify decision criteria such as comparison operators (is blood pressure above a certain threshold);
- abstracts medical terms - for example "high blood pressure";

- represents medical concepts; and
- represents patient information and maps to an electronic medical record.

Chapter 3

Guideline Interchange Format (GLIF)

GLIF is a methodology for modeling and representing clinical guidelines in a structured manner for clinical decision support applications.

GLIF is comprised of classes, attributes and the relationships among the classes needed to model clinical guidelines. It presents a guideline as a flowchart of temporally sequenced nodes called *guideline steps*. There are five principal steps that are in the specification: the *decision step*, the *action step*, the *branch step*, the *synchronization step* and the *patient state step*.¹

GLIF is designed to be used in a variety of contexts including guidelines involving different stages of medical problem (screening, diagnosis), setting (inpatient or outpatient) and time frame (emergency, chronic)[19].

Figure 3.1 illustrates a guideline for the management of chronic cough in non-immunodeficient adults.² The guideline begins with a patient state step called Chronic Cough. The second step is an action step to fetch relevant patient cough related data. The next step is a decision step with two branches: one branch if an ace inhibitor (ACEI) is suspected as the cause of the cough and another if ACEI is not suspected.

If ACEI is suspected as the cause of the cough, an action steps tells the clinician to order that the patient stop using the ACEI for four weeks. A patient state step follows to indicate that four weeks must have passed for the guideline to continue. Once four weeks have passed, an action step is invoked : evaluate patient. The evaluate patient task calls a subguideline to investigate other possible causes.

¹This implementation focuses on the GLIF 3.4 specification.

²Figure 3.1 is a screen shot of a portion of the cough guideline provided by the Peleg[17] using Protege software.

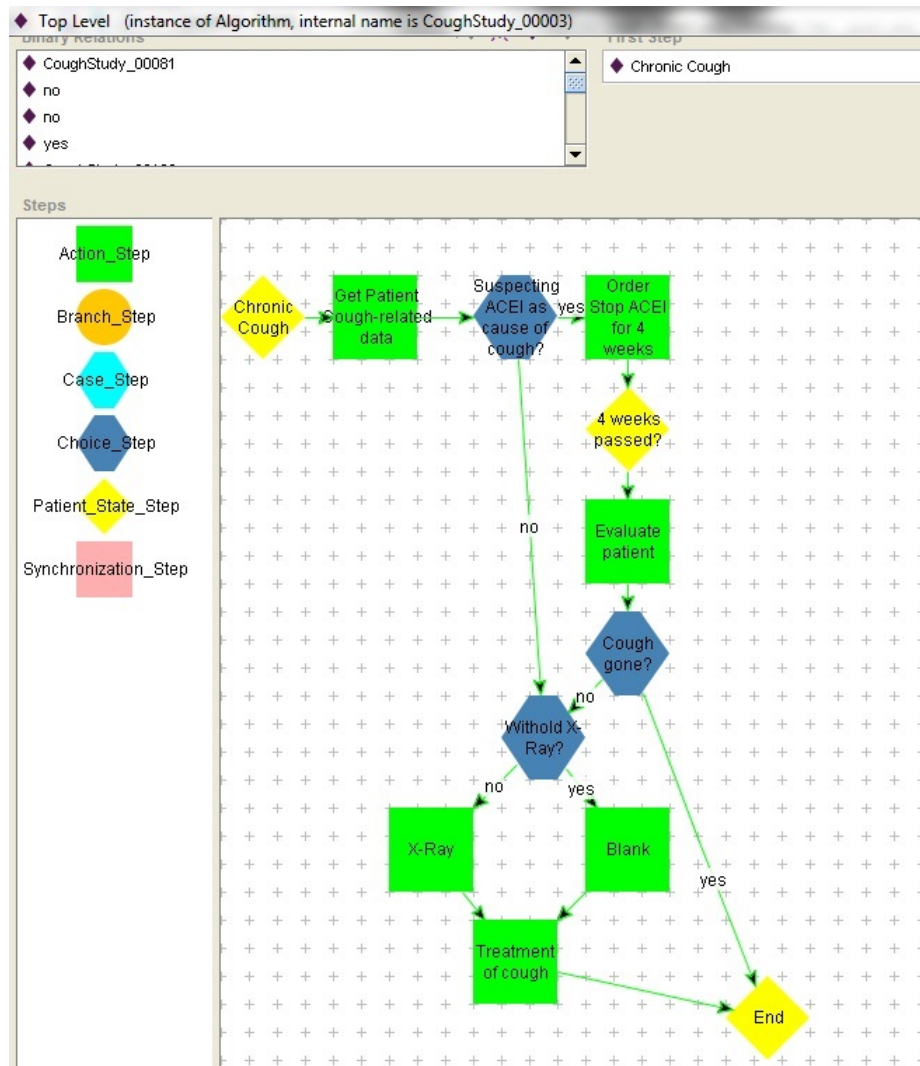


Figure 3.1: Cough Guideline

Once the patient is evaluated, another branch step asks whether the cough is gone. If the cough is gone, the guideline ends. If the cough is not gone, the guideline follows the same path as it would have followed if the clinician did not suspect ACEI as the cause of the cough.

If ACEI is not suspected as the cause of the cough, another decision step (specifically a choice step) asks whether to withhold x-ray. If the clinician wants an x-ray, an action step tells him/her to order one. If not, an action step prescribes a treatment for the cough.

The developers of GLIF also developed an execution engine called GLIF3 Guideline Execution Engine (GLEE) but unfortunately the software is unavailable.³ GLEE allows

³The software belongs to Stanford University and is no longer in the public domain.

guideline authors to create a guideline in Resource Description Framework (RDF) format and imported to be executed. Fortunately, there are many good free RDF authoring tools for ontology development such as Protege⁴.

3.1 The Three Layers of Abstraction

GLIF represents clinical guidelines at three levels of abstraction labeled A to C, with level A being the highest level of abstraction. At Level A, the guideline author is concerned with conceptualizing a guideline as a flowchart, while ignoring details such as decision criteria. At level B, the details omitted in Level A are specified. Finally, at Level C, the guideline is implemented in a specific institutional setting. Patient data references must be mapped to institutional electronic medical records.

3.2 Major Classes

3.2.1 Guideline Class

The *Guideline* class is used to model clinical guidelines and subguidelines. Each guideline instance contains an *Algorithm* instance which describes the guideline's flowchart of steps. A step in a guideline may also call upon other guidelines (called subguidelines) to execute certain aspects.

Guidelines are designed to be independent of one another. GLIF contemplates a parameter passing mechanism between a guideline and subguideline.

3.2.2 Steps

An algorithm is comprised of one or more steps that implement some aspect of the guideline.

The different steps are as follows:

1. The PatientStateStep class is an entry point into a guideline and allow for labeling patient states.
2. The DecisionStep class represents a decision step in the guideline.

⁴<http://protege.stanford.edu/>

3. The `ActionStep` class represents an action to be performed and can be medically oriented, such as a recommendation to a clinician, or be programming-oriented actions such as data retrieval.
4. The `BranchStep` executes child activities simultaneously on separate threads.
5. The `SynchronizationStep` controls the flow through the guideline when one or more children of the Branch Step finish.

Patient State Step

Patient state steps are used as entry points in a guideline. A patient may have already had an encounter with a clinician and have been on medication for a certain amount of time. The clinician will have followed some of the early steps in the guideline and will need to resume at a later point.

When a patient arrives at the hospital, his current state is compared to the last patient state that was recorded for him. The `PatientStateStep` contains a field called `patient_state_description` that specifies (using an expression) the patient condition contemplated by the patient state step.

The `patient_state_description` is expressed using an expression language such as GELLO (discussed later). An example of a description is provided below.

```
((now - coughStartTime) > 3 weeks) and  
(Age > 18 years) and  
not (immunocompromisedEndTime >= now)
```

The above GELLO expression states that the `PatientStateStep` is applicable to a patient who has had a cough for at least three weeks, is at least 18 years of age, and who is not immunocompromised. This example is found in the chronic cough guideline from Peleg [17].

Decision Step

Decision steps are used where a choice has to be made among alternatives. Such steps resemble an *if..then* statement in a programming language.

The GLIF standard defines two types of decision steps:

- A *case step* where control flows to a branch in the decision tree based on criteria specified in the decision option, or a default step; and

- A *choice step* where the guideline suggests preferences among alternative choices but leaves the actual choice to an external agent (i.e. the clinician)

In a *case step*, the guideline developer states a condition which is represented as an expression. The execution engine attempts to find a decision option that matches the expression. If no match is found then the default option is selected.

In a *choice step*, the guideline does not provide deterministic selection criteria and requires an external agent (i.e. a human or another program) to make the decision. This type of step would be used, for example, where the decision cannot be represented unambiguously or where the decision is critical. The decision options do not have to be mutually exclusive.

Case Step

The *Case Step* contains a condition which determines the control flow to one of a set of possible guideline steps. The condition is an expression that is evaluated. The expression result is compared to each option's `options.condition_value`. If the condition matches, control flows to the guideline step that is specified by that decision option's destination. Should there be no match, control flows to a default destination.

Choice Step

In a Choice Step, there is no deterministic selection criteria. Generally a human user must make the decision. The decision options need not be mutually exclusive. The GLIF specification sometimes provides data to assist the end user when making a decision.

The specification contains a class called Choice representing the base class for the type of choice.

The *utility choice step* represents a choice that uses utility theory when deciding among the available options. The data structure would point to an algorithm to evaluate the choices.

The *rule in choice* specifies rule-in, rule-out, strict-rule-in and strict-rule-out criteria for each decision option. These criteria are used by the execution engine to evaluate the alternatives. A rule-in is a factor favoring a particular choice. A rule-out is a factor suggesting that a particular choice not be selected. A strict-rule-in ranks a choice the best among the alternatives. A strict-rule-out excludes a choice. For example, if the decision is to prescribe a certain medication, a strict-rule-out condition would be that the person has an allergy to that medication.

An algorithm can be used to evaluate rule-in and rule-out conditions and provide a suggestion to the user.

3.2.3 Action Steps

Action steps specify the clinical actions that are going to be performed in the patient-care process. Action steps consist of one or more tasks that need to be performed known as action specifications. An action specification can include a call to another guideline (a *subguideline_action*).

The following are some of the more important actions.

AssignmentAction

The *AssignmentAction* is used to create or modify a primitive data item.

MedicallyOrientedActionStep

A *MedicallyOrientedActionStep* is used to create a medical action such as prescribing a medication or recording an observation.

MessageAction

The *MessageAction* is used to send a message to the user.

GetDataAction

GetDataAction retrieves patient data from the EMR as HL7 RIM objects and transforms them to query result data types.

Subguideline Action

The *SubguidelineAction* executes a subguideline. Input arguments can be provided to the subguideline and output arguments can be retrieved.

3.2.4 Branch and Synchronization Steps

A *BranchStep* is used to model concurrent guideline steps. Control is directed to multiple guideline steps for parallel execution.

A *SynchronizationStep* is used in conjunction with a branch step. The parallel steps executed by the Branch Step eventually converge to a single step. The Synchronization

Step specifies whether all, some or one of the preceding steps must have been completed before control can move to the next step. The continuation criteria is expressed as a logical expression such as “Step_A or Step_B” to indicate that the next step is to continue once either Step A or Step B are completed.

3.2.5 Three Value Criteria

GLIF uses three value logic when boolean expressions are evaluated. Three value criteria can evaluate to true, false or unknown.

3.3 Authoring using Protege

The GLIF examples in the literature rely on modeling using Protege - an ontology development framework developed at Stanford University.

3.4 Expression Languages

An expression language is used to build up statements that query data, logically manipulate them, provide for reasoning over them and facilitate calculations[11].

GELLO was designed to represent logic in clinical guidelines and was developed initially to represent the procedural component of GLIF. Since then, it has been used outside of GLIF and is now an HL7 standard.

GELLO is based on Object Constraint Language (OCL). OCL is a language for describing constraints and querying UML Class Models. For example, a UML class diagram might contain one class for Patient and one for Hospital. A constraint could be, for example, that the maximum number of beds at the hospital is 100. GELLO is a superset of OCL and was created to constrain and query medical ontologies including the Reference Information Model (RIM) discussed earlier.

The GLIF implementation outlined in this thesis does not use GELLO. A generic expression evaluator found in the public domain was used as a placeholder.

Chapter 4

Implementing GLIF in C# .NET

Figure 4.1 depicts an *integrated development environment (IDE)* for the development and execution of clinical guidelines. The left hand side of the IDE shows the list of available activities such as *PatientStateStep*. The bottom of the IDE shows important debugging information listing the order of execution of guideline steps (activities) and important information about the execution of each step (activity). The right hand side is the guideline drawing canvas. Developers can drag and connect guideline steps and set important parameters. On the bottom of the drawing canvas is a tab labeled “Variables.” The workflow developer is able to set global variables that will be used throughout the workflow. Developers can also add arguments to a workflow by clicking “Arguments.” An *InArgument* is used to pass a value to a workflow variable from outside the workflow. An *OutArgument* is a value produced by the workflow for retrieval from outside the workflow. InArgument is analogous to a parameter in a C# function and an OutArgument is analogous to a return value of a function.

