Faculdade de Engenharia da Universidade do Porto

Mestrado Integrado em Engenharia Informática e Computação

MIEIC 2007 Masters Project Report

Clinical Protocols Enabling Evidence Based Medicine Practice in Healthcare Software Solutions

Hugo André Amaral Rodrigues

Supervisor at FEUP: Prof. António Fernando Vasconcelos Cunha Castro Coelho

April 2008

NOTE: In accordance with the terms of the internship protocol and the confidentiality agreement executed with Alert Life Sciences Computing, S.A. ("Alert"), this report is confidential and may contain references to inventions, know-how, drawings, computer software, trade secrets, products, formulas, methods, plans, specifications, projects, data or works protected by Alert's industrial and/or intellectual property rights. This report may be used solely for research and educational purposes. Any other kind of use requires prior written consent from Alert.

To my parents.

Abstract

Evidence based medicine (EBM) is an attempt to combine individual clinical expertise with the best available external evidence in order to provide a faster, more accurate and effective healthcare. The medical community interest on EBM has been growing increasingly, even though there isn't a consensus on the benefits of its use.

Clinical protocols are a form of EBM practice, consisting on decision trees built after solid clinical evidence, with the purpose of providing pathways for treatment, diagnosis and prevention of certain pathologies.

The project, entitled "Clinical Protocols Enabling Evidence Based Medicine Practice in Healthcare Software Solutions", aimed at creating support for building and using clinical protocols within existing healthcare software. This report describes the analysis of existing solutions in the area, the study of the problem premises and implementation details for the proposed solution.

Alert® clinical software was used as the basis for the implementation of a protocols module, which will be made available as a core component, meaning it will be shared by all Alert® products. The final results met the initial expectations and a stable tool has been implemented and deployed. This work is therefore not only a proof of concept, but represents a great contribution to Alert®, and may impact healthcare environments worldwide.

This project serves as the final project for the Masters degree in Informatics and Computing Engineering, at Oporto University's Faculty of Engineering (Faculdade de Engenharia da Universidade do Porto).

Table of Contents

1. Introduction 2
1.1. Enterprise Context 3
1.2. Problem Contextualization....6
1.3. Planning....7

2. Problem Analysis....8
2.1. Evidence Based Medicine (EBM)....8
2.2. Clinical protocols 11
2.3. Objectives....12
2.4. Technological background....13

3. State Of The Art....20
3.1. Computer interpretable protocol representations....20
3.2. Visualization tools 26
3.3. Overview 37

4. Solution – Clinical Protocols Module in Alert® 38
4.1. Requirements....39
4.2. Development Methodologies....44
4.3. Context of the developed work....47
4.4. Flowchart Development....48
4.5. Protocols Tool Development....58
4.6. Testing and debugging 68

5. Conclusions and Future Work 69
5.1. Results 69
5.2. Project overview and highlights....70
5.3. Future work....73

References....75

Glossary / Index of Terms....77

Image Index

Figure 1: Alert software in use at a healthcare unit. 3
Figure 2: The Alert® Suite. 4
Figure 3: Project Planning - Gantt Diagram. 7
Figure 4: Clinical expertise.(5) 9
Figure 5: Data flow in Evidence Based Medicine.(adapted from (6)) 9
Figure 6: Data flow in Evidence Based Medicine, adapted from (5). 9
Figure 7: An example protocol. 11
Figure 8: Alert® tiered structure. 13
Figure 9: XRay interface. 19
Figure 10: Service capture interface – viewing input/output variables. 19
Figure 11: Protocol representation languages and respective visualization tools. 20
Figure 12: The PROforma class model. 21
Figure 13: Task sub-types in PROforma. 22
Figure 14: Task state transitions. 22
Figure 15: Arezzo modules. 27
Figure 16: Arezzo Performer – Arezzo's graphical view of the cough guideline encoding in PROforma. 28
Figure 17: Tallis Composer – building a guideline flow diagram. 29
Figure 18: Tallis Composer – defining task attributes. 30
Figure 19: Tallis Tester – Interface components. 31
Figure 20: Tallis Engine – customizable front-end. 32
Figure 21: AsbruView - Topological View. 33
Figure 22: AsbruView – Timeline View. 33
Figure 23: AsbruView user interface. 34
Figure 24: CareVis – Logical View. 35
Figure 25: CareVis – Temporal View. 35
Figure 26: CareVis – user interface. 36
Figure 27: Proposed solution 38
Figure 28: Protocols Use Case Diagram. 43
Figure 29: Product Life Cycle. 44
Figure 30: Development workflow for the 3 software layers. 45
Figure 31: Global product development process – Evolution Model. 46
Figure 32: Content project development process – modified Waterfall Model. 47
Figure 33: Context of the project. 48
Figure 34: Flowchart components interaction. 50
Figure 35: Box component. 51
Figure 36: Connectors. 53
Figure 37: The connector hit area. 53
Figure 38: Stage component. 55
Figure 39: Stage with some Boxes and Connectors in place. 57
Figure 40: ZoomBar component. 57
Figure 41: Alert® package diagram. 58

Figure 42: Alert® screen components. 59
Figure 43: Flash Authoring Application: the Object Library. 60
Figure 44: Flash Authoring Application : layers. 60
Figure 45: Compiling Flash files. 62
Figure 46: ProtocolElement components. 64
Figure 47: Sequence diagram illustrating use of local protocol data across screens. 65
Figure 48: Protocol saving workflow. 66
Figure 49: Applying a protocol to a patient. 67
Figure 50: Connector routing improvements. 73

Table Index

Table 1: Alert® Products 5
Table 2: Attributes of the generic task. 23
Table 3: Flowchart components requirements 40
Table 4: Protocols tool requirements. 41
Table 5: Box parameters. 52
Table 6: Box element events. 52
Table 7: Connector parameters. 53
Table 8: Connector element events. 54
Table 9: Stage parameters. 56
Table 10: Stage element events. 56
Table 11: ZoomBar parameters. 58
Table 12: ZoomBar events. 58
Table 13: Protocol parameters. 63
Table 14: ProtocolElement parameters. 64
Table 15: ProtocolElement events. 64
Table 16: Technologies comparison. 69

1. Introduction

"If physiology literally means 'the logic of life', and pathology is 'the logic of disease', then health informatics is the logic of healthcare. It is the rational study of the way we think about patients, and the way that treatments are defined, selected and evolved. It is the study of how clinical knowledge is created, shaped, shared and applied. (…) It is likely that in the next century, the study of informatics will become as fundamental to the practice of medicine as anatomy has been to the last." (1)

Being able to maintain an integrated healthcare software solution, while keeping up with constant evolution in knowledge and technology is a major challenge. Solid medical operational tools already exist, that are custom-shaped for each clinical department and enable inter-department collaboration, and focus is now set on content: content management, content centralization, content sharing and collaborative content. Organizing information and assuring its quality are key factors to guarantee efficient and quality health care.

The project described in this report represents part of this effort to create reliable content and make it accessible to the medical community. By gathering the best available evidence and organizing it in the form of protocols, it's also an innovative way to "*think about patients, and the way that treatments are defined, selected and evolved"*.

This project is in line with the work previously described on the report ***Decision Support Systems – Integrating Guidelines and Bibliography*** (2), and included the analysis and implementation of a Protocols module as an important Decision Support System.

This introductory section provides a general context for the research activities developed and for the implementation of the proposed solution. Section 1.1 describes the "host" company profile, and section 0 provides a contextualization of the problem within the commercialized software. Section 1.3 describes the project timeline and planned activities.

1.1. Enterprise Context

Alert Life Sciences Computing S.A. (Alert LSC) is currently mainly dedicated to the development of clinical software solutions, to create paper-free environments within clinical facilities. With a 100% per year growth, both in profits and in staff, the company currently employs over 400 permanent collaborators, including medical staff, designers, architects engineers and mathematicians, among others.

Alert LSC's attained success comes from the innovative nature of the products it commercializes within the different healthcare departments, which recently started to follow the global tendency to automate healthcare services. It's the first Portuguese organization that managed to create innovative solutions for Emergency departments, having trademarked several brands it uses.

The company is undergoing an internationalization process, which is only possible due to the success obtained in Portuguese territory. It's also currently investing in diversified fields such as research, academic support and data warehouse technologies.

1.1.1. The Alert® Suite

The Alert® software suite is a clinical software solution that provides a 100% paper free environment with intuitive features and functionalities. This paper free solution can be installed throughout an entire healthcare facility or individual department.

Alert® was conceived to be used with touch-screen monitors (Figure 1). Its design is customized to each user profile (doctor, nurse, ancillary personnel, information desk, lab or imaging technician, department manager, social worker, administrative personnel) and interconnects the activities of all healthcare professionals through workflow concepts.

Figure 1: Alert software in use at a healthcare unit.

The application suite began as an operational tool, but grew into content and artificial intelligence, positioning itself as an enabler of healthcare. Different products exist for distinct purposes and healthcare departments. The same *look & feel* is assured for all the products, and there are functionalities which are common to some or all of them, but each product is adapted to the needs of the specific environment in which it will be used.

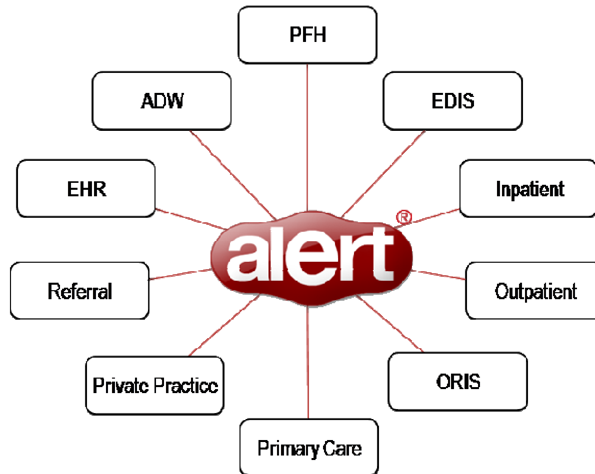

Figure 2: The Alert® Suite.

As shown on Figure 2, the Alert® Suite comprises the products summarized on Table 1.