The file menu contains numerous options. It allows the user to import an RDF formatted guideline created using Protege into the system. The user can also save the guideline into Workflow Foundation’s XAML based format. The file menu also allows the user to run the guideline and view execution steps.

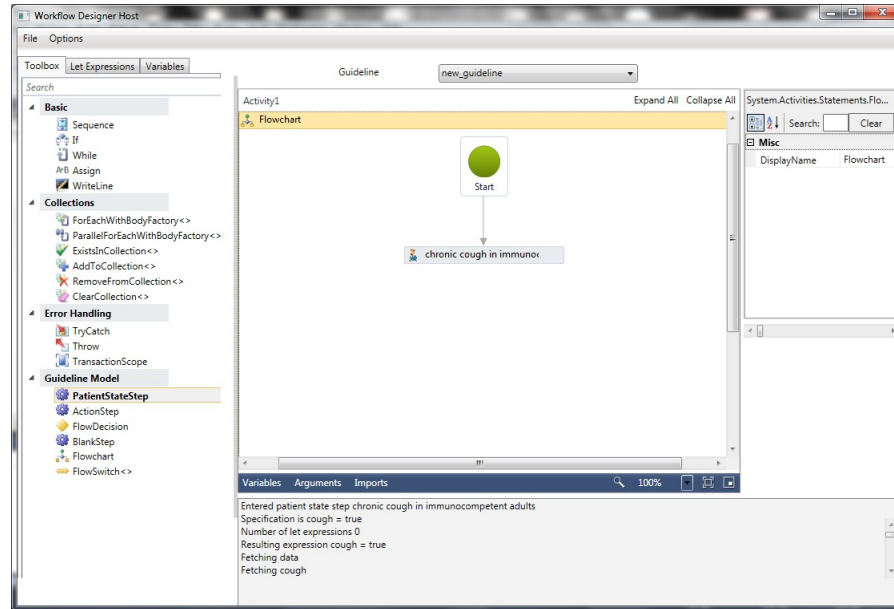


Figure 4.1: Integrated Development Environment (IDE)

Figure 4.2 shows the major assemblies created during the development process and their dependencies. The assembly *DesignerHost* is the executable file that contains the integrated development environment. This assembly also parses *GLIFObjects* into workflow activities. The *GLIFObjects* assembly contains classes that correspond to concepts in the GLIF specification (Guideline, PatientStateStep etc) and is used to parse RDF files produced by Protege. The *ActivityLibrary* assembly contains the classes used in this GLIF implementation such as PatientStateStep. The *ActivityLibrary.Design* assembly contains the GUI designers for the activities in the *ActivityLibrary* assembly. The *IHostCommunication* assembly contains the interface that is used by the *DesignerHost* and the workflow runtime to communicate back and forth. The arrow linking each assembly shows that an assembly is dependent on another. For example, the *DesignerHost* assembly depends on the *IHostCommunication* assembly.

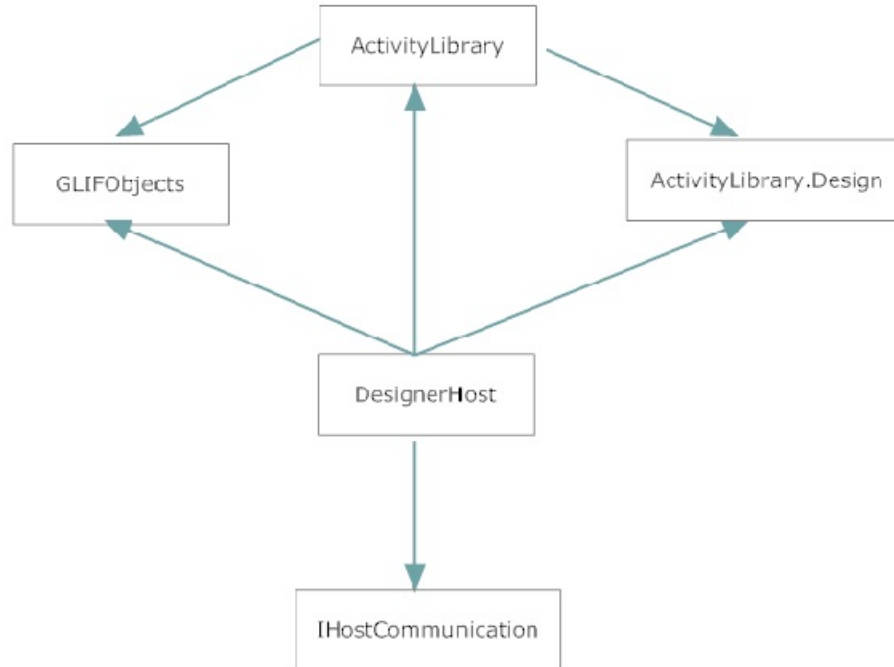


Figure 4.2: Major Assemblies

4.1 Introduction to .NET Framework

The .NET Framework is a software framework designed by Microsoft for application development. Microsoft created an object oriented programming language called C# (pronounced C Sharp) specifically for programming .NET. The developers of C# wanted the language to be simple, modern, general-purpose and object-oriented[7]. Programs targeting the .NET framework are typically written using Visual Studio in either C# or Visual Basic.

One benefit of .NET is that it frees developers from low level hardware details. C# .NET programs are compiled into an intermediate format called the Common Intermediate Language (CIL) to be run by a Common Language Runtime (CLR). The CLR is similar to the Java Virtual Machine. A major benefit of the CLR is that programmers no longer have to compile source code for each execution environment.

The .NET Framework contains a rich library of extensible classes for application programming including classes for graphics, cryptography, web development, network communication and database access. The Framework is constantly improving with new functionality coming out each new version.

4.2 Windows Workflow Foundation

A typical real world problem can be broken down into individual tasks that are to be carried out in some order. The set of all the tasks to solve a problem is the *workflow*. To write an application in a traditional language (e.g. C++), one must determine what the application is to do and how to do it. An application written in a traditional programming language is also a bundle of tasks that are completed in some order. The “how to” refers to all the code consisting of variables, functions, classes, events, etc that make up a modern application program. There is no clear distinction between the “what” and the “how.” In order to modify a task, a developer often has to traverse numerous unrelated lines of code to modify the relevant code and hope that there will be no unintended consequences.

Windows Workflow Foundation (WF) offers a *declarative approach* to programming that separates the programming task from the business logic[5]. Windows Workflow Foundation is declarative, visual and “infinitely flexible” [5]. A programming task is broken down into numerous components called *activities* that accomplish a unit of work. The workflow itself specifies the “what” to do without attention to the “how”. A workflow might start by requesting some data from an external source and then sending an email. The workflow designer need only focus on the high level concepts consisting of data access and email. The actual “how to” is left to the implementation of the activities. Activities can be developed and tested individually and reused in a different context. Suppose the workflow designer wanted to insert a new task between the data access activity and the email activity. The designer need only insert the relevant activity and set relevant parameters without any focus on the minute implementation details of each activity.

Windows Workflow Foundation was first released with version 3 of the .NET Framework. The framework underwent a major revision with version 4 of the .NET Framework. The .NET Framework is available free of charge, greatly expanding the number of potential users of WF to execute clinical guidelines.

In the parlance of WF, a *workflow* is a set of tasks called *activities* that are executed in some defined order. The *declarative approach* allows for separation of the design of the task from the implementation details.

Workflows are executed by the *workflow runtime*. The workflow is run by a *host* such as an application program. The WF Framework allows a host to communicate with the workflow (and vice versa) using *workflow services*. The .NET Framework also comes bundled with Windows Communication Foundation (WCF) that allows developers to

build connected, service-oriented applications. The facilities of WCF are available to WF, making WF a good choice when developing distributed systems.

4.3 Workflow Modeling Styles

WF 4.0 offers two workflow modeling styles: sequential and flowchart. A sequential workflow executes activities in a sequential order. The flowchart style represents a workflow as a series of steps that are not necessarily sequential. The flowchart style offers branching logic whereby the next step selected depends on a designer specified condition.

4.4 Activities

Activities are the building blocks of Windows Workflow Foundation. An activity carries out some aspect of the workflow task. For example, an activity might output text to a device, send an email, or branch to another activity depending on the result of a rule. WF is bundled with a forward chaining rules engine and can be used for applications requiring execution of rules.

WF 4.0 comes with numerous built in activities from which a developer can use, or customize. These activities are contained in the `System.Activities.Statements` namespace. In this section, some of the basic workflow activities are discussed as well as how to create a custom activity in order to illustrate the flexibility of WF 4.0.

If Activity

The *If Activity* adds branching logic to workflow, similar to the *if...then...else* syntax found in many programming languages.

WriteLine Activity

The *WriteLine Activity* writes a string to either the console, or to a specified text writer.

Assign Activity

The *Assign Activity* is used to change the value of a variable. For example, a variable in the workflow could change as a result of information supplied by a clinician on a form.

SendAndReceiveActivity

The *SendAndReceiveActivity* is used to send a message to a specified location and receive a response back. A practical use of this activity is to send a message to a host application and await a response.

4.4.1 Creating a Custom Activity

The first step to create a custom activity is to determine the specific tasks that the activity needs to perform. An activity can be a *composite* activity consisting of multiple activities, or a single activity. Composite activities inherit from the *SequentialWorkflowActivity* class. Single activities generally inherit from *CodeActivity*, *NativeActivity*, *AsyncCodeActivity*, or *Activity*.

The *CodeActivity* class is used as base class for an activity that cannot be modeled using existing activities alone. Developers can override the *Execute* method to provide custom code.

Suppose a developer wanted to create an activity to simply output “Test” to the Visual Studio console. The developer can create a new public class by inheriting from *CodeActivity*. The implementation logic is provided by overriding the *Execute* method.

Sample code is provided below:

```
using System.Activities;
using System;

namespace ActivityLibrary
{
    public class TestActivity:CodeActivity
    {
        protected override void Execute(CodeActivityContext context)
        {
            Console.WriteLine("Test");
        }
    }
}
```

Figure 4.3: CodeActivity sample

The *TestActivity* class can be added to a workflow to be run by the workflow runtime.

4.5 The Flowchart Style

The *Flowchart* is a special type of activity. To create a flowchart, one drags a Flowchart activity onto the drawing surface.

All nodes on the Flowchart inherit from *FlowNode* which is an abstract base class. Only three types of objects can be added to a Flowchart: *FlowStep*, *FlowSwitch<>* and *FlowDecision*.

4.5.1 FlowStep

Custom activities inherit (directly or indirectly) from the `System.Activities.Activity` class. The *Activity* class encapsulates all the behaviour associated with an activity. It is important to note that an activity cannot be added directly to a Flowchart. Rather, the activity is converted to a FlowStep and the FlowStep is added to the Flowchart. Example code is provided below.

```
PatientStateStep ps = new PatientStateStep();  
FlowStep fs = new FlowStep();  
fs.Action = ps;
```

The above code is automatically created when an activity is added to a Flowchart in the workflow designer.

4.5.2 FlowSwitch<>

FlowSwitch<> allows the user to specify two or more possible branches to take based upon the result of a condition. For example, a workflow handling a bank transaction might choose the next node to execute based on whether the transaction is a deposit, withdrawal or other transaction. The workflow would contain a branch for each of these possibilities and follow the branch based on the type of action the customer is taking.

4.5.3 FlowDecision Activity

FlowDecision is similar to *FlowSwitch<>* in that it provides branching based on the results of a condition. There are two major differences: a *FlowDecision* condition evaluates to true or false only and can only branch to one of two possible branches. The *FlowSwitch<>* can branch to more than two branches and based on a non-boolean value such as an integer or string.

4.6 Workflow Services

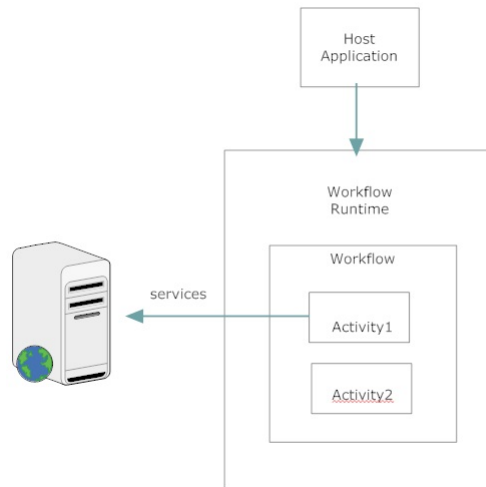


Figure 4.4: Workflow Services

Software is becoming more distributed. Workflow services allow a workflow to use the facilities of Windows Communication Foundation (WCF) for communicating across a network. This is a useful capability in today's world of distributed computing.

An activity can call external services and receive communications.

4.7 XAML Format

Workflow Foundation workflows can be created (and saved) in a format called eXtensible Application Markup Language (XAML).

Guidelines developed using the designer developed in this thesis are saved in XAML format. A possible future extension would be an option to save them in RDF format.

4.8 Implementation of GLIF3.4 in WF

In the remainder of this chapter, the implementation of GLIF3.4 in WF 4.0 is discussed in detail. This section discusses implementation of the hosting environment and communication with the host. The following sections discuss expressions, custom activities, and parsing a guideline developed in RDF (using Protege).

4.8.1 Implementing the Host

The base code for the workflow host was obtained from Bruce Bukovic's book *Pro WF: Windows Workflow in .NET 4*. The main window is found in the `MainWindow` class and is opened when the user runs the application.¹

The appearance of each activity on the drawing surface is provided by the activity designer associated with the activity. Each activity has an `ActivityDesigner` attribute associated with it. The purpose of the `ActivityDesigner` is to specify how the activity is to look visually on the screen. Note that *FlowDecision* and *FlowSwitch*<> do not derive from `Activity` and are not activities.

4.8.2 Host-Workflow Communication

Two helper interfaces were created for communication between the host application and the workflow: *IHostMessaging* and *IHostDataRequest*.

IHostMessaging is used to send messages to the host application such as debugging information. The interface also allows the workflow to send a request to terminate to the host. This occurs, for example, when a *PatientStateStep* evaluates to false meaning that the guideline cannot continue. Finally, the `FetchDisplayName` fetches the `DisplayName` property of a specified *FlowDecision*. This method is a workaround for challenges associated with implementing GLIF's Decision Step (discussed later).

The *IHostMessaging* interface is shown in Figure 4.5.

```
namespace IHostCommunication{
    public interface IHostMessaging {
        void SendMessageToConsole(string text);
        void SendMessageToUser(string text);
        void Terminate(string reason);
        void SendMedicalInformationTask(MedicalActionSpecification medicalInformation);
        string FetchDisplayName(string node);
        void ClearDisplayNames();
        Dictionary<string, string> GetDisplayNames();
        void SetDisplayName(string node, string name);
    }
}
```

Figure 4.5: *IHostMessaging*

¹Bukovic granted license to use the code for commercial or non-commercial purposes but restricts publishing the code

IHostDataRequest is used to request data and user input from the host. Data requests consist primarily of Concept Unique Identifiers (CUI) for items of data. The class *DataItem* encapsulates data requested from the EMR. The RequestUserInput method is used to request input from the user.

Code for the *IHostDataRequest* interface is shown in Figure 4.6.

```
using System.Collections.Generic;
using System;
using System.Activities.Statements;
using System.Activities;

namespace IHostCommunication {
    public interface IHostDataRequest {
        void RequestData(string sourceActivity, List<DataItem> data);
        event EventHandler<ResponseDataEventArgs> SendDataBack;
        void RequestUserInput(UserInput input);
        event EventHandler<BooleanResponseEventArgs>
        SendUserInputBack;
    }
}
```

Figure 4.6: IHostDataRequest

MainWindow contains two instance variables to refer to classes that implement IHostMessaging and IHostDataRequest : hostCom and hostDataSvc respectively. These variables are set when the user clicks Run on the File Menu (via the menuRun_Click event handler). The menuRun_Click event handler is a method that initializes the workflow and then executes it. The method subscribes for messages from the workflow. The “+=” operation specifies the method to call when the event is triggered. For example, the event NotifyHost calls method hostCom_Notification when raised by the workflow.

Code used by the host application to subscribe to events is shown in Figure 4.7.

```
hostCom.NotifyHost += new EventHandler<HostNotifyEventArgs>(hostCom_Notification);
hostCom.TerminateEvent += new EventHandler<TerminateEventArgs>(OnTerminate);
hostDataSvc.RequestDataMethod += new
EventHandler<RequestDataArgs>(hostData_Request);
hostDataSvc.RequestUserInputEvent +=
new EventHandler<IHostCommunication.UserInputEventArgs>(UserInputRequest);
```

Figure 4.7: Event Handlers

The variables hostCom and hostDataSvc are added to the WorkflowApplication instance (called app) as follows:

```
app = new WorkflowApplication(activity);
```



```
app.Extensions.Add(hostCom);
app.Extensions.Add(hostDataSvc);
app.Run();
```

Once the extensions are added and the workflow is run, the workflow activities obtain a reference to the `hostCom` and `hostDataSvc` objects. Each activity inherits from *BaseActivity* which contains the following four instance variables:

```
protected IHostCommunication.IHostMessaging host;
protected IHostCommunication.IHostDataRequest requestdata;
protected WorkflowDataContext dataContext;
protected WorkflowInstanceProxy proxy;
```

Figure 4.8: Instance Variables of BaseActivity

The *host* variable contains the reference to the host communicator and is used for sending messages to the host. The *requestdata* variable is used to request data from the host such as the values of parameters. The *dataContext* variable is used to get or set the workflow variables. For example, variables will need to be set in response to data received from the host. Finally, the *proxy* variable is used to obtain a reference to the *Flowchart* instance of the workflow. The *Flowchart* instance is needed to locate the next activity in the chart. As will be discussed later, this variable is needed to work around a problem associated with decision steps.

The *BaseActivity* class sets *host* and *requestdata* using the following code:

```
host = context.GetExtension<IHostCommunication.IHostMessaging>();
requestdata = context.GetExtension<IHostCommunication.IHostDataRequest>();
```

The variable *context* is passed to the *Execute* method when the activity is run by the workflow runtime. The method *GetExtension* fetches the instance of *IHostMessaging* and *IHostDataRequest* that were created by the host when the workflow was run.

To communicate back to the host, the activity simply needs to call the relevant method in the interface. For example, the *SendMessageToConsole* method sends a message to the host to print in the debug area.

```
host.SendMessageToConsole(message);
```

Inside the implementation of *IHostMessaging* (called *HostMessaging*), the *SendMessageToConsole* method raises the *NotifyHost* event which triggers a call to *hostCom_Notification* (discussed earlier).

```
public void SendMessageToConsole(string message) {  
    if (NotifyHost != null)  
        NotifyHost(this, new HostNotifyEventArgs(message));  
}
```

To implement host-to-workflow communication, a similar technique is used. The host calls a method of the HostMessaging instance (hostCom) which then raises an event that is handled by the workflow and the workflow continues.²

4.9 Evaluating Expressions

Many steps and tasks in GLIF require the ability to evaluate an expression. Expressions are typically written in GELLO syntax. Implementation of GELLO is beyond the scope of this thesis.

The execution engine in this thesis uses an evaluation engine from the public domain that is easily extendible for future implementation of GELLO.³

A screen shot of the rules engine tester provided in the public domain is provided in Figure 4.9.

Some guidelines require the ability to calculate a person's age, or to determine if a certain amount of time has passed. The evaluation engine used does not have this ability. Two functions were added to the evaluation engine to implement this functionality: age[d,m,y] (or age[variable] where variable is a date) and datedif[date[d,m,y] , date[d,m,y]].

4.10 Designing GLIF Activities

4.10.1 The BaseActivity Class

All of the custom activities developed in this thesis inherit from the *BaseActivity* class. *BaseActivity* provides needed “plumbing” common to all custom activities.

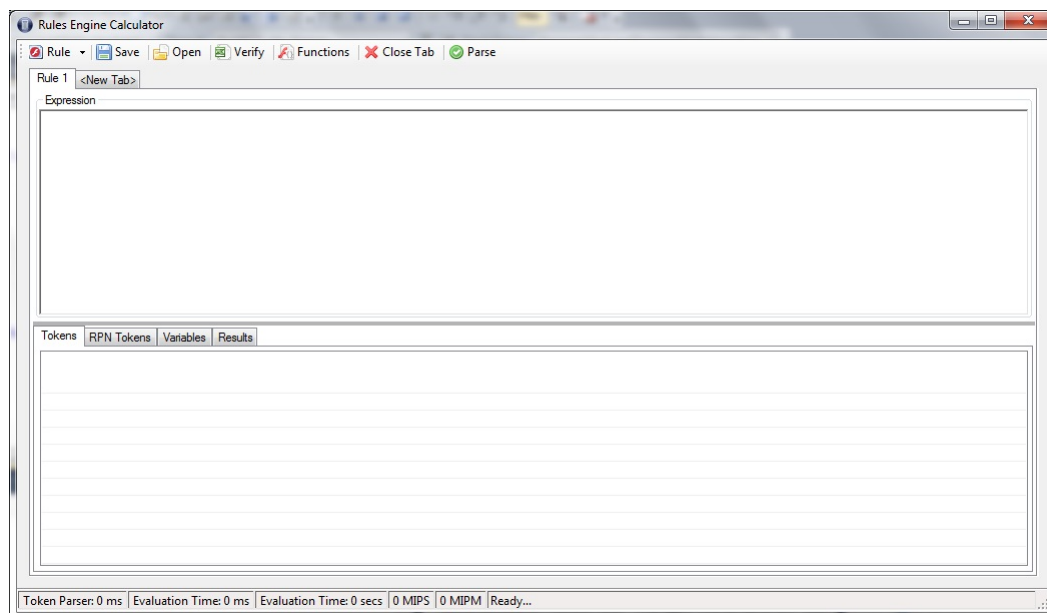


Figure 4.9: Rules Engine Calculator

4.10.2 Implementing PatientStateStep

The *PatientStateStep* class diagram is shown in Figure 4.10. The class contains one important property: *Criteria* which is an instance of class *ThreeValueCriteria*. As shown in the diagram, the *ThreeValueCriteria* class contains three properties: *Specification*, *List<LetExpression>* and *List<GetDataTask>*. The *Specification* property is a string containing the specification that will determine if the patient state step applies. In actual practice, this would be an expression using a syntax such as GELLO. The *List<LetExpression>* contains a representation of all macros (each known as a “Let Expression”) used. The *List<GetDataTask>* property is a list of all the data access tasks associated with this activity.

4.10.3 Implementing ActionStep

An *Action Step* represents the actions to be taken by the execution engine. An action could involve fetching data from an EMR, or requesting input from an end user.

²The host and workflow are executed on the same thread.

³The author of the Evaluation Engine is Donald Snowdy. The code and documentation is available on CodeProject at <http://www.codeproject.com/search.aspx?q=evaluation+enginesbo=kwx=0y=0> and is licensed under the The Creative Commons Attribution-ShareAlike 2.5 License.

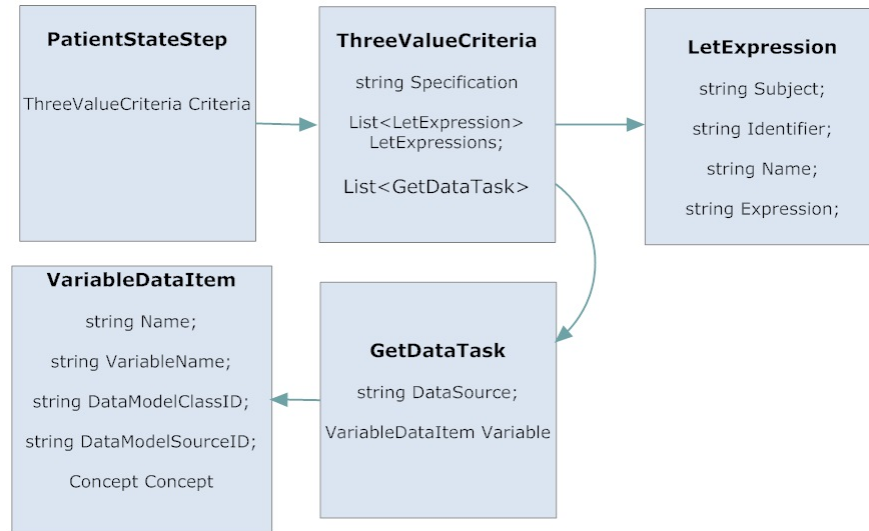


Figure 4.10: PatientStateStep

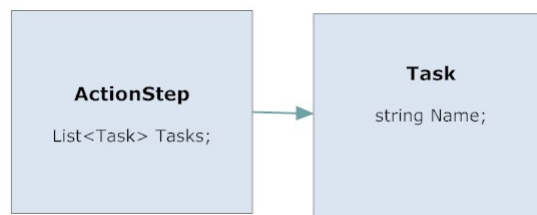


Figure 4.11: ActionStep

The class diagram for an *ActionStep* is shown in Figure 4.11.

The *ActionStep* class contains a list of *Task* objects. The *Task* class is the base class for the *GetDataTask*, the *MedicalActionTask*, the *MessageTask*, the *SubguidelineTask* and the *AssignmentTask*. Other tasks can be added easily by inheriting from *Task*.

The *Task* class diagram is shown in Figure 4.12.

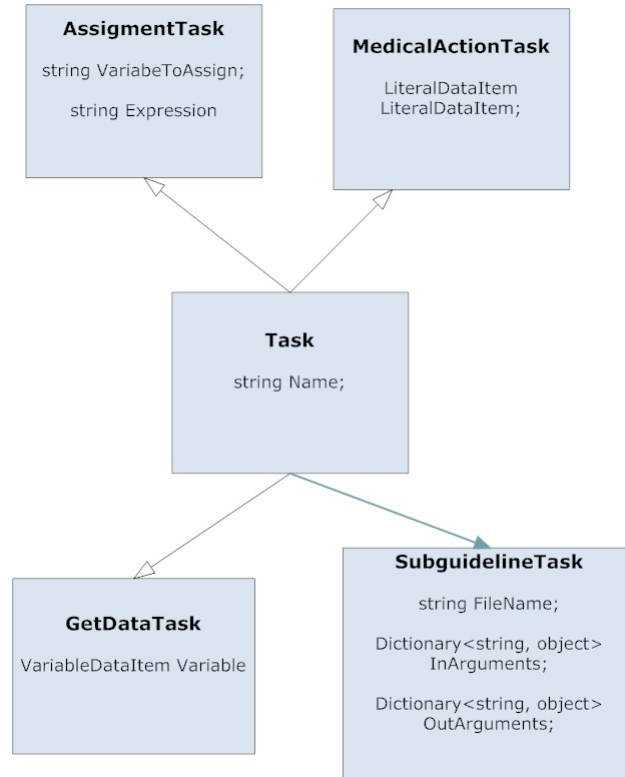


Figure 4.12: Task Diagram

GetDataTask

The *GetDataTask* retrieves data from an external source such as an Electronic Medical Record.