Table 1: Alert® Products

Product	Description
Alert® PFH (Paper Free Hospital)	A solution to computerize entire hospitals making it possible to document, integrate and review all information relating to hospital operations. Increases efficiency and eases management and cost control tasks.
Alert® EDIS (Emergency Department Information System)	Complete solution for emergency departments, used to document, review and integrate all clinical information from emergency department episodes.
Alert® Inpatient	A clinical and management software for hospital Inpatient Units, used to document and share information immediately, avoiding legibility and communication problems.
Alert® ORIS (Operating Room Information System)	Allows for the complete computerization of operating rooms. Increases operating room efficiency throughout the pre-operative, peri-operative and post-operative periods. Information is available immediately and can be shared with other users, avoiding legibility and communication problems.
Alert® Outpatient	Clinical and management software for Outpatient units, facilitates the management of staff, processes and technical resources.
Alert® Primary Care	Solution for the computerization of Primary Care Centers. It's an efficient communication system between different healthcare services, making it possible to access clinical information collected in Primary Care Centers, Hospitals and other facilities.
Alert® Private Practice	Software solution for the computerization of private Physician Offices and Outpatient Clinics. Makes it possible to document and review each individual patient record, including information collected at other facilities.
Alert® Referral	Integrated computer system that creates a medical information network between healthcare facilities.
Alert® EHR (Electronic Health Record)	Registers, archives and interconnects each patient's clinical information, including that originating from other applications and entities, integrating the clinical history of each patient.
Alert® Data Warehouse (ADW)	Archives and analyzes clinical and operational data. Alert® Data Warehouse (ADW) can perform complex queries, analysis and reporting, with effective control of data access. Information within ADW, can be used to detect trends and identify patterns making it a valuable support for accurate interpretation of events within a clinical environment. Information presented in ADW reports statistics on data obtained through Alert® applications.

1.2. Problem Contextualization

Evidence-based medicine aims at the prevention, diagnosis and treatment of diseases using medical evidence (the concept is further explored on section 2.1). Integration of external evidence-based data sources into the existing clinical information system may allow the definition of appropriate therapy alternatives for a given patient and a given disease. This is however a major challenge, not feasible without the aid of IT.

Alert® works as the IT infrastructure for this project. Clinical protocols come in as a decision support system that relies on evidence-based information, and play an increasingly important role in the field of health care.

The project aimed at the development of a tool to allow the creation and use of protocols within Alert® as a Decision Support System. The work is in line with the modules developed for the project **Decision Support Systems in Alert® – Integrating Guidelines and Bibliography** (2), previously accomplished during a curricular internship.

From the user interface perspective, this module can be considered a more complete and complex version of Alert® Guidelines, sharing some features and background concepts, but taking them one step further. The use of clinical protocols by healthcare professionals is not something new or even recent, but their robustness and applicability is evolving rapidly. It's mostly the valuable combination of clinical expertise with external evidence that grants a vital importance to protocols as decision support systems.

This work featured implementation of a protocol creation tool, but most importantly its integration as part of the Alert® products. The tool will be made available to the 6 major Alert® products:

- Alert® **Care**
- Alert® **EDIS**
- Alert® **Inpatient**
- Alert® **ORIS**
- Alert® **Outpatient**
- Alert® **Private Practice**

Protocols can be shared by different Alert® products within an institution, as well as among medical staff, improving collaboration.

The protocol creation and edition functionality will be accessible from each professional's personal definitions area, commonly called *BackOffice*, while protocol application will be part of the patients' health record.

1.3. Planning

The project planning defined two distinct phases, being that step 2 is dependent on the infrastructure built on step 1.

1. Requirements analysis and implementation of the flowchart components
2. Requirements analysis for the protocols module
    a. Analysis and implementation of protocol creation functionality
    b. Analysis and implementation of protocol application functionality

A plan of action was defined for the implementation of the new tool, estimating task times, milestones and deliverables. This analysis has been condensed on the Gantt diagram on Figure 3.

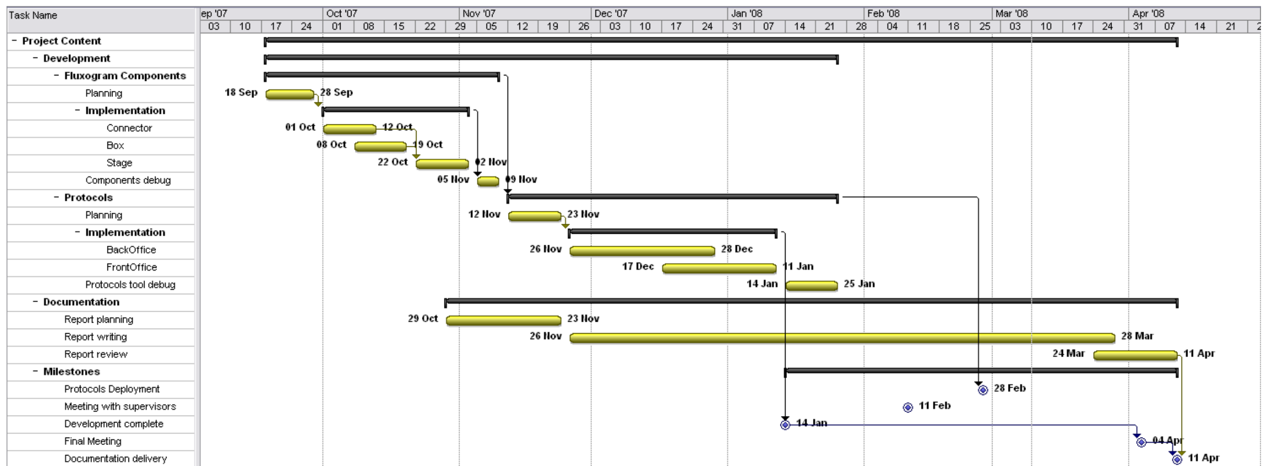

Figure 3: Project Planning - Gantt Diagram.

2. Problem Analysis

This section presents the actual problem in analysis, exposing the relevant concepts and underlying theory. In order to understand clinical protocols, and their importance, it's important to know the context in which they exist: a method of medicine practice known as Evidence Based Medicine (EBM). Section 2.1 describes EBM general concepts and characteristics to some detail, while clinical protocols in particular are analyzed on section 2.2.

The objectives proposed for the final solution are listed on section 2.3. Because the technologies that will be used for implementation are defined from the start (they are bounded by the framework in use by Alert®), a contextualization in terms of background architecture and technologies is also provided on section 2.4 (Technological background).

2.1. Evidence Based Medicine (EBM)

EBM is a scientific medicine attempt to apply more uniformly the standards of evidence[1] gained from the scientific method to certain aspects of medical practice. Specifically, EBM seeks to assess the quality of evidence relevant to the risks and benefits of treatments (including lack of treatment) (3).

"EBM is the conscientious, explicit, and judicious use of current best evidence in making decisions about the care of individual patients" (4), using techniques from science, engineering, and statistics, such as meta-analysis of medical literature, risk-benefit analysis, and randomized controlled trials. It allows integration of individual clinical expertise with the best available external clinical evidence from systematic research.

Individual clinical expertise comprises the proficiency and judgment that individual clinicians acquire through clinical experience and clinical practice (Figure 4). Increased expertise is reflected in many ways, but especially in more effective and efficient diagnosis and in the more thoughtful identification and compassionate use of individual patients' predicaments, rights, and preferences in making clinical decisions about their care.

[1] Evidence gained from scientific method can be understood as rules that are believed to be true, given the strong proof that supports them, based on scientific studies.

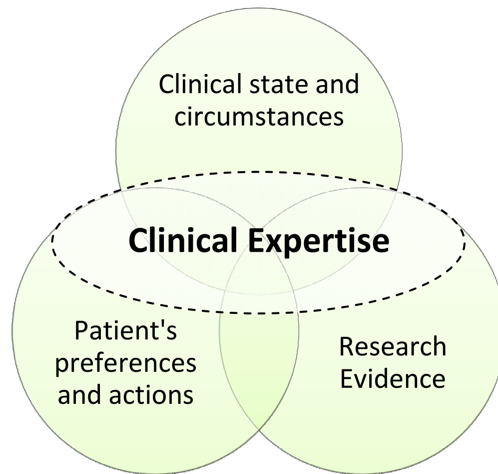

Figure 4: Clinical expertise.(5)

Best available external clinical evidence includes all clinically relevant research (Figure 5), often from the basic sciences of medicine, but especially from patient centered clinical research into the accuracy and precision of diagnostic tests (including the clinical examination), the power of prognostic markers, and the efficacy and safety of therapeutic, rehabilitative, and preventive regimens. External clinical evidence both invalidates previously accepted diagnostic tests and treatments and replaces them with new ones that are more powerful, more accurate, more efficacious, and safer.