Assignment Task

The assignment task is used to assign a value to a variable. For example, the string “true” could be assigned to a variable called “treated” to indicate that a patient was treated.

MedicalActionTask

The *MedicalActionTask* performs any task that requires reference to medical information. For example, a *MedicalActionTask* could order a particular medication indicated by a UMLS concept unique identifier (CUI).

SubguidelineTask

The *SubguidelineTask* invokes another workflow. Parameters are passed to the workflow invoked via the `InArgument<>` property. Parameters are returned to the calling workflow via the `OutArgument<>` property.

4.10.4 DecisionStep

The decision step brought out many challenges in this project. The only type of decision presently supported is the boolean decision (true/false).

Implementation Challenges

The WF 4.0 framework includes a class called *FlowDecision* that implements a boolean decision. *FlowDecision* determines which `FlowNode` to follow based on the result of a property called `Condition`. To implement a *Decision Step*, additional functionality is needed from the *FlowDecision* class. For example, a *ChoiceStep* requires user input. The *FlowDecision* class (or a class derived therefrom) would ideally fetch the required user input.

Unfortunately the authors of the WF 4 *FlowDecision* decided to seal the class meaning that it cannot be extended. It is very challenging to develop a custom `FlowNode` to replicate the behaviour. Existing designers for activities do not allow multiple out paths. A substitute for *FlowDecision* could not be found in the public domain nor could a solution be crowd sourced for a fee. A workaround solution had to be sought.

The solution selected involved adding code to the base class of the custom activities - namely *BaseActivity* - to inspect the next activity to see if it is a *FlowDecision* and execute custom code before the *FlowDecision* is executed. The solution is not ideal and could cause errors in real world implementation. Another *FlowDecision* type of activity will need to be developed.

A further challenge is storing relevant data about each *DecisionStep*. Since properties cannot be added directly to a *FlowDecision*, a workaround was sought. The solution was to save an XML file under a unique file name in a specified directory. Before each *FlowDecision* is executed, the XML file would be located and parsed to obtain relevant information.

XML File for FlowDecision

Details about each *FlowDecision* are provided in an XML format. At execution time, the preceding activity “sniffs” the next node and executes a method called `HandleFlowDecision()` if the next step is a *FlowDecision*. For this method to work, each *FlowDecision* needs to have a unique identifier to serve as the file name. Unfortunately, the *FlowDecision* class does not provide a unique identifier. The class provides a hash function, but the hash code could differ with each execution. The *FlowDecision* does however have a property called `DisplayName` which represents the text that is printed on the designer canvas on top of the *FlowDecision* node. The `DisplayName` property was used as the basis for a unique identifier.⁴ It is assumed that no two *FlowDecision* instances will be have the same `DisplayName` property.

FlowDecision.Condition Property

The `FlowDecision.Condition` property refers to the condition that is executed by the runtime to determine which path of execution to take. The `Condition` property uses Visual Studio expressions. This implementation uses an expression evaluation engine obtained from the public domain (discussed earlier). The `FlowDecision.Condition` property was not used to evaluate expressions. Rather, the expression engine from the public domain was used.

In order for the workflow to run, the `FlowDecision.Condition` property must be set. Prior to running, a temporary variable called “result” was created. Each `FlowDecision.Condition` gets a unique “result” variable. For example, if there are ten `FlowDecisions`, each will have a `Condition` property beginning with “result” and an integer from 0 to 9.

The *BaseActivity* class sets the “result” parameter based upon the result of executing the decision. For example, if the decision is a “choice” type, the user is queried. If the user selects “yes” or “true”, the “result” property is set to true, so that the *FlowDecision* will follow the “true” branch.

Before the workflow is run, the host application needs to do some pre-processing. The host application first needs to find the *ModelItem* instance representing the *Flowchart* object.⁵

Figure 4.13 shows how to find the *ModelItem* referring to the *Flowchart* reference.

⁴All white spaces and special characters were removed first.

⁵At the time of preprocessing, the workflow is not running. Changes to the workflow need to be made by iterating through *ModelItem* instances.

```

// find ModelItem for Flowchart reference
ModelItem fcMI = null;
foreach (ModelItem m in ms.Find(ms.Root, typeof(object)))
{
    if (m.GetCurrentValue().GetType() == typeof(Flowchart))
    {
        fcMI = m;
        break;
    }
}

```

Figure 4.13: Find Flowchart Reference

The next step is to delete all temporary variables that may have been created during a previous execution of the workflow. Recall that for each *FlowDecision*, the property *FlowDecision.Condition* is set to a boolean called *result* with a unique number identifying the temporary variable. For example, if there are two *FlowDecision* steps, the code will create *result0* and *result1*.

The code to delete all temporary variables is shown in Figure 4.14.

```

// delete temporary variables
foreach (ModelProperty p in fcMI.Properties)
{
    if (p.Name == "Variables")
    {
        if (p.Value != null)
        {
            ModelItemCollection coll = p.Value as ModelItemCollection;
            if (coll != null)
            {
                List<ModelItem> itemsToRemove = new List<ModelItem>();
                foreach (ModelItem micoll in coll)
                {
                    foreach (ModelProperty mmp in micoll.Properties)
                    {
                        if (mmp.Name == "Name")
                        {
                            if (mmp.Value.ToString().Contains("result"))
                                itemsToRemove.Add(micoll);
                        }
                    }
                }
                foreach (ModelItem itemToRemove in itemsToRemove)
                {
                    coll.Remove(itemToRemove);
                }
                p.SetValue(coll);
            }
        }
    }
}

```

Figure 4.14: Deleting Temporary Variables

The final preprocessing step is to set all temporary variables.


```

int valueref = 0;

foreach (ModelItem m in ms.Find(ms.Root, typeof(object)))
{
    // found flowdecision
    if (m.GetCurrentValue().GetType() == typeof(FlowDecision))
    {
        // add variable to Flowchart
        Variable<bool> result = new Variable<bool>();
        // find flowchart reference
        foreach (ModelProperty p in fcMI.Properties)
        {
            if (p.Name == "Variables")
            {
                ModelItemCollection coll = p.Value as
                    ModelItemCollection;
                result.Name = "result" + valueref;
                valueref++;
                // add to collection
                coll.Add(result);
                // find Condition property of FlowDecision
                foreach (ModelProperty mp in m.Properties)
                {
                    if (mp.Name == "Condition")
                    {
                        VisualBasicValue<bool> vbv = new
                            VisualBasicValue<bool>();
                        vbv.ExpressionText = result.Name;
                        mp.SetValue(vbv);
                    }
                }
            }
        }
    }
}

```

Figure 4.15: Setting Temporary Variables

The code shown in Figure 4.15 iterates through all nodes of the Flowchart and adds a special variable for each FlowDecision.

A further challenge is handling a situation where the *DecisionActivity* is the first step on the workflow, or if there are two DecisionActivity steps next to each other. The solution was to create a *BlankStep* that derives from *BaseActivity* and implements the same behaviour as the *PatientStateStep* and *ActionStep* described earlier. If a workflow starts out with a decision, the blank activity must be added before the decision node. If a workflow involves two consecutive decision steps, a blank activity must be interposed.

4.10.5 Branch and Synchronization Steps

The Branch and Synchronization steps of the GLIF framework were not implemented.

Branch Step

C# .NET provides a class called *ParallelActivity*. The class does not execute the child activities each on a separate thread of execution unless the activities are written to support asynchronous execution

Synchronization

The Branch Step is implemented as a *CompositeActivity*. The next activity is not executed until the *ParallelActivity* completes.

Conclusion

Implementing Branch and Synchronization Steps can be implemented in a future release of the execution engine.

4.10.6 Parsing RDF

One important task for interoperability with existing GLIF3 guidelines is to properly parse an RDF file. GLIF is typically authored using Protege and exported to Resource Description Framework (RDF) format. RDF is a specification for conceptual description of information and is used typically on the World Wide Web.

An excerpt of the RDF produced for a *PatientStateStep* in GLIF (from the cough guideline) is shown below.

```
<kb:Patient_State_Step rdf:about="&kb;CoughStudy_00005"
  kb:display_name="Chronic Cough"
  kb:name="chronic cough in immunocompetent adults"
  kb:new_encounter="false"
  rdfs:label="Chronic Cough">
<kb:next_step rdf:resource="&kb;CoughStudy_00010"/>
<kb:patient_state_description rdf:resource="&kb;CoughStudy_00046"/>
</kb:Patient_State_Step>
```

RDF at its core is a set of triples consisting of subject, object and predicate. In the triple above, the subject is shown in the about attribute - namely *CoughStudy_00005*. The triple contains several predicates: *display_name*, *name*, *new_encounter*, *label*, *next_step* and *patient_state_description*.

An RDF parser from the Internet was used to break RDF documents down into the component triples⁶.

The Figure below shows the stages of parsing an RDF guideline.

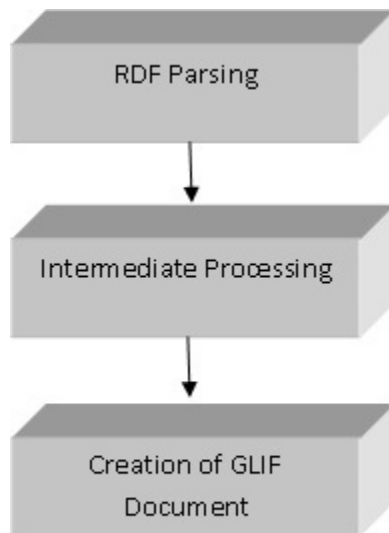


Figure 4.16: Parsing Stages

The first stage is to break down the RDF document. The import algorithm begins by iterating through each triple in the RDF document and creating a representation of the document in code. The second stage iterates through the representation created during the first stage and creates *intermediate objects*. The purpose of the intermediate object is to link all related triples together. For example, a PatientStateStep contains a list of data items that will be required. The third stage involves creating an object oriented representation of the guideline using the classes in the GLIFObjects namespace (and assembly). Finally, the objects created are sent to the host to be converted to a Flowchart with the relevant steps.

One challenge is creating an algorithm that will arrange the Flowchart in a neat logical manner. The Flowchart created from parsing an RDF document is jumbled and requires effort to rearrange so that the Flowchart is visually appealing.

⁶See <http://www.dotnetrdf.org/api/VDS.RDF.Parsing.html> for the RDF parser. I would like to thank the authors for developing this useful tool.

Chapter 5

Authoring a Guideline in Workflow Foundation

5.1 The Guideline

In this chapter, a guideline called *Group A Streptococcal Disease Surveillance Protocol for Ontario Hospitals* (the “protocol”) is implemented.

Group A Streptococcus (GAS) is a bacterium commonly found in the throat and on the skin. GAS is spread by direct contact with mucus from the nose or throat of infected persons, or through contact with the skin.

Most people who acquire GAS do not have serious complications; however, for some patients, GAS is a severe life threatening disease. Invasive GAS occurs when the bacteria gets into certain parts of the body such as blood, muscle or lung tissue. Two invasive forms of GAS are *necrotizing fasciitis* and *streptococcal toxic shock syndrome*. Necrotizing fasciitis destroys muscles, fat and skin tissue. Streptococcal toxic shock syndrome causes blood pressure to drop and organs to fail.

The *Group A Streptococcal Disease Surveillance Protocol for Ontario Hospitals* (the “protocol”) was developed to provide direction to hospitals to prevent transmission of GAS among health-care workers and patients. The authors provide a decision tree in Appendix A of the protocol to outline recommended management for health care workers (HCW) epidemiologically linked to patient cases and colonized with Group A Streptococcus (GAS).

The protocol involves two major stages: treatment of the health-care worker with medication, and subsequent monitoring with possible treatment of household contacts.

The computer implemented guideline uses a main guideline with a subguideline. The subguideline was created to deal specifically with the medical therapy of GAS. The protocol provides for several potential medication regimens (chemoprophylaxis) based on a number of variables such as the site of the infection, whether the carrier is pregnant or lactating and whether the carrier is allergic to penicillin. The subguideline could also be changed without impacting the main guideline should treatment options change.

The steps to build the subguideline are described first.

5.1.1 Subguideline - Treatment of GAS

A subguideline was used to encapsulate the logic for prescribing a medical treatment regimen. The subguideline can be modified without the need to modify the main guideline. For example, if medical research suggests a new factor to consider when prescribing a medication, the subguideline alone can be changed.

The first step to implementing the protocol is to make explicit the medical regimens. Currently, the regimen for a patient who is not allergic to penicillin and who is neither pregnant nor lactating is Penicillin V plus rifampin. The dosage is 500 milligrams of Penicillin V four times daily for ten days, plus 300 milligrams of Rifampin twice daily for the last four days.

For patients who are pregnant or lactating, the protocol recommends Cephalexin 250 milligram tablets taken four times daily for ten days. The penicillin V plus rifampin regimen is not recommended in this case because rifampin has been found to be teratogenic in laboratory animals and may also interfere with oral contraceptives[2].

For patients who are rectal carriers of GAS, the protocol recommends Clindamycin 300 milligram tablets taken orally three times daily for ten days. Clindamycin is acceptable for persons who are allergic to penicillin[2]. An alternative is 500 mg of Azithromycin taken once daily for five days.

To be useful, a guideline must be specific when suggesting a medication. UMLS concept unique identifiers (CUI) were used for this purpose. CUI codes were located for each of the five medications referred to in the guideline. A guideline must also be precise when describing when a patient needs to begin and end a medication. An expression language would be used for this purpose.

The medications, concept unique identifiers, daily doses, start time and end times are shown in the table below:

Concept	CUI	# Daily	Start	End
Cephalexin 250 mg oral tablet	C0975442	4	now	now + 10
Penicillin V 500 mg oral tablet	C0689921	4	now	now + 10
Rifampin 300 mg oral tablet	C1704481	2	now + 6	now + 10
Clindamycin 300 mg oral tablet	C0706942	3	now	now + 10
Azithromycin 500 mg oral tablet	C1126333	1	now	now + 5

5.1.2 Creating the Subguideline

The first step is to create a new document by clicking File, hovering over New and selecting Guideline. A blank FlowChart opens in the drawing canvas.

The first step in the guideline is to ask the user if the health care worker is pregnant or lactating in which case the guideline would suggest Cephalexin which is preferred for pregnant or lactating health care workers. Due to the limitations for decision branches explained earlier, it is necessary to insert a BlankStep. Drag a BlankStep underneath the start node, right click the BlankStep and select “Set as Start Node”. A line will connect the start node to the blank node.

Next add a DecisionStep. Connect the BlankStep to the DecisionStep that was just added. Right click the DecisionStep and complete the form as shown in Figure below.

The screenshot shows a dialog box titled 'frmEditDecisionStep'. It has the following fields and controls:

- Type of Decision Step:** A dropdown menu set to 'choice'.
- Display Name:** A text box containing 'pregnantorlactating'.
- Query:** A text box containing 'is the health care worker pregnant or lactating?'. To its right are two buttons labeled 'true' and 'false'.
- Rule In | Rule Out:** Two tabs, with 'Rule In' currently selected.
- Specification:** A table with a header row labeled 'Specification' and several empty rows below it.
- OK:** A button at the bottom right of the dialog.

Figure 5.1: Decision Step - Is HWC Pregnant or Lactating?

The next step is to handle the two possibilities: true or false. Drag an ActionStep underneath the DecisionStep but slightly to the left. Connect the ActionStep to the left side of the DecisionStep (indicating “true”). Right click the ActionStep.

The ActionStep form will open up. Click the “Add Medical Action” button. The Medical Oriented Task form will appear. Complete the form as shown in Figure 5.2.

The screenshot shows a window titled "frmEditMedicalOrientedTask" with the following fields and values:

- Name: Prescribe Cephalexin
- Data Model:
 - Data Model Class ID: Medication
 - Data Model Source ID: HL7 3.0 RIM
- Concept:
 - Concept ID: C0975442
 - Concept Source ID: UMLS
- Dosage Quantity: 250 mg 4 times daily
- Route: po
- Mood: order
- Critical Time:
 - Start: now
 - End: now + 10

An "OK" button is located at the bottom right of the form.

Figure 5.2: Medical Action Task - Prescribe Cephalexin

The Data Model Class ID refers to “Medication” because the data described by this medical action refers to medication. The Data Model Source ID refers to HL7 3.0 RIM indicating that the Medication model is found in the Reference Information Model (RIM). The Concept ID and Concept Source ID refer to the medication and the source (or dictionary) describing the ID. In this case, C0975442 of UMLS describes Cephalexin.

The dosage quality refers to the amount of medication to be taken and how often. The route refers to how the medication is taken. For example, “po” means orally. The “mood” is stating that the medication is being ordered. A “mood” code could also tell the practitioner to instruct a patient to stop a particular medication. The Start and End critical times provide information as to when a medication should be started and finished. In this case, the medication should be started now and completed in ten days.

The subguideline is designed to output a true or false value to the calling guideline. To accomplish this, add an OutArgument to the workflow. Click the “Arguments” tab below the canvas and add a string argument called “treated” as shown in Figure 5.3.

The final chart for the subguideline is shown in Figure 5.4.

Note that the “treated” variable is set at the end of the workflow. The variable could have been set after each medication recommendation, but the user of the guideline may

not accept the recommendation, or may prescribe a different medication. As a result, the user is asked later if the health care worker was treated.

Name	Direction	Argument type	Default value
treated	Out	String	<i>Default value not supported</i>

Create Argument

Variables Arguments Imports

100%

Figure 5.3: Adding an OutArgument

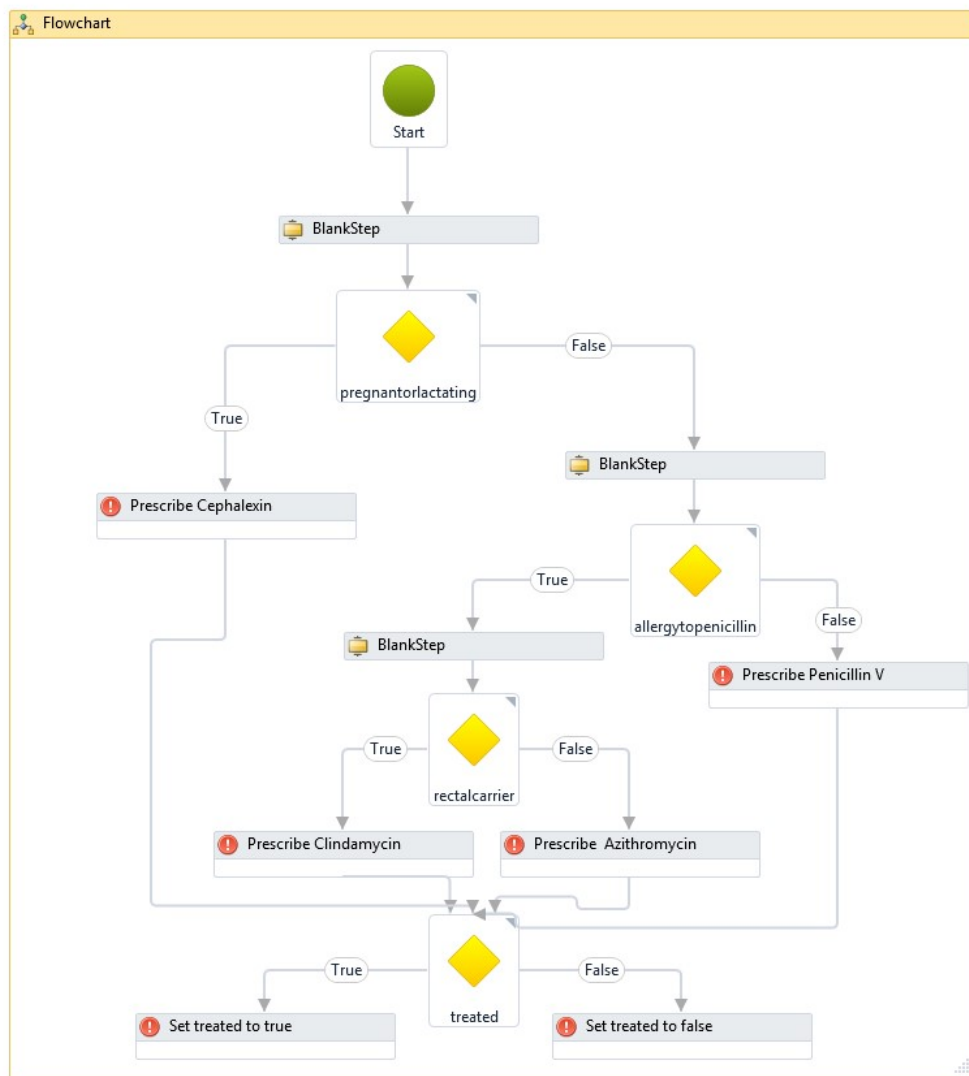


Figure 5.4: Treatment of GAS Subguideline

Once the subguideline is created, it is necessary to save it in XAML format to be executed later. Click File on the file menu and select Save, then file a suitable path and name the file “subguideline.xaml”.

5.1.3 Implementing the Main Guideline

To implement the main guideline, create a new guideline by clicking File on the file menu, hovering over New and clicking Guideline.

A new canvas will open up with a FlowChart and start node.

Add an ActionStep, left click the ActionStep and type “Treat HCW”. Right click this node and select “Start as Start Node” on the context menu. A connector will join the start node to the ActionStep just created.

The main guideline will use a return variable from the subguideline. Click “Variables” and add a boolean variable called “treated.”

Right click the ActionStep and select Properties. A form will open up to edit the ActionStep. On the form, click the “Add Subguideline” button. A form will display prompting for a file name, InArguments and OutArguments. Click the “Get Path” button and select the subguideline created earlier. One OutArgument will be displayed as shown in the Figure below:

The screenshot shows a dialog box titled "frmSubguidelineTask". At the top, there is a "File" field containing the path "G:\subguideline.xaml" and a "Get Path" button. Below this, there are two tables: "In Arguments" and "Out arguments".

Subguideline Variable	Guideline Variable

Subguideline Variable	Guideline Variable
treated	

An "OK" button is located at the bottom right of the dialog box.

Figure 5.5: Adding a Subguideline

The subguideline outputs a string called “treated” with a value of “true” if the patient was treated or a value of “false” if the patient was not treated. Local variable “treated”

needs to be set to the output from the subguideline. Right click “treated” in the OutArgument pane and select “Assign” when the context menu appears. A form will appear as shown below:

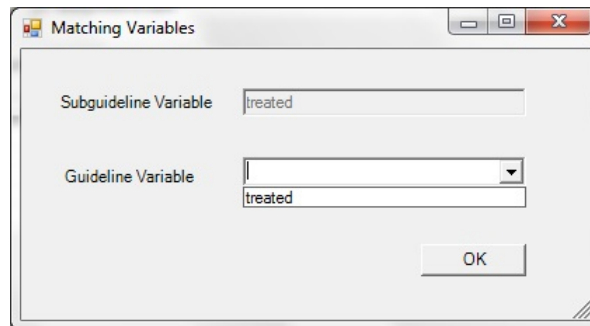


Figure 5.6: Matching Guideline Variables with Subguideline Variables

Select “treated” from the ComboBox for “Guideline Variable” and click “OK.”

Click “OK” to exit the subguideline form and then “OK” again to exit the ActionStep form.

The next step is to create a PatientStateStep to indicate that further steps depend on the health care worker having been treated.

Drag a PatientStateStep on to the canvas and connect it to the previous ActionStep. Right click the PatientStateStep and select Properties. The patient state is indicated by an expression. To indicate that the patient must have been treated, type in “treated = true”, as shown in Figure 5.7.

The next step is to order that the health care worker not work for the next 24 hours. This will be indicated to the user by an ActionStep and MessageTask.

The screenshot shows a window titled "PatientStateStep (Patient_State_Step)". It contains three main sections:

- Specification:** A text box containing the expression "treated = true".
- Let Expressions:** A table with columns "Name", "Identifier", and "Expression". The table is currently empty.
- GetDataActions:** A table with columns "Name", "Variable", "Source", "Class", and "ConceptID". The table is currently empty. An "Add" button is located to the right of this table.

An "OK" button is located at the bottom right of the window.

Figure 5.7: PatientStateStep for treated = true

Drag an ActionStep underneath the previous PatientStateStep and connect it to the PatientStateStep. Right click the new ActionStep and select Properties. Click the “Add Message” button and complete the form as shown in the figure below:

The screenshot shows a window titled "frmEditMessage". It contains two main sections:

- Display Name:** A text box containing the text "Suspect HCW".
- Message:** A text box containing the text "Suspend health care worker (HCW) for 24 hours".

An "OK" button is located at the bottom right of the window.

Figure 5.8: MessageTask - Suspend HCW for 24 Hours

Implementation of the rest of the guideline is not described here. The full guideline is illustrated in Figure 5.9.