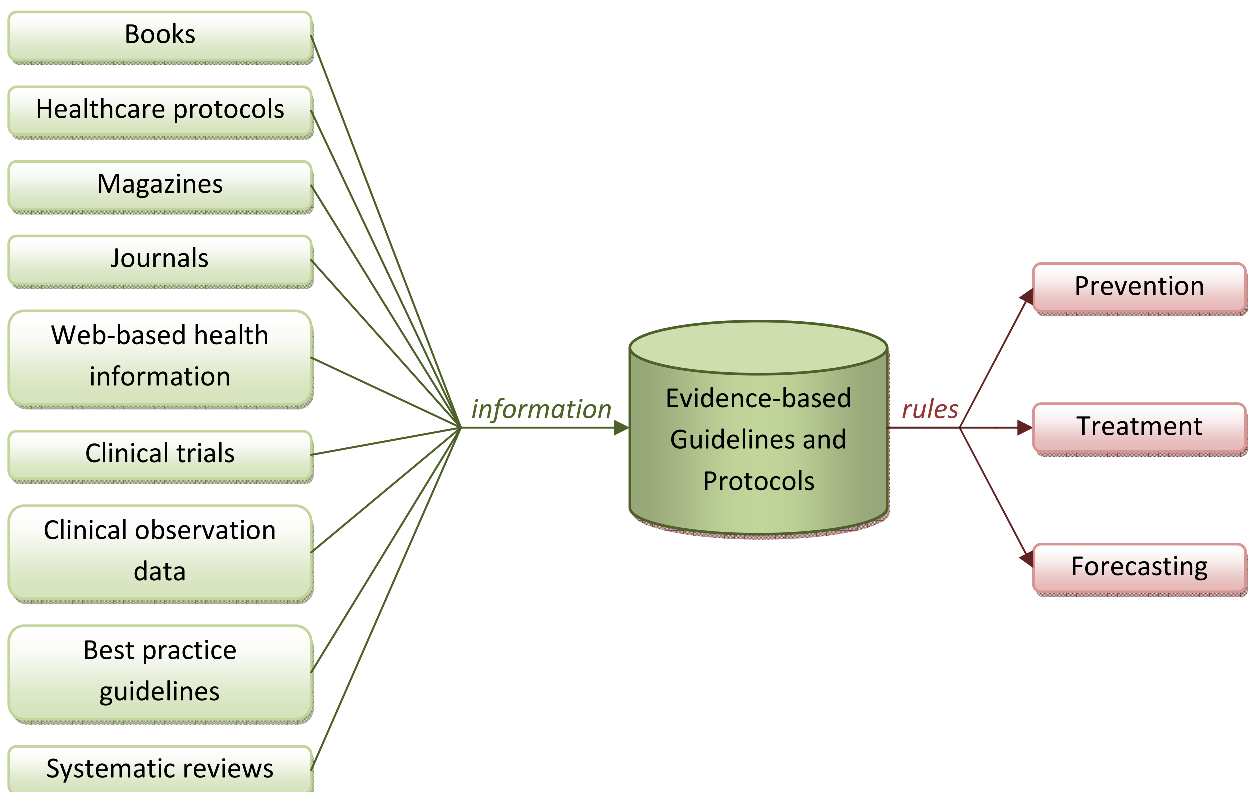

Figure 5: Data flow in Evidence Based Medicine.(adapted from (6))

EBM recognizes that many aspects of medical care depend on individual factors such as quality and value-of-life judgments, which are only partially subject to scientific methods. EBM, however, seeks to clarify those parts of medical practice that are in principle subject to scientific methods and to apply these methods to ensure the best prediction of outcomes in medical treatment, even as debate about which outcomes are desirable continues.

Evidence based medicine origins extend back to the 19th century, but remains a popular topic for clinicians, public health practitioners, purchasers, planners, and the general public. Opinions on the proliferation of EBM practice are divergent. Some critics see evidence based medicine as being old, others as a dangerous innovation. Its supporters look at it as a way to cut off clinical care costs and as an indispensible support when it comes to making clinical decisions.

However, both parts seem to agree on some aspects:

- Good doctors use both individual clinical expertise and the best available external evidence, and neither alone is enough. Without clinical expertise, practice risks becoming tyrannized by evidence, for even excellent external evidence may be inapplicable to or inappropriate for an individual patient. Without current best evidence, practice risks becoming rapidly out of date, to the detriment of patients.
- Evidence based medicine is not "cookbook" medicine. Because it requires a bottom up approach that integrates the best external evidence with individual clinical expertise and patients' choice, it cannot result in slavish, cookbook approaches to individual patient care. External clinical evidence can inform, but can never replace individual clinical expertise, and it is this expertise that decides whether the external evidence applies to the individual patient at all and, if so, how it should be integrated into a clinical decision.
- Hijacking of EBM by purchasers and managers to cut the costs of health care must be considered. This would not only be a misuse of evidence based medicine but suggests a fundamental misunderstanding of its financial consequences. Doctors practicing evidence based medicine will identify and apply the most effective interventions to maximize the quality and quantity of life for individual patients; this may raise rather than lower the cost of their care.
- Despite its ancient origins, evidence based medicine remains a relatively young discipline whose positive impacts are just beginning to be validated, and it will continue to evolve.

While it might seem out of the context of this project, the analysis of evidence based medicine concepts and of its acceptance among healthcare professionals was the basis for the definition of the protocols tool developed. Understanding the end-user needs, expectations and fears regarding evidence based decision support systems is essential to determine the set of features to be developed for the protocols solution.

2.2. Clinical protocols

A clinical protocol is a decision support system, specified based in research studies carried out in clinical trials that rely on EBM (Section 2.1). It can be generically defined as a formal design of an action plan explaining what will be done, when, how, and why. In a medical context, a protocol provides a precise and detailed plan for the study of a medical or biomedical problem and/or plans for a treatment regimen or therapy. In its essence, protocols resemble guidelines, with the advantage of incorporating decision trees, making them more universal and adaptable.

A protocol typically has an associated diagram defining diagnostic and treatment pathways, which should be followed according to each specific situation. Starting at the *root node* of the diagram, usually the title of the protocol, in each step (*node*) the professional is requested to perform some task or provide some kind of input. The following step is determined by comparing the condition(s) on each possible pathway with the condition of the patient in question and proceeding with the best match.

The process is repeated until a final stage is reached (i.e. one with no subsequent *nodes*), where the protocol is considered to have been applied to the patient. Figure 7 shows an example of a clinical protocol, where the rounded rectangle represents the *title node*, the rectangular boxes represent *task nodes* and the diamond-shaped boxes are *decision nodes.* Each decision node splits a pathway into two (or more) distinct paths.

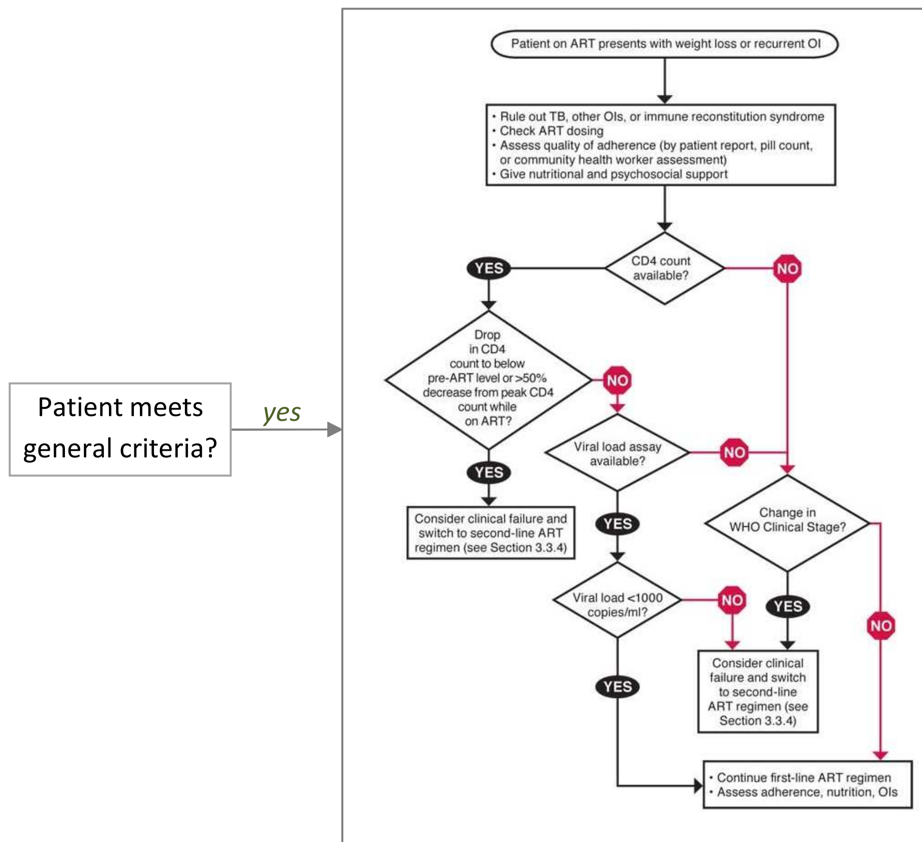

Figure 7: An example protocol.

Additionally, there can be general criteria that determine a group of patients (cohort) to which the protocol should be applied. This criteria is to be verified prior to applying the associated protocol to a patient.

In clinical context and medical literature, protocols are often referred to as *Guidelines*, *Protocols, Care plans, Plans* or less frequently as *Consultation Maps*, *Macros* or *Templates*. These multiplicity of terminologies can be misleading, thus it’s important to define what is meant by protocol through the course of this work:

> *A clinical protocol is a formal specification of a care plan’s possible pathways, with the intent of preventing, treating, diagnosing or managing a clinical condition associated with a patient.*

2.3. Objectives

The purpose of this project was to obtain a tool that would integrate an existing clinical software suite, providing support for the use of clinical protocols, namely an interface to create protocol flow diagrams, and the possibility of applying these protocols to patients and interactively following them in real-time. With this in mind the following general objectives have been defined:

1. **Requirements definition**
   - 1.1. Analysis of existing literature on evidence based medicine, decision support systems and more specifically on clinical protocols
   - 1.2. Determine expected use-cases for the protocols module
   - 1.3. Define functional and architectural requirements
   - 1.4. Prioritize functionalities to implement
2. **Component planning and development**
   - 2.1. Define necessary components
   - 2.2. Determine inter-component interaction methods
   - 2.3. Implement components according to requirements
   - 2.4. Perform unit tests and functional tests on developed components
3. **Protocols tool implementation**
   - 3.1. Implement a protocol creation tool integrated on the professionals’ BackOffice area in Alert®
   - 3.2. Implement a protocol execution tool integrated on the patients’ clinical record
   - 3.3. Perform unit tests and functional tests on developed protocol modules

2.4. Technological background

Although introducing new technologies in Alert® is a possibility (if they bring an extra value to the software), the priority for this project is to take advantage of the capabilities of technologies already adopted. Alert®'s architecture and the underlying technologies are therefore exposed on the following section. The tools used for development are also generally described, since their characteristics and capabilities are relevant to define implementation strategies for the proposed solution.

2.4.1. Alert® Architecture

Alert® is structured in two main components: the Data tier and the Presentation tier. These two layers interchange data through an intermediate JAVA tier, which makes up a 3-tiered architecture, as shown on Figure 8.

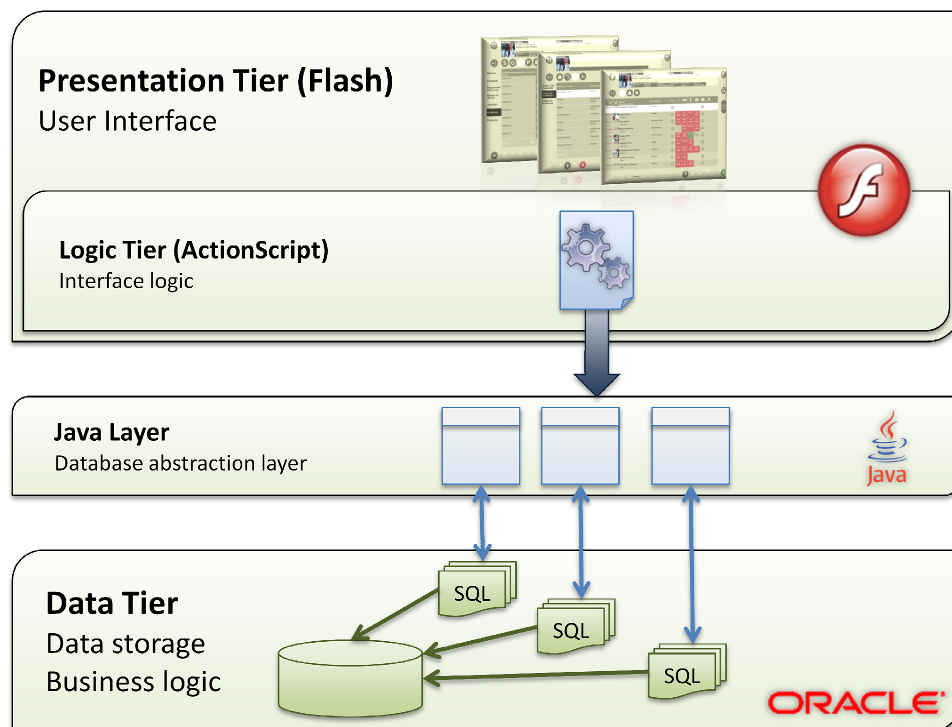

Figure 8: Alert® tiered structure.

Data Tier

The data layer uses Oracle (7) technology, specifically Oracle Database 10g.

The adopted data model supports:

- **Multiple institutions:** the same database stores configurations and data for different institutions and healthcare facilities.
- **Multiple applications:** the same database stores configurations and data related with different Alert® applications.
- **Multiple languages:** supports content translations to multiple languages.
- **Access control:** grants and permissions for each user are controlled from the database.

These characteristics make it possible to use the application in different environments, and the use of a centralized database shared by multiple institutions with little effort.

Unlike many 3-tiered products that keep the business logic on an intermediate logic layer, Alert®'s business logic resides on the database layer, which guarantees that any required changes have a reduced impact on the other layers. This option was made from the beginning to allow a faster development, since the company had already a strong background on database technologies. However, given the fast-growing complexity of the software, a migration of the business logic to the Java layer has been studied, to keep it better structured and database-independent.

The data tier exchanges data with Java through Java Data Base Connection (JDBC, an API for the Java programming language that defines how a client may access a database, provides methods for querying and updating data in a database).

Java Tier

Flash does not communicate directly with the database. Instead, there is an intermediate Java layer that manages the connection between interface and database. Although the Java component of the software is relatively small when compared to the other layers, and is essentially made of automatically generated classes, it represents a vital part of the system. It exposes the database functions, providing corresponding services, which can be accessed remotely. All information between the presentation and logic/data tiers is exchanged through these Java services, using Flash Remoting. This allows for interface development to be independent of database structure (and vice-versa), thus any structural changes in database are transparent to a Flash developer.

This layer is responsible for:

- **Database connection management:** Java manages every database connection and method invocation.
- **Session management:** logins, session timeouts and edit timeouts are managed within this layer.
- **Service logging and reports:** Java maintains log files to track service exceptions.

Presentation Tier

The top layer consists on the User Interface. It's developed with Flash technologies, using ActionScript programming language. Alert® screens are built with reusable and customizable components such as

DataGrids[2], *MultiChoices*[3], and *Keypads*[4]. The interface relies on ActionScript classes to implement UI logic, and takes advantage of inheritance capabilities to generalize concepts and features common to different screens.

For every screen within Alert® there is a corresponding Flash authoring file (**.FLA**). This file contains the basic screen structure and incorporates all the necessary pre-compiled components into the screen. It also defines build directories for the final ***.SWF**[5] files. Besides importing the necessary pre-compiled components, every **.FLA** file imports a corresponding ActionScript class, which is defined on a file with the same name and with the ***.AS** extension. These procedures are detailed on section 4.5.1 - The Screen Creation Process.

Data exchange with JAVA Classes is accomplished with Flash Remoting components and JAVA web services.

2.4.2. Alert® Technologies

As mentioned on section 2.4.1, Flash, Java and Oracle are the foundations of Alert® products at different levels. These technologies and a summary of their features and advantages are presented on this section.

Adobe Flash

Flash is an authoring tool widely used to create a diversity of interactive content and applications. Utilities created with Flash can be as simplistic as a promotional banner, or as complex as 3D games, charting tools and complete websites. Although Flash is mostly used for web content (in great part due to its great flexibility, reduced size of the files it produces, and because it's a cross-browser tool), it's also frequently adopted by developers to build their software's User Interface.

There are several reasons why Flash was chosen as Alert®'s interface authoring tool:

- Flash is **extremely flexible**. Among other things, it works with vectorized components and does not rely on strict component layouts. This makes it possible to run a Flash application on displays with different sizes and resolutions, without compromising its look and feel.
- Making Alert® software **remotely accessible** to healthcare professionals has been on the company's plans since its foundation. This migration is already taking place, and having a Flash-based interface has greatly simplified this process, since it was designed as a technology for the web. Apart from programmers, end user requirements to access Alert® remotely are virtually inexistent: a browser with Flash player plugin is the only software required, and both are found on most computers or can be easily downloaded.
- In addition to being a powerful design tool, Flash is bundled with its own object-oriented scripting language, ActionScript, enabling the creation of complex applications behind an appealing and flexible user interface.

[2] a Datagrid is a component used to display information structured in a grid
[3] A *combo box* selection component, used in Alert to make selections from a pre-built set of options
[4] A component used in Alert to input numbers and dates
[5] *"Small Web Format"*, also known as *Swiff*, the name for the compressed and uneditable files produced by Flash

Other technologies such as Oracle Forms (8), Windows forms (9), Java Applets (10) or Ajax (HTML and Javascript) (11) could be eventual alternatives for Flash as Alert®'s interface. In spite of having their own advantages, whose discussion is out of the scope of this report, Adobe Flash was considered a more adequate solution for the reasons explained.

ActionScript

The first version of ActionScript was released with Flash 2, in 1997. Since then, it has suffered profound improvements, and has become a robust scripting language.

ActionScript 2.0 was made available with Flash MX 2004, and it radically improved object-oriented development in Flash by formalizing objected-oriented programming (OOP) syntax and methodology. Most of the OOP syntax in ActionScript 2.0 is based on the proposed ECMAScript 4 standard at the time (12).

ActionScript 2.0 most noticeable features include:

- Class and Class Inheritance support;
- Interface support, used to create abstract data types;
- Formal method-definition syntax, used to create instance methods and class methods in a class body;
- Formal getter and setter method syntax;
- Formal property-definition syntax, used to create instance properties and class properties in a class body;
- Controlled access to methods and properties, with private/public declarations;
- Static typing for variables, properties, parameters, and return values, used to declare the data type for each item. This eliminates careless errors caused by using the wrong kind of data in the wrong situation;
- Type casting, used to tell the compiler to treat an object as though it were an instance of another data type, as is sometimes required when using static typing;
- Exception handling, used to generate and respond to program errors;
- Easy linking between movie clip symbols and ActionScript 2.0 classes via the symbol Linkage properties, allowing easier implementation of MovieClip inheritance.

ActionScript 3.0 has been recently released. This new version brings significant improvements in terms of performance, and surely all of Alert® will be adapted to use ActionScript 3.0. For the time being however, and because this transition will require a significant effort and time, version 2.0 is used.

Flash Remoting

Macromedia Flash Remoting MX provides the connection between Flash and web application servers, simplifying the creation of Rich Internet Applications. All the communication with the server is accomplished through Flash Remoting components, which implements the AMF (Action Message Format) protocol. AMF is a binary format used primarily to exchange data between a flash application and a database, using Remote Procedure Calls.

AMF uses Web Services and delivers high compression rates and, consequently, faster data exchange.

Java

Java is an object-oriented programming language, which aims to (13):

1. Allow the same program to be executed on multiple operating systems.
2. Contain built-in support for using computer networks.
3. Execute code from remote sources securely.
4. Include the good aspects of other object-oriented languages, making it easy to use.

Perhaps the built-in support for computer networks and the ability to execute remote code securely were the characteristics that most contributed to choosing Java. Alert® needed a fast, robust and secure way of interchanging data between multiple interfaces on the users' equipments and a central database in a server. Java makes this link possible through the use of web services[6].

Oracle Database

Introduced in the late 1970s, Oracle databases consist of a collection of data managed by an Oracle database management system. Oracle's relational database was the world's first database product to support the Structured Query Language (SQL, now an industry standard), and to run on a variety of platforms. Oracle stands out as one of the most popular databases of the world.

Alert® uses an Oracle database due to its flexibility and efficiency and because of its small management costs. The version currently used in Alert® is Oracle10g, which was introduced in the market in 2005.

2.4.3. Development Tools

The tools described next are typically used for software development in Alert®, through the various development stages.

SVN (Subversion)

SVN (also known as Subversion) is a version control system. It's targeted at large projects, and especially useful when concurrent editing of files is needed. It uses a client-server architecture: a server stores the most current version of the project and its history, and clients connect to the server in order to check out a complete copy of the project. Clients then work on their local copy and later *check in* their changes to the server (repository).

SVN features include:

- Support for concurrent work on the same project, by several developers;
- Automatic merge of changes, if no conflicting changes are made;
- Versioning of files, directories, renames, and file meta-data;
- Branching and tagging (e.g. create a branch for debugging while keeping the main branch for new features development);

[6] Web services can be considered *Web APIs* (application programming interface), are functions that can be accessed over a network, such as the Internet, and executed on a remote system that hosts the requested services.