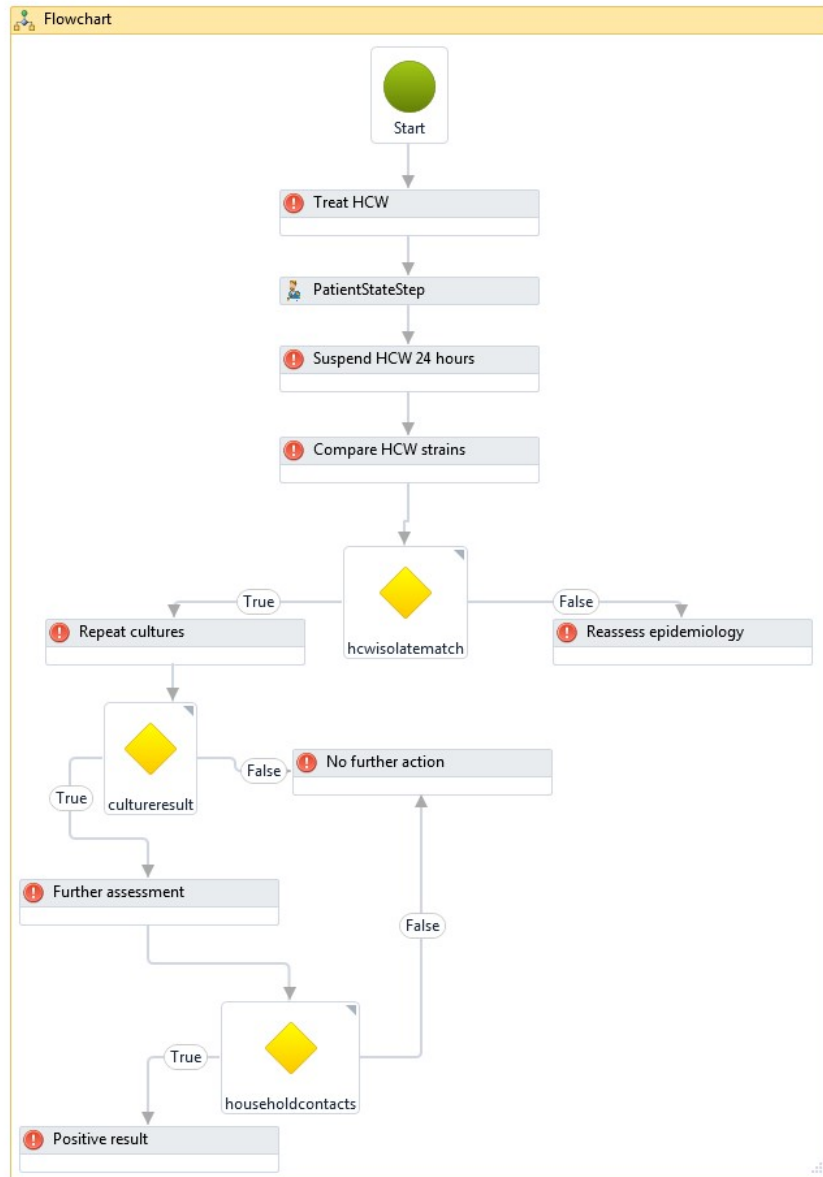


Figure 5.9: Full Main Guideline

5.1.4 Testing the Guideline

Click File and select Run to test the guideline.

A dialogue box will open asking the user whether the patient is pregnant or lactating, as shown in Figure 5.10.

If the user selects “Yes”, the guideline opens up a form to tell the user to prescribe Cephalexin, as shown in Figure 5.11.

If the user selects “No”, the subguideline asks the user if the health care worker is allergic to Penicillin.

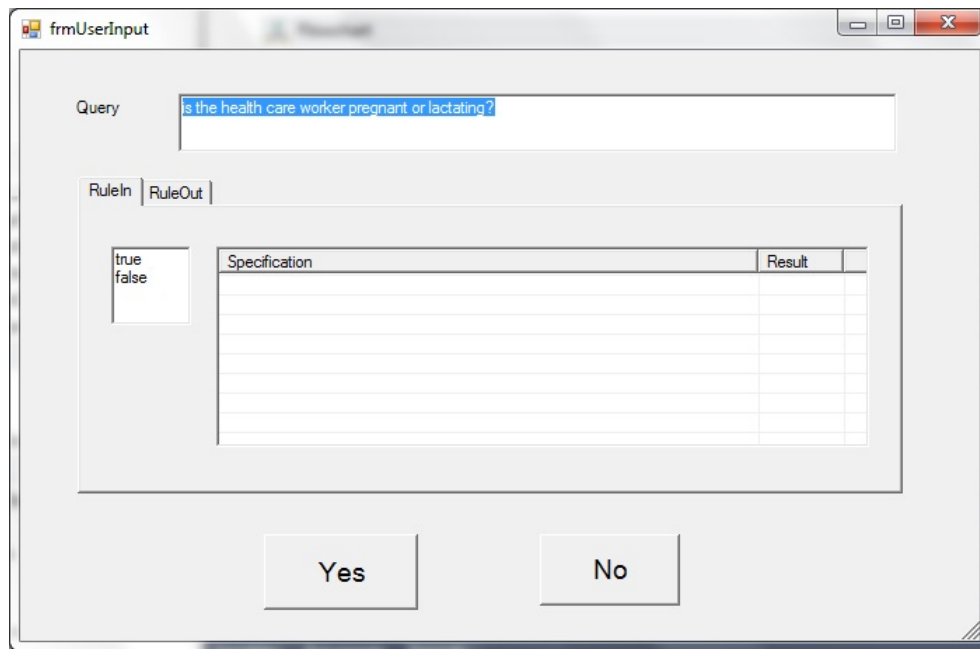
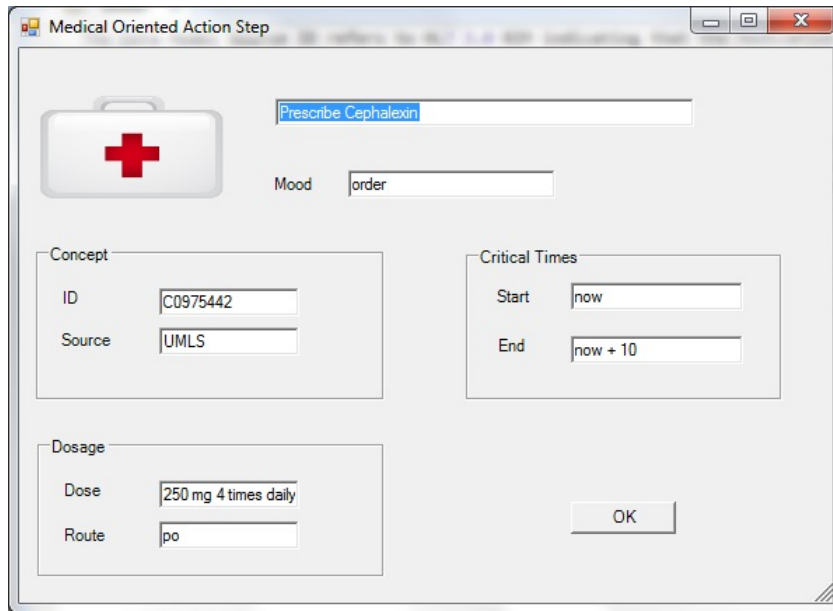


Figure 5.10: Query if Patient is Pregnant or Lactating

Eventually the subguideline will recommend a medication and then ask the user if the patient has been treated. If the patient has been treated, an OutArgument called “treated” is set to “true”; otherwise, it is set to “false.”

The PatientStateStep looks at the “treated” value returned from the subguideline to determine if the guideline can continue. If the patient was treated, a message appears to instruct the user to order that the health care worker stop working for 24 hours.

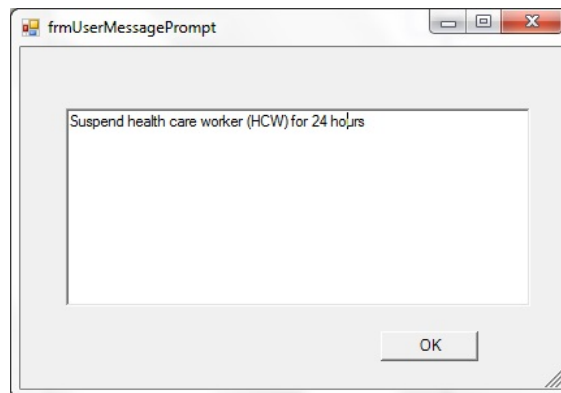


The screenshot shows a window titled "Medical Oriented Action Step" with a red cross icon. The main text is "Prescribe Cephalexin". Below this, there is a "Mood" field with the value "order". The window is divided into three sections: "Concept", "Critical Times", and "Dosage".

Section	Field	Value
Concept	ID	C0975442
	Source	UMLS
Critical Times	Start	now
	End	now + 10
Dosage	Dose	250 mg 4 times daily
	Route	po

An "OK" button is located at the bottom right of the dialog box.

Figure 5.11: MedicallyOrientedActionStep - Prescribe Cephalexin



The screenshot shows a window titled "frmUserMessagePrompt". The main text is "Suspend health care worker (HCW) for 24 ho,rs". An "OK" button is located at the bottom right of the dialog box.

Figure 5.12: Suspend Heath Care Worker for 24 Hours

The guideline continues as instructed by the flow chart in Appendix A of the protocol.

Chapter 6

Conclusion

The research in this paper has demonstrated that Workflow Foundation can be used to implement GLIF. The major computer science obstacle is integration with electronic medical records (EMR). Frameworks such as HL7's RIM and standard vocabularies such as UMLS hold out great promise as far as assisting to implement computer-interpretable guidelines.

GLIF is a task-network model approach to clinical guidelines. It offers a visual approach to designing guidelines. Guidelines are temporally sequenced and unfold over time and are relatively easy to conceptualize. It is hoped that soon there will exist a standard (e.g. HL7) for representing guidelines.

Windows Workflow Foundation (WF 4.0) provides a declarative approach to program design where users can select from a variety of activities to build a workflow. WF 4.0 provides a suitable environment for building a guideline execution engine and executing clinical guidelines. This thesis has provided a foundation for actual implementation of GLIF that can be integrated into an existing health care setting. The techniques in this thesis can be extrapolated to other task-network models.

Bibliography

- [1] Philosophical limits of evidence-based medicine. *Academic Medicine*, 1998.
- [2] Prevention of invasive group a streptococcal disease among household contacts of case patients and among postpartum and postsurgical patients: recommendations from the centers for disease control and prevention. *Clin Infect Dis*, 35(8):950–9, 2002.
- [3] Garg AX, Adhikari NJ, McDonald H, and et al. Effects of computerized clinical decision support systems on practitioner performance and patient outcomes: A systematic review. *JAMA: The Journal of the American Medical Association*, 293(10):1223–1238, 2005.
- [4] G. Ross Baker, William A. Ghali, Philip Hebert, Sumit R. Majumdar, Luz Palacios-derflingher, Robert J. Reid, Sam Sheps, and Robyn Tamblyn. The canadian adverse events study: the incidence of adverse events among hospital patients in canada. *Canadian Medical Association Journal*, 170:1678–1686, 2004.
- [5] Bruce Bukovics. *Pro WF: Windows Workflow in .NET 4*. Apresspod Series. Apress, 2010.
- [6] Canadian Medical Association. *Handbook on Clinical Practice Guidelines*, July 2007.
- [7] ECMA International. *Standard ECMA-334 - C# Language Specification*. 4 edition, June 2006.
- [8] B J Essex. Approach to rapid problem solving in clinical medicine. *BMJ*, 3(5974):34–36, 7 1975.
- [9] Marilyn J. Field and Institute of Medicine Kathleen N. Lohr, Editors; Committee on Clinical Practice Guidelines. *Guidelines for Clinical Practice: From Development to Use*. The National Academies Press, 1992.

- [10] Peter E. Friedland and Yumi Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1:161–208, 1985.
- [11] Robert A. Greens. *Clinical Decision Support: The Road Ahead*. Academic Press, 2007.
- [12] Benson T. R. Spector A. Harrington, J. Ieee p1157 medical data interchange (medix) committee overview and status report. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, pages 230–234, 1990.
- [13] Kensaku Kawamoto, Caitlin A. Houlihan, E. Andrew Balas, and David F. Lobach. Improving clinical practice using clinical decision support systems: a systematic review of trials to identify features critical to success. *BMJ*, 330(7494):765+, April 2005.
- [14] Janet M. Corrigan Linda T. Kohn and Institute of Medicine Molla S. Donaldson, Editors; Committee on Quality of Health Care in America. *To Err Is Human: Building a Safer Health System*. The National Academies Press, 2000.
- [15] NAHIT. Defining key health information technology terms. Report to the office of the national coordinator for health information technology, National Alliance for Health Information Technology, April 2008.
- [16] Vimla L. Patel, Vanessa G. Allen, Jose F. Arocha, and Edward H. Shortliffe. Representing clinical guidelines in glif: Individual and collaborative expertise, 1998.
- [17] Mor Peleg. Cough guideline model in glif. <http://mis.hevra.haifa.ac.il/morpeleg/Intermed/guidelines/ExampleCough.pdf>. Accessed: 2014-04-02.
- [18] Mor Peleg, Samson Tu, Jonathan Bury, Paolo Ciccarese, John Fox, Robert A Greens, Silvia Miksch, Silvana Quaglini, Andreas Seyfang, Edward H Shortliffe, Mario Stefanelli, and et al. Comparing computer-interpretable guideline models: A case-study approach. *JAMIA*, 10:2003, 2003.
- [19] Mor et al. Peleg. Guideline interchange format 3.4 technical specification. Technical report, InterMed Collaboratory, 2001.
- [20] Dianne Miller Wolman Sheldon Greenfield Robin Graham, Michelle Mancher and Editors; Committee on Standards for Developing Trustworthy Clinical Practice Guidelines; Institute of Medicine Earl Steinberg. *Clinical Practice Guidelines We Can Trust*. The National Academies Press, 2011.