- Parseable and human-readable output;
- Efficient *delta* compression (when a file is committed to the server, only the lines that have suffered changes from the previous version are transmitted and stored, saving on resources).

All the database scripts and interface source code for Alert® are kept on a Subversion repository and managed through SVN versioning system. For interface development, access to the repository is commonly accessed through Eclipse's Subclipse plugin.

Eclipse

Eclipse is best known for its Integrated Development Environment (IDE), but it's a multi-purpose open source platform, which can be used for a variety of software development related tasks. It stands as a powerful user-friendly platform that increases productivity and efficiency noticeably. Eclipse relies on *plugins* to provide integrated utilities for most popular programming languages.

The following *plugins* have been frequently used during development:

FDT (Flash Development Tool)

FDT is a *plugin* for Eclipse that adds Flash and ActionScript utilities such as advanced code completion, live error highlighting, *quickfixes*, quick source navigation and Javadoc style documentation[7], among others. This tool makes Eclipse a better development environment for Actionscript than the Flash authoring tool itself. This is why all code creation is made within Eclipse, and Flash is only used for compiling.

SQL Explorer

SQL Explorer adds an SQL Client to Eclipse, making it possible to query and browse any JDBC compliant database within the development environment. It uses Java drivers to access Alert®'s Oracle database, and is fully integrated into Eclipse interface. This is extremely useful for developers to switch between Flash perspective and SQL Explorer perspective to have an understanding of the database structure and contents.

Subclipse

Subclipse is an SVN client for Eclipse, that adds Subversion integration to the Eclipse IDE.

XRay

Xray (The AdminTool) is a "snapshot viewer" of the state of Flash applications that has no impact on their performance or file size. Xray can show a hierarchical view of a running application's components and properties, and allows real-time manipulation of these properties (Figure 9). Xray also displays application outputs, which can be used to trace the application flow and for debugging.

[7] Javadoc is a tool for generating HTML documentation from Java source code.

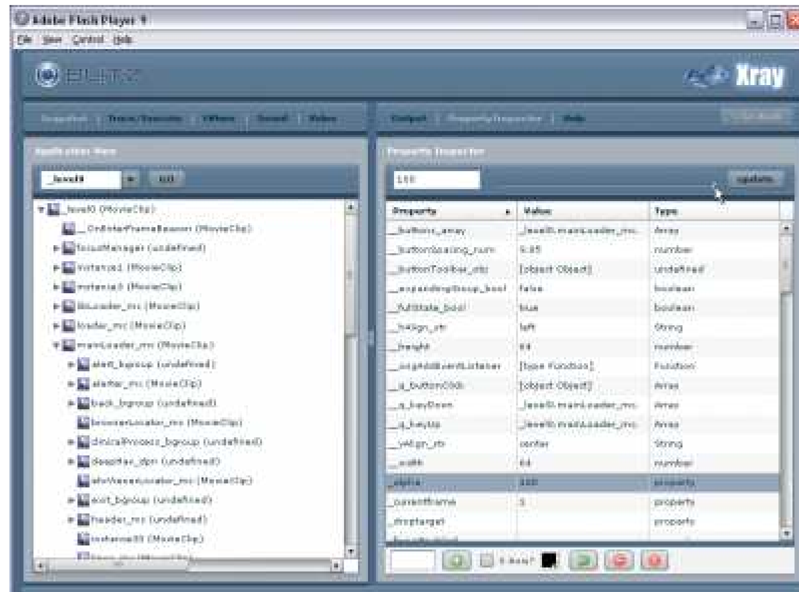

Figure 9: XRay interface.

Xray has proven to be a unique tool when it comes to Flash runtime debugging. It's used very frequently, especially on testing and debugging stages.

Service Capture

ServiceCapture (Figure 10) is an application that captures all HTTP traffic sent and received by the computer where it's being executed, and is designed to help Rich Internet Application (RIA) developers in the debugging, analysis, and testing of their applications.

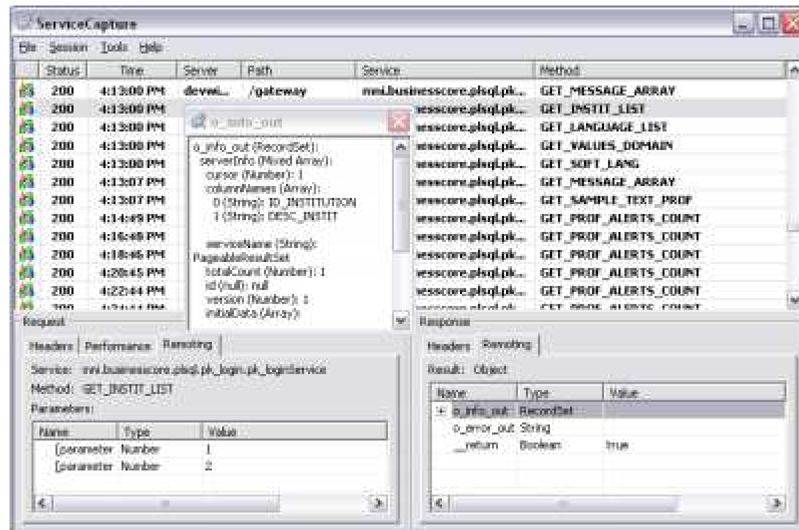

Figure 10: Service capture interface – viewing input/output variables.

Flash components communicate with Alert® server through HTTP, and this tool comes very handy for tracing problems, by looking at what information is being sent and received.

3. State Of The Art

Conventional protocols present population-based recommendations, and the information contained within such protocols may be difficult to access and apply to a specific patient during a medical consultation. To be able to create systems capable of interactively determining which protocols are applicable, and address those protocols independently for individual patients requires prior definition of computer interpretable representations of the clinical knowledge contained in protocols.

A number of groups are actively developing computer interpretable guideline[8] (CIG) representation languages for this purpose. Groups have adopted different approaches reflecting their interests and expertise. Nevertheless, many of the approaches have in common a hierarchical decomposition of guidelines into networks of component tasks that unfold over time. This approach has been described as the *task-based paradigms*, and modeling formats based on this approach are referred to as *Task-Network Models* (TNMs). (14)

This chapter provides an overview of existing CIG representations, as well as an analysis on associated graphical visualization tools.

3.1. Computer interpretable protocol representations

PROForma and *Asbru* are the two most commonly used languages to represent protocols in a way that is interpretable by computers. The following sections expose these languages in more detail, and the most relevant visualization tools associated with each of them (Figure 11): *Arezzo* and *Tallis* (based in *PROforma*), *AsbruView* and *CareVis* (based in *Asbru*).

Figure 11: Protocol representation languages and respective visualization tools.

3.1.1. PROforma

PROforma is a formal knowledge representation language capable of capturing the structure and content of a clinical guideline in a form that can be interpreted by a computer. The language forms the basis of a

[8] In Alert®, Guidelines and Protocols are distinct concepts. In short, a protocol in Alert® has its tasks organized in decision trees (or flow charts), while a Guideline's tasks are an unordered set. However, on most clinical environments and literature, the terms *Guideline* and *Protocol* are used interchangeably when referring to decision support systems that rely on a series of steps that need to be performed. Hence some references to Guidelines along this chapter. The term *plan* is also frequently used to refer to Protocols/ Guidelines

method and a technology for developing and publishing executable clinical guidelines. Applications built using PRO*forma* software are designed to support the management of medical procedures and clinical decision making at the point of care.

In *PROforma*, a guideline application is modeled as a set of tasks and data items. The notion of a task is central - the *PROforma* task model (Figure 12) divides from the keystone (generic task) into four types: plans, decisions, actions and enquiries.

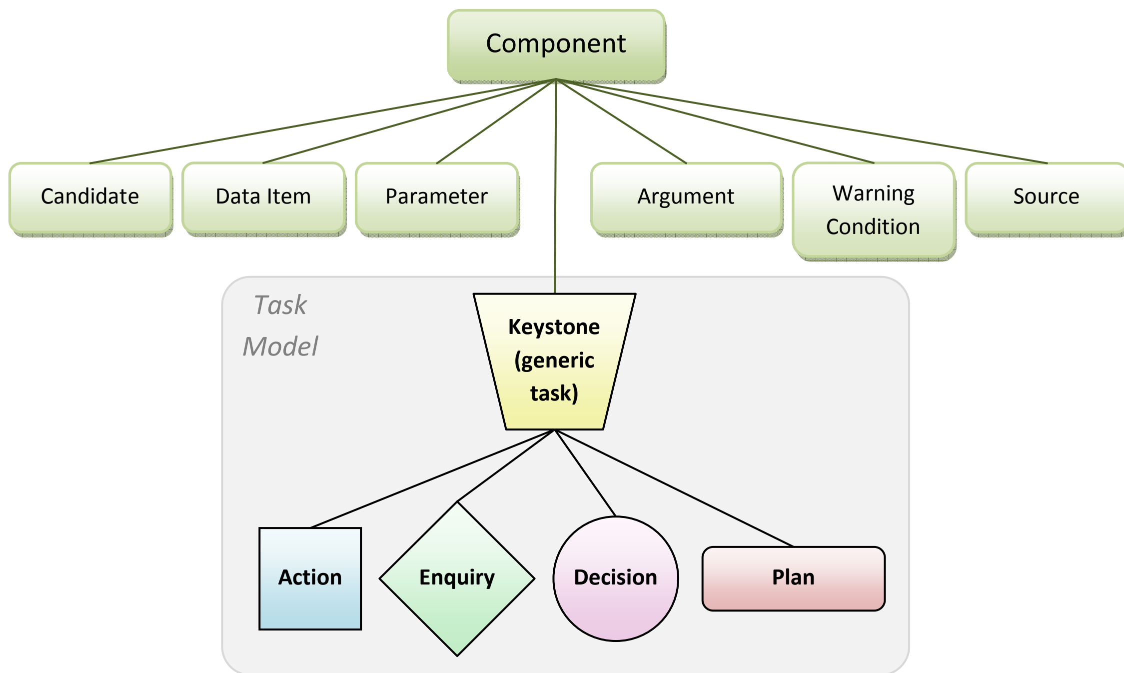

Figure 12: The PROforma class model.

Plan

A plan is a set of tasks to be carried out in order to achieve a clinical goal. Plans are the basic building blocks of a guideline, and may contain any number of tasks of any type, including other plans, usually with an imposed ordering.

Decision

A decision is a task involving choices of some kind, such as a choice of investigation, diagnosis or treatment. The PRO*forma* specification of a decision task defines the decision options, relevant information and a set of argument rules which determine which of the options should be chosen according to current data values.

Action

An action is typically a clinical procedure (e.g. administering an injection) which needs to be carried out.

Enquiry

An enquiry is an action returning required information, typically a request for information or data from the user. To specify an enquiry, a description of the information required and a method for getting it are required.

Any specific clinical task is seen as an instance of some more general class of tasks. Each class is eventually a specialization of a root task. Any class can be further specialized into particular sub-types. For example, plans may be specialized into research protocols and routine guidelines and decisions can be specialized into diagnosis and treatment decisions, or others as shown on Figure 13:

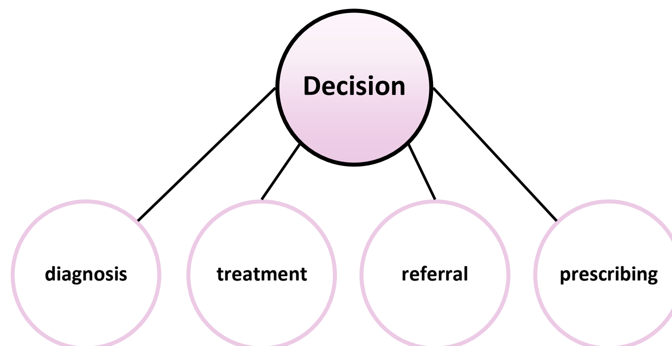

Figure 13: Task sub-types in PROforma.

When tasks are enacted, the communication of information between tasks is achieved by passing messages between encapsulated task objects, rather than by explicit procedure calls or other mechanisms that require access to the internal definition of a task. Task states and possible transitions between states are displayed on Figure 14

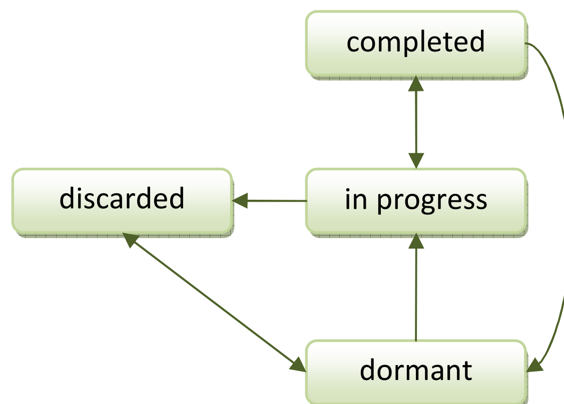

Figure 14: Task state transitions.

Every task inherits part of its specification from the classes above it in the hierarchy, expressed as a set of attributes. A task is distinguished from its parents by the possession of additional attributes, and distinguished from its siblings by different values for their common attributes.

PROforma supports the definition of clinical guidelines and protocols in terms of:

- A well-defined set of tasks that can be composed into networks representing plans or procedures carried out over time. These enable the high level structure of a guideline to be represented.
- Logical constructs (such as situations, events, constraints, pre and post-conditions, and inference rules) which allow the details of each task and inter-relationships between tasks to be defined using templates.

The properties and behavior of each task type are determined by its attributes and the values attached to them. These determine how and when a task is processed when the electronic guideline of which they form a part, is enacted.

Attributes of the (top level) root task are generic and are inherited by each task class and sub-class; attributes of task classes are specific to that class and only inherited by sub-classes of that type.

Attributes and their allowable values are specified in templates. These show us, for example, that every task has a description, a title and preconditions, but only tasks of type action have a procedure, and only tasks of type decision have candidates. The attributes of the root (generic) task are shown in Table 2.

Table 2: Attributes of the generic task.

Attribute	Description
Name	Unique identifier of task
Caption	Descriptive title of task
Description	Textual description of task
Goal	Purpose of task
Pre-conditions	Conditions necessary before a task may be started
Trigger conditions	Conditions which will initiate a task
Post-conditions	Conditions true on task completion

Development of a PROforma application is a two-step process:

- Developing a high level diagram which describes the outline of the protocol in terms of the PROforma set of tasks (logical and temporal relationships between tasks are captured naturally by linking them as required with arrows);
- Converting and storing this graphical structure in a database, using software implementations of the task templates, with the detailed procedural and medical knowledge required to execute the guideline.

Both these steps are carried out using a graphical editor (or knowledge authoring tool) such as the *Arezzo Composer* or *Tallis Composer*. The resulting computerized clinical guideline is tested and executed using a *PROforma*-compatible engine, such as *Arezzo Performer* or *Tallis Engine*.

3.1.2. Asbru

Asbru is a time-oriented, intention-based, skeletal plan-specification representation language that is used in the **Asgaard Project** to represent clinical guidelines and protocols in XML.(15)

Asbru is part of an effort to create information technologies capable of supporting protocol based care. It was developed with the purpose of capturing all aspects of a medical treatment plan into a guideline representation language. This language tries to deal with real world domain complex aspects like time constraints, temporal uncertainties, intentions, plan conditions…

Asbru can be used to express clinical protocols as skeletal plans that can be instantiated for every patient. It was designed specifically to the set of plan management tasks. *Asbru* enables the designer to represent both the prescribed actions of a skeletal plan and the knowledge roles required by the various problem-solving methods performing the intertwined supporting subtasks.

Language features include:

- continuity of prescribed actions and states;
- intentions, conditions, and world states as temporal patterns;
- flexible representation of uncertainty in temporal scopes and parameters by bounding intervals;
- sequential, parallel, periodical, custom-ordered or unordered plan execution;
- plan execution monitoring, by defining particular conditions;
- Statement of explicit intentions and preferences separately for each plan.

The following example shows parts of an *Asbru* plan for artificial ventilation of newborn infants. The plan is represented in XML and contains domain definitions and a set of sub-plans. The ventilation plan consists of conditions and the plan body including a sequential execution of the initial plan and controlled ventilation plan.

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <!DOCTYPE plan-library SYSTEM "asbru_7_3.dtd">
3:  <plan-library>
4:     <domain-defs>
5:         <domain name="controlled_ventilation_domain">
6:             ...
7:         </domain>
8:     </domain-defs>
9:     <plans>
10:        <plan-group>
11:            <plan name="ventilation_plan">
12:                <intentions> ... </intentions>
13:                <conditions>
14:                    <complete-condition>
15:                        <constraint-combination type="and">
16:                            <parameter-proposition parameter-name="FiO2">
17:                                <value-description type="less-or-equal">
18:                                    <numerical-constant value="40"/>
19:                                </value-description>
20:                            ...
21:                        </constraint-combination>
22:                    </complete-condition>
23:                    <abort-condition>
24:                        <constraint-combination type="or">
```