- [21] David L Sackett, William M C Rosenberg, J A Muir Gray, R Brian Haynes, and W Scott Richardson. Evidence based medicine: what it is and what it isn't. *BMJ*, 312(7023):71–72, 1 1996.
- [22] Matthias Samwald, Karsten Fehre, Jeroen S. de Bruin, and Klaus-Peter Adlassnig. The arden syntax standard for clinical decision support: Experiences and directions. *Journal of Biomedical Informatics*, 45(4):711–718, 2012.
- [23] Gunther Schadow, Daniel Russler, Charles Mead, and Clement McDonald. Integrating Medical Information and Knowledge in the HL7 RIM. pages 764–768, Indianapolis, IN, 2000.
- [24] R.N. Shiffman. Representation of clinical practice guidelines in conventional and augmented decision tables. *J Am Med Inform Assoc*, 4(5):382–93.
- [25] Samson W. Tu and Mark A. Musen. A flexible approach to guideline modeling. pages 420–424, Washington DC, 1999.
- [26] Steven H Woolf, Richard Grol, Allen Hutchinson, Martin Eccles, and Jeremy Grimshaw. Potential benefits, limitations, and harms of clinical guidelines. *BMJ*, 318(7182):527–530, 2 1999.

```

using System.Activities;
using System.Activities.Statements;
using System;
using System.Activities.Hosting;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Activities.Presentation.Model;
using System.Xml;
using IHostCommunication;
using System.Xml.Linq;
using System.ComponentModel;
using Microsoft.VisualBasic.Activities;

namespace ActivityLibrary
{
    public class BaseActivity : NativeActivity
    {
        {
            public class WorkflowInstanceInfo : IWorkflowInstanceExtension
            {
                public IEnumerable<object> GetAdditionalExtensions() {
                    yield break;
                }

                public void SetInstance(WorkflowInstanceProxy instance) {
                    this.proxy = instance;
                }

                WorkflowInstanceProxy proxy;

                public WorkflowInstanceProxy GetProxy() { return proxy; }
            }

            protected IHostCommunication.IHostMessaging host;
            protected IHostCommunication.IHostDataRequest requestdata;
            protected WorkflowDataContext dataContext;
            protected WorkflowInstanceProxy proxy;

            public virtual GLIFObjects.BaseStep CreateStepWithoutLinkages()
            {
                throw new Exception("CreateStep must be overridden");
            }

            protected override void Execute(NativeActivityContext context)
            {
            }

            protected void HandleFlowDecision() {
                FlowDecision fd = IsNextStepFlowDecision();
                if(fd != null) {
                    VisualBasicValue<bool> vbv = fd.Condition as VisualBasicValue<bool>;
                    string id = host.FetchDisplayName(vbv.ExpressionText);
                    UserInput userInput = new UserInput();
                    List<RuleInCondition> ruleInConditions = new List<RuleInCondition>();

                    // fetch file
                    XmlDocument doc = new XmlDocument();
                    doc.Load(Environment.GetEnvironmentVariable("decisionpath") + id + ".xml");

                    // get type
                    XmlNodeList type = doc.GetElementsByTagName("Type");
                    if (type.Count == 0)
                        throw new ArgumentException("Attempt to parse decision file with no type property");
                    // choice step
                    if (type[0].InnerText == "choice") {
                        XmlNodeList text = doc.GetElementsByTagName("Text");
                        userInput.Text = text[0].InnerText;
                    }
                }
            }
        }
    }
}

```

```

XmlNodeList rulesIn = doc.GetElementsByTagName("rulein");
// set ruleinconditions
foreach (XmlNode rule in rulesIn)
{
    RuleInCondition newRule = new RuleInCondition();

    // get child elements of rule
    foreach (XmlNode child in rule.ChildNodes) {

        if (child.Name == "specification")
            newRule.ThreeValueCriteria.Specification = child.InnerText;
        else if (child.Name == "truefalse") {
            bool result = false;
            if (child.InnerText == "True")
                result = true;
            newRule.TrueFalse = result;
        }
    }
    ruleInConditions.Add(newRule);
}

// do rule conditions (if any)
host.SendMessageToConsole("Evaluating rule conditions");
host.SendMessageToConsole("-----");
host.SendMessageToConsole("# RuleIn conditions " + ruleInConditions.Count);
// add rule conditions

foreach (RuleInCondition rule in ruleInConditions) {
    string evalRule = EvaluatorWrapper.GetExpressionString(rule.ThreeValueCriteria.
Specification, dataContext);
    EvaluationEngine.Parser.Token token = new EvaluationEngine.Parser.Token(evalRule);
    host.SendMessageToConsole("Evaluating " + evalRule);

    // add variables to evaluator
    // note that evaluation engine does not consider DateTime to
    // be variable
    foreach (PropertyDescriptor pd in dataContext.GetProperties()) {
        if (pd.GetValue(dataContext) == null)
            token.Variables[pd.Name].VariableValue = string.Empty;
        else
        {
            if (pd.GetValue(dataContext).GetType() != typeof(DateTime)) {
                token.Variables.Add(pd.Name);
                token.Variables[pd.Name].VariableValue = pd.GetValue(dataContext).ToString();
            }
        }
    }
}

EvaluationEngine.Evaluate.Evaluator eval = new EvaluationEngine.Evaluate.Evaluator
(token);

// run the evaluation
string ErrorMsg = "";
string result = "";
// fetch result
try
{
    if (!eval.Evaluate(out result, out ErrorMsg))
        host.SendMessageToConsole("Rule " + evalRule +
            "evaluates to " + result);
}

catch (Exception e)
{
    MessageBox.Show(e.InnerException.ToString());
}

```

```

        finally
        {
            IHostCommunication.RuleInCondition rulein = new IHostCommunication.RuleInCondition()
;
            rulein.TrueFalse = rule.TrueFalse;
            rulein.Specification = rule.ThreeValueCriteria.Specification;
            rulein.Result = result;
            userInput.RuleInConditions.Add(rulein);
        }
    }

    requestdata.SendUserInputBack += new EventHandler<BooleanResponseEventArgs>(BooleanResponse);
    requestdata.RequestUserInput(userInput);
    requestdata.SendUserInputBack -= BooleanResponse;
}

private void BooleanResponse(object sender, BooleanResponseEventArgs args) {
    // find result variable
    FlowDecision fd = IsNextStepFlowDecision();
    string expressiontext = string.Empty;

    if (fd.Condition is VisualBasicValue<bool>) {
        VisualBasicValue<bool> vbv = fd.Condition as VisualBasicValue<bool>;
        expressiontext = vbv.ExpressionText;
    }

    foreach (PropertyDescriptor pd in dataContext.GetProperties()) {
        if (pd.Name == expressiontext)
            pd.SetValue(dataContext, args.Result);
    }
}

private FlowDecision IsNextStepFlowDecision() {
    FlowDecision result = null;
    Activity root = proxy.WorkflowDefinition;
    Flowchart fc = null;

    foreach (Activity act in WorkflowInspectionServices.GetActivities(root)) {
        if (act is Flowchart) {
            fc = act as Flowchart;
            break;
        }
    }

    foreach (FlowNode node in fc.Nodes) {
        if (node is FlowStep)
        {
            FlowStep fs = node as FlowStep;

            if (fs.Action == this) {
                if (fs.Next != null && fs.Next is FlowDecision) {
                    FlowDecision fd = fs.Next as FlowDecision;
                    result = fd;
                }
            }
        }
    }

    return result;
}
}
}

```

C:\Users\Ryan\Desktop\WF 4.0 examples\chapter 16\ActivityLibrary\PatientStateStep.cs

1

```
using System.Activities;
using System.Windows;
using System.ComponentModel;
using System.ComponentModel.Design.Serialization;
using ActivityLibrary.Design;
using IHostCommunication;
using System.Windows.Documents;
using System.Collections.Generic;
using System;
using System.Activities.Statements;
using System.Activities.Presentation.Services;
using System.Activities.Presentation.Model;
using System.Reflection;
using System.Activities.Hosting;

namespace ActivityLibrary
{
    [Designer(typeof(PatientStateStepDesigner))]
    public class PatientStateStep:BaseActivity
    {
        private ThreeValueCriteria criteria = null;
        public ThreeValueCriteria Criteria {
            get {if (criteria == null)
                criteria = new ThreeValueCriteria();
                return criteria;
            } set
            {
                criteria = value; }
        }

        protected override void CacheMetadata(NativeActivityMetadata metadata) {
            metadata.AddDefaultExtensionProvider<WorkflowInstanceInfo>(() => new WorkflowInstanceInfo());
            base.CacheMetadata(metadata);
        }

        protected override void Execute(NativeActivityContext context)
        {
            // get data context for binding variables
            dataContext = context.DataContext;
            proxy = context.GetExtension<WorkflowInstanceInfo>().GetProxy();

            // get host communication service for communication with host app
            // (debugging output etc)
            host = context.GetExtension<IHostCommunication.IHostMessaging>();
            requestdata = context.GetExtension<IHostCommunication.IHostDataRequest>();

            // connect method to handle response from host
            requestdata.SendDataBack += new EventHandler<ResponseDataEventArgs>(ResponseHandler);

            // update console that we are entering patient state step
            OutputMessageToConsole("Entered patient state step " + this.DisplayName);
            // output specification to console
            OutputMessageToConsole("Specification is " + Criteria.Specification);
            // evaluate let expressions (macros)
            OutputMessageToConsole("Number of let expressions " + Criteria.LetExpressions.Count.ToString());

            string expression = criteria.Specification;

            // do let expression substitutions
            foreach (LetExpression le in Criteria.LetExpressions) {
                OutputMessageToConsole("Replacing " + le.Identifier + " with " +
                    le.Expression);
                expression = expression.Replace(le.Identifier, le.Expression);
            }

            OutputMessageToConsole("Resulting expression " + expression);

            // fetch required data
```

```

    OutputMessageToConsole("Fetching data");
    // create list of data
    List<IHostCommunication.DataItem> ItemsToFetch = new List<IHostCommunication.DataItem>();

    foreach (GetDataTask task in Criteria.GetDataTasks) {
        OutputMessageToConsole("Fetching " + task.Variable.VariableName);
        IHostCommunication.DataItem item = new IHostCommunication.DataItem();
        item.Name = task.Name;
        item.VariableName = task.Variable.VariableName;
        item.CurrentValue = FetchCurrentValue(item.VariableName);
        ItemsToFetch.Add(item);
    }

    requestdata.RequestData(this.DisplayName, ItemsToFetch);

    // evaluate result
    // tokenize specification
    // preprocessing non strings
    string rule = EvaluatorWrapper.GetExpressionString(Criteria.Specification, dataContext);
    EvaluationEngine.Parser.Token token = new EvaluationEngine.Parser.Token(rule);
    OutputMessageToConsole("Evaluating " + rule);

    // add variables to evaluator
    // note that evaluation engine does not consider DateTime to
    // be variable
    foreach (PropertyDescriptor pd in dataContext.GetProperties()) {
        if (pd.GetValue(dataContext) == null)
        {
            token.Variables.Add(pd.Name);
            token.Variables[pd.Name].VariableValue = string.Empty;
        }
        else
        {
            if (pd.GetValue(dataContext).GetType() != typeof(DateTime))
            {
                token.Variables.Add(pd.Name);
                token.Variables[pd.Name].VariableValue = pd.GetValue(dataContext).ToString();
            }
        }
    }

    EvaluationEngine.Evaluate.Evaluator eval =
        new EvaluationEngine.Evaluate.Evaluator(token);

    // run the evaluation
    string errorMsg = "";
    string result = "";
    // fetch result
    if (!eval.Evaluate(out result, out errorMsg))
        MessageBox.Show(errorMsg);

    if (result.Trim().ToLower() != "true") {
        MessageBoxResult oride = MessageBox.Show("Patient state not met. Do you wish to override?",
        "Patient State Step Not Met", MessageBoxButton.YesNo, MessageBoxImage.Question);
        if (oride == MessageBoxResult.No) {
            host.Terminate("Patient statecondition not met.");
        }
    }

    // if next step is flow decision, this activity handles the
    // processing (ugly workaround)
    HandleFlowDecision();
}

private string FetchCurrentValue(string variableName) {
    foreach (PropertyDescriptor pi in dataContext.GetProperties()) {
        if (pi.Name == variableName) {
            if (pi.GetValue(dataContext) != null)

```

```

        return pi.GetValue(dataContext).ToString();
    else
        return string.Empty;
    }
}

throw new Exception("Variable not found.");
}

private void ResponseHandler(object sender, ResponseDataEventArgs args) {
    // iterate through all values to be added
    foreach (KeyValuePair<string, string> itemsToChange in args.ValuesChanged) {
        // find variable
        foreach (PropertyDescriptor pd in dataContext.GetProperties())
        {
            // found
            if(pd.Name == itemsToChange.Key) {
                if (pd.PropertyType == typeof(DateTime))
                {
                    DateTime newDT = DateTime.Parse(itemsToChange.Value);
                    pd.SetValue(dataContext, newDT);
                    OutputMessageToConsole("Changed " + itemsToChange.Key + " to "
                        + newDT.ToLongDateString());
                }
                else
                {
                    pd.SetValue(dataContext, itemsToChange.Value);
                    OutputMessageToConsole("Changed " + itemsToChange.Key + " to "
                        + itemsToChange.Value);
                }
            }
        }
    }
}

private void OutputMessageToConsole(string message) {
    if (host != null)
        host.SendMessageToConsole(message);
}

private void OnComplete(NativeActivityContext context, ActivityInstance completedInstance) {
    // unsubscribe for events
    requestdata.SendDataBack -= new EventHandler<ResponseDataEventArgs>(ResponseHandler);
}

public PatientStateStep Clone() {
    PatientStateStep pss = new PatientStateStep();
    pss.Criteria = this.Criteria.Clone();
    return pss;
}

public override GLIFObjects.BaseStep CreateStepWithoutLinkages() {
    GLIFObjects.PatientStateStep pss = new GLIFObjects.PatientStateStep();
    pss.DisplayName = DisplayName;
    pss.Criteria.Specification = this.Criteria.Specification;

    foreach (ActivityLibrary.GetDataTask task in this.Criteria.GetDataTasks) {
        GLIFObjects.GetDataAction gda = new GLIFObjects.GetDataAction();
        gda.VariableName = task.Variable.VariableName;
        pss.Criteria.GetDataActions.Add(gda);
    }
    return pss;
}
}
}
}

```


C:\Users\Ryan\Desktop\WF 4.0 examples\chapter 16\ActivityLibrary\ActionStep.cs

1

```

using System.Collections.Generic;
using System.ComponentModel;
using ActivityLibrary.Design;
using System.Windows.Forms;
using System.Activities;
using IHostCommunication;
using System;
using System.Activities.XamlIntegration;
using Microsoft.VisualBasic.Activities;
using System.Reflection;

namespace ActivityLibrary
{
    [Designer(typeof(ActionStepDesigner))]
    public class ActionStep:BaseActivity
    {
        private List<Task> tasks;
        public List<Task> Tasks {
            get {
                return tasks;
            }
            set
            {
                if (value == null)
                    throw new ArgumentNullException("Tasks cannot be null");
                tasks = value;
            }
        }

        public ActionStep() {
            tasks = new List<Task>();
        }

        protected override void CacheMetadata(NativeActivityMetadata metadata) {
            metadata.AddDefaultExtensionProvider<WorkflowInstanceInfo>(() => new WorkflowInstanceInfo());
            base.CacheMetadata(metadata);
        }

        protected override void Execute(System.Activities.NativeActivityContext context) {
            // get data context for binding variables
            dataContext = context.DataContext;
            proxy = context.GetExtension<WorkflowInstanceInfo>().GetProxy();
            // get host communication service for communication with host app
            // (debugging output etc)
            host = context.GetExtension<IHostCommunication.IHostMessaging>();
            requestdata = context.GetExtension<IHostCommunication.IHostDataRequest>();
            // connect method to handle response from host
            requestdata.SendDataBack += new EventHandler<ResponseDataEventArgs>(ResponseHandler);
            // update console that we are entering action step
            OutputMessageToConsole("Entered action step " + this.DisplayName);
            // output number of asks for debugging
            OutputMessageToConsole("Number of tasks is " + Tasks.Count.ToString());

            // iterate through data tasks and medical oriented tasks
            List<IHostCommunication.DataItem> ItemsToFetch = new List<IHostCommunication.DataItem>();
            List<MedicalActionSpecification> MedicalOrientedTasks = new List<MedicalActionSpecification>();
            List<AssignmentTask> Assignments = new List<AssignmentTask>();
            List<SubguidelineTask> Subguidelines = new List<SubguidelineTask>();

            foreach (Task task in this.Tasks) {
                if(task is GetDataTask) {
                    OutputMessageToConsole("Fetching " + ((GetDataTask)task).Variable.VariableName);
                    IHostCommunication.DataItem item = new IHostCommunication.DataItem();
                    item.Name = ((GetDataTask)task).Name;
                    item.VariableName = ((GetDataTask)task).Variable.VariableName;
                    item.DataSource = ((GetDataTask)task).Variable.DataModelSourceID;
                    item.CurrentValue = FetchCurrentValue(item.VariableName);
                    ItemsToFetch.Add(item);
                }
            }
        }
    }
}

```

```

    }
    else if(task is MedicalActionTask) {
        // populate list of medical tasks
        MedicalActionSpecification specification = new MedicalActionSpecification();
        specification.DisplayText = ((MedicalActionTask)task).Name;
        specification.Concept.ID = ((MedicalActionTask)task).LiteralDataItem.Concept.ID;
        specification.Concept.Source = ((MedicalActionTask)task).LiteralDataItem.Concept.Source;

        if (((MedicalActionTask)task).LiteralDataItem.Value is Medication) {
            Medication med = (Medication)((MedicalActionTask)task).LiteralDataItem.Value;
            specification.Mood = med.Mood;
            specification.Route = med.Route;
            specification.DosageQuantity = med.DosageQuantity;
            specification.CriticalStart = med.CriticalTime.Start.DisplayName;
            specification.CriticalEnd = med.CriticalTime.End.DisplayName;
        }

        MedicalOrientedTasks.Add(specification);
    }
    else if (task is AssignmentTask) {
        Assignments.Add( (AssignmentTask)task);
    }
    else if (task is SubguidelineTask) {
        Subguidelines.Add((SubguidelineTask)task);
    }
    else if (task is MessageTask) {
        host.SendMessageToUser(((MessageTask)task).Message);
    }
}

// fetch data if there is any to be fetched
if(ItemsToFetch.Count > 0)
    requestdata.RequestData(this.DisplayName, ItemsToFetch);
// do assignments (if any)
foreach (AssignmentTask assignment in Assignments)
{
    // create evaluate engine
    EvaluationEngine.Parser.Token token = new EvaluationEngine.Parser.Token(assignment.Expression);

    // set variables
    foreach (PropertyDescriptor pd in dataContext.GetProperties()) {
        token.Variables.Add(pd.Name);
        if (pd.GetValue(dataContext) == null)
            token.Variables[pd.Name].VariableValue = string.Empty;
        else
            token.Variables[pd.Name].VariableValue = pd.GetValue(dataContext).ToString();
    }

    EvaluationEngine.Evaluate.Evaluator eval =
        new EvaluationEngine.Evaluate.Evaluator(token);

    // run the evaluation
    string errorMsg = "";
    string result = "";
    // fetch result
    eval.Evaluate(out result, out errorMsg);

    // find variable for assignment
    foreach (PropertyDescriptor pd in dataContext.GetProperties()) {
        if (pd.Name == assignment.VariableToAssign) {
            // is it date time?
            if (pd.PropertyType == typeof(DateTime))
            {
                // fetch date time
                DateTime dtResult = DateTime.Parse(result);
                pd.SetValue(dataContext, dtResult);
            }
        }
    }
    else

```

```

        pd.SetValue(dataContext, result);
        break;
    }
}

// output debugging info to console
host.SendMessageToConsole("Assignment task");
host.SendMessageToConsole(assignment.Expression);
host.SendMessageToConsole("Evaluates to " + result);
}

// do medical oriented action steps if any
foreach (MedicalActionSpecification medicalTask in MedicalOrientedTasks) {
    host.SendMedicalInformationTask(medicalTask);
}

// do subguidelines
foreach (SubguidelineTask subguideline in Subguidelines) {
    Activity activity = ActivityXamlServices.Load(subguideline.FileName);
    Dictionary<string, object> inparameters = new Dictionary<string,object>();

    foreach (SubguidelineArgument arg in subguideline.InArguments) {
        inparameters.Add(arg.GuidelineVariable, arg.SubguidelineVariable);
    }

    host.SendMessageToConsole("Invoking subguideline");
    // copy existing display names
    Dictionary<string, string> existingDisplayNames = host.GetDisplayNames();
    // clear display names out of host communicator for subguideline
    // (to be returned later)
    host.ClearDisplayNames();

    // handle FlowDecision Steps
    Activity sub = ActivityXamlServices.Load(subguideline.FileName);
    foreach (Activity act in WorkflowInspectionServices.GetActivities(sub))
    {
        if (act is System.Activities.Statements.Flowchart)
        {
            List<Variable> itemsToDelete = new List<Variable>();

            System.Activities.Statements.Flowchart fc = act as System.Activities.Statements.
Flowchart;
            foreach (Variable var in fc.Variables) {
                if (var.Name.StartsWith("result")) {
                    itemsToDelete.Add(var);
                }
            }

            // delete temp variables
            foreach (Variable var in itemsToDelete)
                fc.Variables.Remove(var);

            // add temp variables
            int valueref = 0;

            // iterate and find FlowDecision steps
            foreach (System.Activities.Statements.FlowNode fn in ((System.Activities.Statements.
Flowchart) act ).Nodes )
            {
                if (fn is System.Activities.Statements.FlowDecision)
                {
                    System.Activities.Statements.FlowDecision fd =
                        fn as System.Activities.Statements.FlowDecision;
                    // add boolean variable
                    Variable<bool> newVar = new Variable<bool>();
                    newVar.Name = "result"+valueref;
                    valueref++;
                    ((System.Activities.Statements.Flowchart)act).Variables.Add(newVar);
                    VisualBasicValue<bool> vbv = new VisualBasicValue<bool>();

```

```

        vbv.ExpressionText = newVar.Name;
        fd.Condition = vbv;
        // get DisplayName for FlowDecision
        string displayName = string.Empty;
        Type fdType = fd.GetType();
        foreach (PropertyInfo pi in fdType.GetProperties())
        {
            if(pi.Name == "DisplayName") {
                displayName = (string)pi.GetValue(fd, null);
                break;
            }
        }

        displayName = displayName.Trim();
        displayName = displayName.Replace(" ", string.Empty);
        host.SetDisplayName(newVar.Name, displayName);
    }
}

// invoke sub workflow
WorkflowInvoker inv = new WorkflowInvoker(activity);
inv.Extensions.Add(host);
inv.Extensions.Add(requestdata);
IDictionary<string, object> outarguments = inv.Invoke(inparameters);

// set result (outarguments)
foreach (KeyValuePair<string, object> kvp in outarguments)
{
    foreach (Activity act in WorkflowInspectionServices.GetActivities(sub))
    {
        if (act is System.Activities.Statements.Flowchart)
        {
            System.Activities.Statements.Flowchart fc = act as
                System.Activities.Statements.Flowchart;

            // find variable
            foreach (SubguidelineArgument s in subguideline.OutArguments)
            {
                // match variable to assign
                if (s.SubguidelineVariable == kvp.Key)
                {
                    foreach (PropertyDescriptor pd in dataContext.GetProperties())
                    {
                        if (pd.Name == s.GuidelineVariable)
                        {
                            host.SendMessageToConsole("Setting variable " + s.GuidelineVariable +
                                + " to " +
                                    + kvp.Value);
                            pd.SetValue(dataContext, kvp.Value);
                        }
                    }
                }
            }

            /*
            // found
            foreach (Variable v in fc.Variables)
            {
                // found guideline variable
                if (v.Name == s.GuidelineVariable) {
                    v.Set(context, kvp.Value);
                    host.SendMessageToConsole("Set variable " +
                        s.GuidelineVariable + " to " +
                            kvp.Value);
                }
            }
            */

```

```

        }
        */
    }
}
foreach (Variable v in fc.Variables)
{

}

}

break;

}

}

// reset display names
host.ClearDisplayNames();

foreach (KeyValuePair<string, string> kvp in existingDisplayNames)
    host.SetDisplayName(kvp.Key, kvp.Value);
}

HandleFlowDecision();
}

private void OutputMessageToConsole(string message) {
    if (host != null)
        host.SendMessageToConsole(message);
}

private string FetchCurrentValue(string variableName) {
    foreach (PropertyDescriptor pi in dataContext.GetProperties()) {
        if (pi.Name == variableName)
        {
            if (pi.GetValue(dataContext) != null)
                return pi.GetValue(dataContext).ToString();
            else
                return string.Empty;
        }
    }
    throw new Exception("Variable not found.");
}

private void ResponseHandler(object sender, ResponseDataEventArgs args) {
    // iterate through all values to be added
    foreach (KeyValuePair<string, string> itemsToChange in args.ValuesChanged)
    {
        // find variable
        foreach (PropertyDescriptor pd in dataContext.GetProperties())
        {
            // found
            if (pd.Name == itemsToChange.Key)
            {
                if (pd.PropertyType == typeof(DateTime)) {
                    DateTime result = DateTime.Parse(itemsToChange.Value);
                    pd.SetValue(dataContext, result);
                    OutputMessageToConsole("Changed " + itemsToChange.Key + " to "
                        + result.ToLongDateString());
                }
                else
                {
                    pd.SetValue(dataContext, itemsToChange.Value);
                    OutputMessageToConsole("Changed " + itemsToChange.Key + " to "

```

```
        + itemsToChange.Value);  
    }  
} } } } }
```