```
25:                          <parameter-proposition parameter-name="FiO2">
26:                              <value-description type="greater-than">
27:                                  <numerical-constant value="90"/>
28:                              </value-description>
29:                          ...
30:                      </constraint-combination>
31:                  </abort-condition>
32:              </conditions>
33:              <plan-body>
34:                  <subplans type="sequentially">
35:                      ...
36:                      <plan-activation>
37:                          <plan-schema name="initial_plan"/>
38:                      </plan-activation>
39:                      <plan-activation>
40:                          <plan-schema name="controlled_ventilation_plan"/>
41:                      </plan-activation>
42:                  </subplans>
43:              </plan-body>
44:          </plan>
45:          ...
46:          <plan name="controlled_ventilation_plan">
47:              <plan-body>
48:                  <subplans type="parallel">
49:                      ...
50:                      <plan-activation>
51:                          <plan-schema name="handle_PCO2_plan"/>
52:                      </plan-activation>
53:                      <plan-activation>
54:                          <plan-schema name="handle_tcSaO2_low_plan"/>
55:                      </plan-activation>
56:                      <plan-activation>
57:                          <plan-schema name="handle_tcSaO2_high_plan"/>
58:                      </plan-activation>
59:                  </subplans>
60:              </plan-body>
61:          </plan>
62:          ...
63:      </plan-group>
64:  </plans>
65:</plan-library>
```

This format provides not only machine-readable plans, but also human-readable content (to some extent). However, it's clear that for slightly more complex situations, the length and density of the code would make it impossible to be interpreted for any average person.

This language would clearly benefit with a visualization tool, and projects such as *AsbruView*, or *CareVis* were created for that purpose.

3.1.3. Other languages

EON

Developed at Stanford University, *EON*(16) is intended to provide a suite of models and software components for creating guideline-based applications. It views the guideline model as the core of an extensible set of models, such as a model for performing temporal abstractions. *EON* uses a task-based approach to define decision-support services that can be implemented using alternative techniques. *EON*'s guideline execution server uses formalized clinical guidelines and patient data to generate

situation specific recommendations. A temporal data mediator supports queries involving temporal abstractions and temporal relationships. A third component provides explanation services for other components.

GLIF

The Guideline Interchange Format (*GLIF*)(17) has been collaboratively developed by groups at Columbia, Stanford and Harvard universities (working together as the InterMed Collaboratory). *GLIF* stresses the importance of sharing guidelines among different institutions and software systems. *GLIF* tries to build on the most useful features of other guideline models, and to incorporate standards that are used in health care. Its expression language was originally based on the Arden Syntax (a subsequent object-oriented language, *GELLO*, is now being refined for consideration as an HL7[9] standard), and its default medical data model is based on the HL7 Reference Information Model (*RIM*).

GUIDE

GUIDE (18)is part of a guideline modeling and execution framework being developed at the University of Pavia and supports:

1. integrating modeled guidelines into organizational workflows,
2. using decision analytical models such as decision trees and influence diagrams, and
3. simulating guideline implementation in terms of Petri nets.

GUIDE considers issues such as patient data, the implementing facility's organizational structure, and resource allocation. This paper considers the guideline model as presented in the *GUIDE* tool, which is a graphical authoring tool that a modeler uses to create a guideline flowchart.

PRODIGY

Developed at the University of Newcastle upon Tyne, the purpose of *PRODIGY*(19) is to provide support for chronic disease management in primary care. The *PRODIGY* project has aimed to producing the simplest, most readily comprehensible model necessary to represent this class of guidelines. The language has been used to encode three complex chronic disease management guidelines, and at least two vendors have integrated *PRODIGY* components into their clinical information systems for general practitioners.

3.2. Visualization tools

The first implementation of software to create, visualize and enact *PROforma* guidelines was *Arezzo,* now a commercial product available from InferMed Ltd.. *Arezzo*® is designed for Microsoft Windows platforms, and comprises *Composer*, a graphical knowledge authoring tool, and *Performer*, an application tester and execution engine.

Tallis is a Java implementation of *PROforma*-based authoring and execution tools developed by Cancer Research UK. *Tallis* is based on a later version of the *PROforma* language model. It comprises *Composer* (to support creation, editing, and graphical visualization of guidelines), *Tester* and *Engine* (to enact

[9] Health Level Seven (HL7), is an organization involved in development of international healthcare standards. "HL7" is also used to refer to some of the standards created by the organization (i.e. HL7 RIM etc.).

guidelines and allow them to be manipulated by other applications). *Tallis* is also designed for delivering web-based services; applications will run on any platform and integrate with other components, including 3rd party applications.

The *Tallis Publisher* (based on Java Servlets) forms part of the *Tallis* software suite, and has also been built to allow guidelines to be published and enacted over the internet.

The Institute of Software Technology and Interactive Systems at Vienna University of Technology is developing *AsbruView* and *CareVis*, two graphical user interfaces to support visualization and understanding of Asbru guidelines.

3.2.1. Arezzo

Arezzo technology consists of three main modules: the Composer, a knowledge authoring tool, the Tester, and the Performer, an application enactment engine. Figure 15 illustrates the different purposes of these modules.

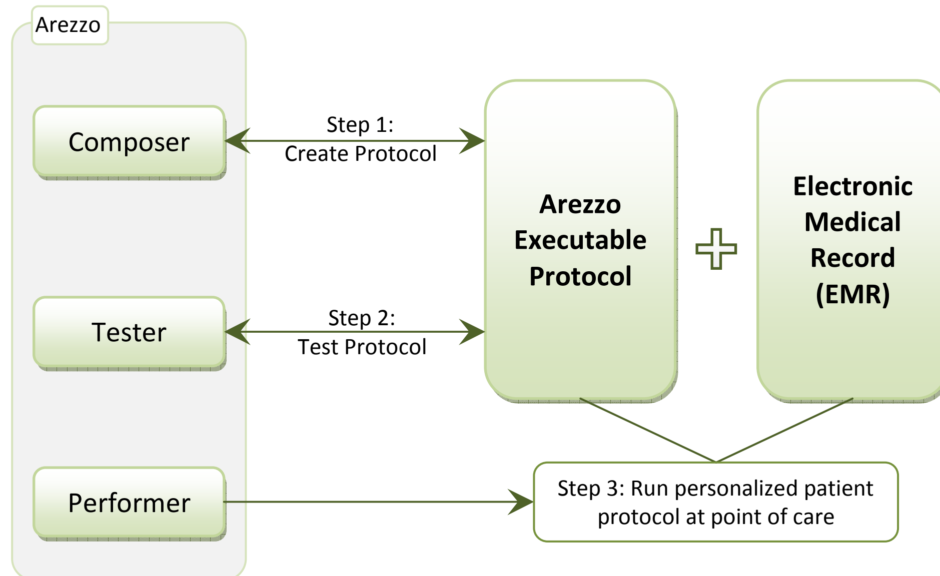

Figure 15: Arezzo modules.

Composer / Tester

The *Composer* is a graphical editor or knowledge authoring tool which uses *PROforma* notation to capture the structure of a guideline (laid out by an author) and generate an executable specification. The *Arezzo Composer* tool is the developer's GUI. The building blocks used to construct a guideline of any level of complexity are the four *PROforma* task types, each type being represented by its own icon. Data items and their properties to be collected during protocol enactment are also defined using the *Composer*. The *Tester* is used to test the guideline logic before deployment. Guideline applications can be run, debugged and validated within this module.

Performer

The Arezzo enactment engine (the *Performer*) tests and executes guidelines defined in the *PROforma* language. The *Performer* interprets the guideline specification and during guideline enactment, prompts the user to perform actions, collects data, carries out procedures and makes decisions as required (Figure 16). During enactment, the *Performer* also maintains a local database of patient data, which is added to by the user and queried by the Engine to evaluate specified conditions.

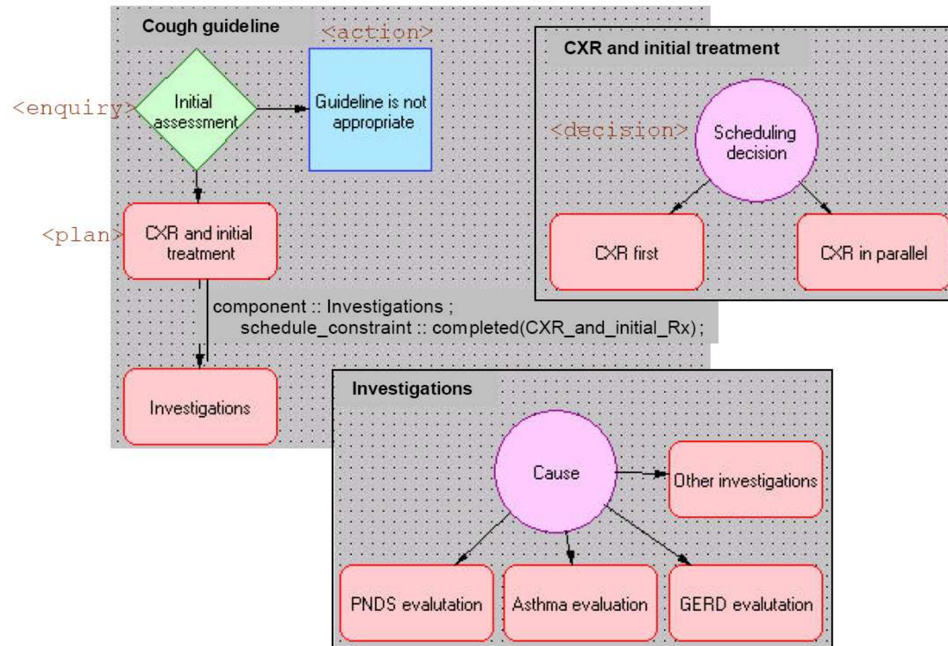

Figure 16: Arezzo Performer – Arezzo's graphical view of the cough guideline encoding in PROforma.

3.2.2. Tallis

The *Tallis* toolset relies on the *PROforma* language to model clinical processes (20). Developed at the Advanced Computation Laboratory (ACL) of Cancer Research UK it's a suite of software tools to support authoring, publishing and enacting of clinical knowledge applications over the web – applications designed to support the management of medical procedures and clinical decision making at the point of care. The *Tallis Composer* is a graphical editor that supports the authoring process. The resulting clinical knowledge application can then be tested in the *Tallis Tester*. Finally, the *Tallis Engine* enables the execution of the clinical application over the web.

Composer

Tallis Composer is a tool for creating and editing *PROforma* applications, or “process-descriptions”: descriptions in the *PROforma* language of the tasks that are to be carried out by interacting agents, whether human beings or software components, in order to accomplish some objective.

Using the set of *PROforma* task types (plans, actions, enquiries and decisions), the high level structure of the process is laid out and assembled as a network.

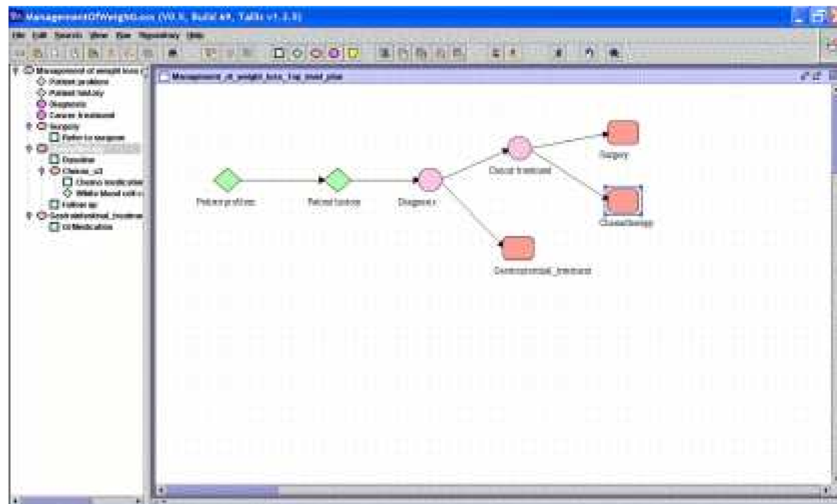

Figure 17: Tallis Composer – building a guideline flow diagram.

Figure 17 shows a network of tasks making up part of a guideline for cancer diagnosis and treatment. The work flows in the direction of the arrows, from left to right. It begins with an enquiry about the patient's problem (green diamond, ◇), which in this example is unexplained weight loss. The enquiry is followed by a diagnosis decision (pink circle, ○). In this case the diagnosis decision is a choice among a number of diseases that might be the cause of the weight loss. If cancer is diagnosed the workflow continues with a second decision: whether to treat the cancer with chemotherapy or surgery. Depending on the decision taken, the guideline is completed by enacting the appropriate treatment plan (red rounded rectangle, ▭).

Process-descriptions are displayed in *Tallis* both in a network view and in a tree view:

- The tree view displays the process-description’s hierarchical structure
- The network view displays task ordering according to scheduling constraints

The hierarchy of a process-description is based on plans: each plan defines a new level in the hierarchy.

- You can view all the plans and their contents at a glance in the tree view.
- The network is more suited for viewing the contents of one plan at a time.

The detailed knowledge that is required to enact each component task is entered as task attributes.

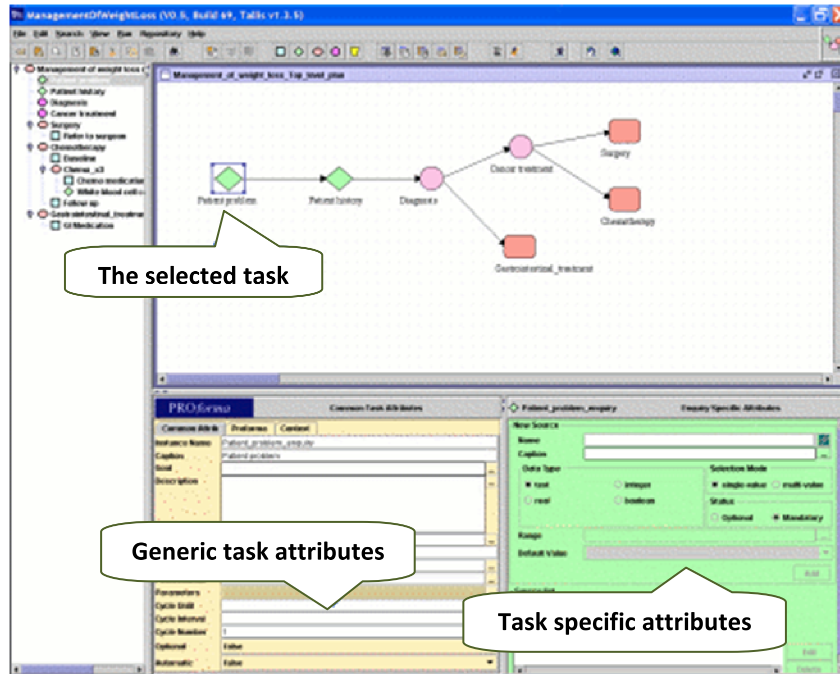

Figure 18: Tallis Composer – defining task attributes.

Each task type is defined by sets of generic and specific attributes. On the left side of Figure 18 there is a set of fields where the author can enter values for the generic attributes of the selected task (here, an enquiry). On the right side is a set of fields that represent the distinguishing attributes of the task type.

Once the *PROforma* application is developed, it can be tested in the *Tallis Tester*.

Tester

The *Tallis Tester* is a tool for testing and debugging the logic of a developed *PROforma* application, or "process-description".

The *Tester* keeps track of which tasks need to be performed to advance the process, and provides information regarding the current state of the process. It can also receive messages indicating that certain tasks have been completed, or that data relevant to the running of the process has been provided.

The *Tallis Tester* is used in the authoring process, and is not intended for clinical use. It differs from the web enactment interface in some respects:

User interface modifications such as customized web pages do not appear in the *Tallis Tester*.

As the Web Enactment is based around a sequential set of web pages (rather than a set of interactive panels as in the Tester) there are certain other minor differences in the runtime behavior (most notably concerning the relationship between sources and enquiries).

Tallis Tester is a stand-alone application, which can be launched from *Tallis Composer*. When a process-description is submitted to the *Tallis Tester*, it is first checked for syntactic and basic semantic errors. If no errors are found, the *Tallis Tester* is launched, and the process-description is enacted.

The process-description is displayed in a tree view (Figure 19), similar to the one in *Tallis Composer*. The colors of the tasks change during the enactment, as they represent task states.

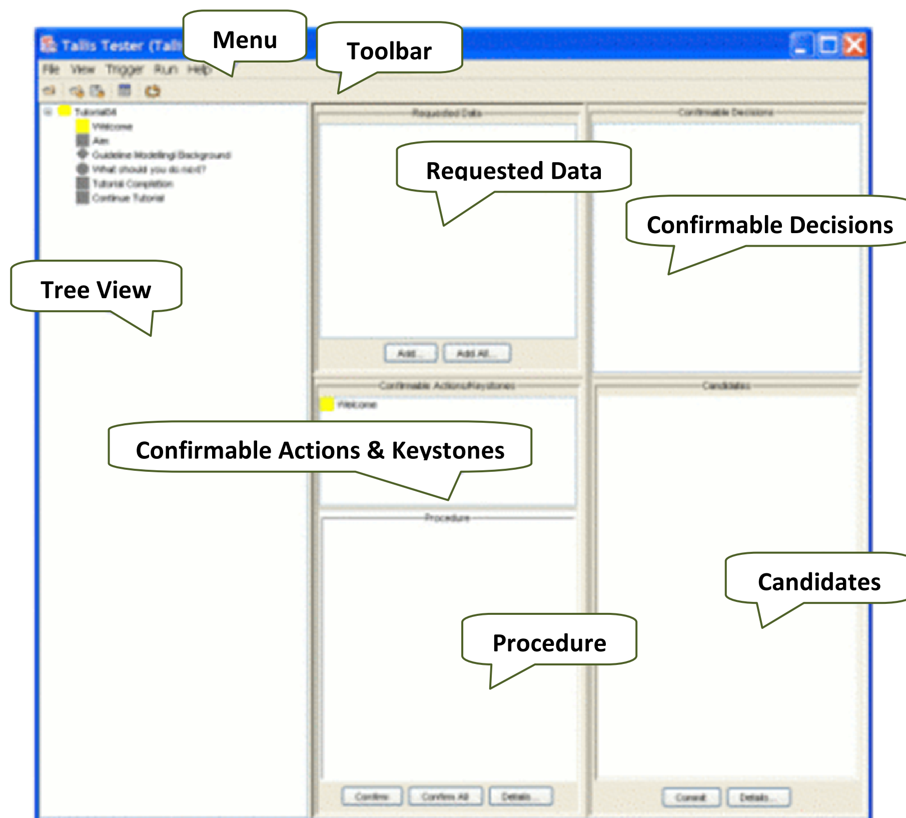

Figure 19: Tallis Tester – Interface components.

Once the *PROforma* application is tested and debugged, it can be enacted over the web using the *Tallis Engine*.

Engine

PROforma applications, or "process-descriptions", can be enacted by the *Tallis Engine*. The engine keeps track of which tasks need to be performed to advance the process, and provides information to external agents regarding the current state of the process. The engine can also receive messages from agents indicating that they have completed certain tasks or provided data relevant to the running of the process.

The *Tallis Engine* can enact *PROforma* applications either locally or over the web. The enactment interface consists of dynamically generated web pages that take the end-user through the application's component tasks. The standard web pages can be replaced, if required, by web pages customized for an individual application (Figure 20).

Figure 20: Tallis Engine – customizable front-end.

3.2.3. AsbruView

An *Asbru* plan offers some readability to humans, since it's represented in XML. However, understanding a plan in such a representation requires training as well as semantic and syntactic knowledge about the language. This kind of knowledge is not frequently found among physicians – supposedly the end users of the care plan. The formal representation needs to be translated into a visually richer format, one that is familiar to domain experts.

AsbruView (21), as the name itself indicates, is visualization and user interface to deal with treatment plans expressed in *Asbru*. It uses graphical metaphors to make the underlying concepts easier to grasp, employs glyphs to communicate complex temporal information and colors to make it possible to understand the connection between views.

This tool is essentially made up of two distinct views, *Topological View* and *Temporal View*. The topological view stands out as it's actually a set of metaphors used to illustrate concepts.

Topological view

Each plan is displayed as a running track (Figure 21). The time dimension (from left to right) is symbolic, as it does not actually reflect the plan's duration. The traffic signs symbolize the preconditions: the "no entrance with exceptions" stands for the filter precondition (which must be true for the plan to be applicable at all); the barrier symbolizes the setup precondition (which must also be true - but which can be achieved by other plans if it is not - for the plan to be applicable).

The colors of the lights of the traffic lights stand for one further condition:

- **Red:** the abort condition (which specifies when the plan has to be stopped and regarded as failed);
- **Yellow:** the suspend condition (which defines when a plan has to be interrupted to treat an emergency, for example);
- **Green:** the reactivate condition (which specifies when a suspended plan can be continued).

The finishing flag, finally, symbolizes the complete condition, which specifies when the plan has reached its goal and can be considered successful.

Plans can have sub-plans, which are then stacked on top of the containing plan.

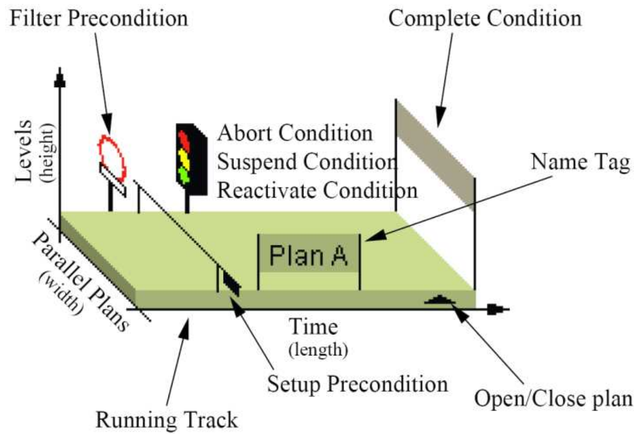

Figure 21: AsbruView - Topological View.

Timeline view

The timeline or temporal view is based on timelines (Figure 22). The focus of this view is not so much the topology of plans (which is also visible), but the exact temporal dimensions of plans and conditions (unlike the *Topological view*).

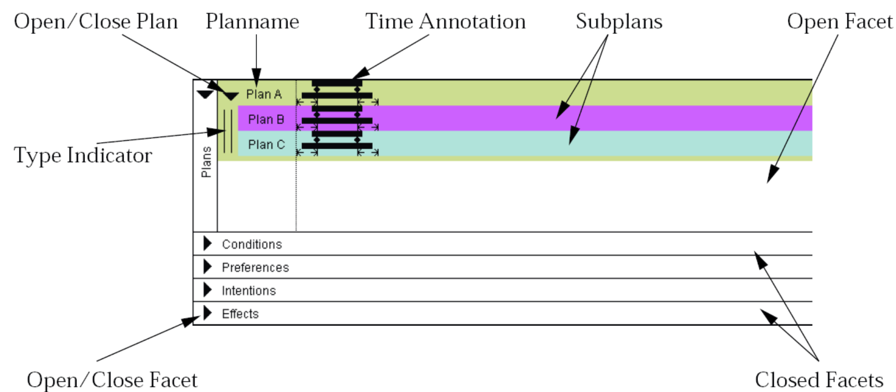

Figure 22: AsbruView – Timeline View.

The combination of these views provides a much more intuitive solution to display *Asbru* based protocols, than plain XML (Figure 23).

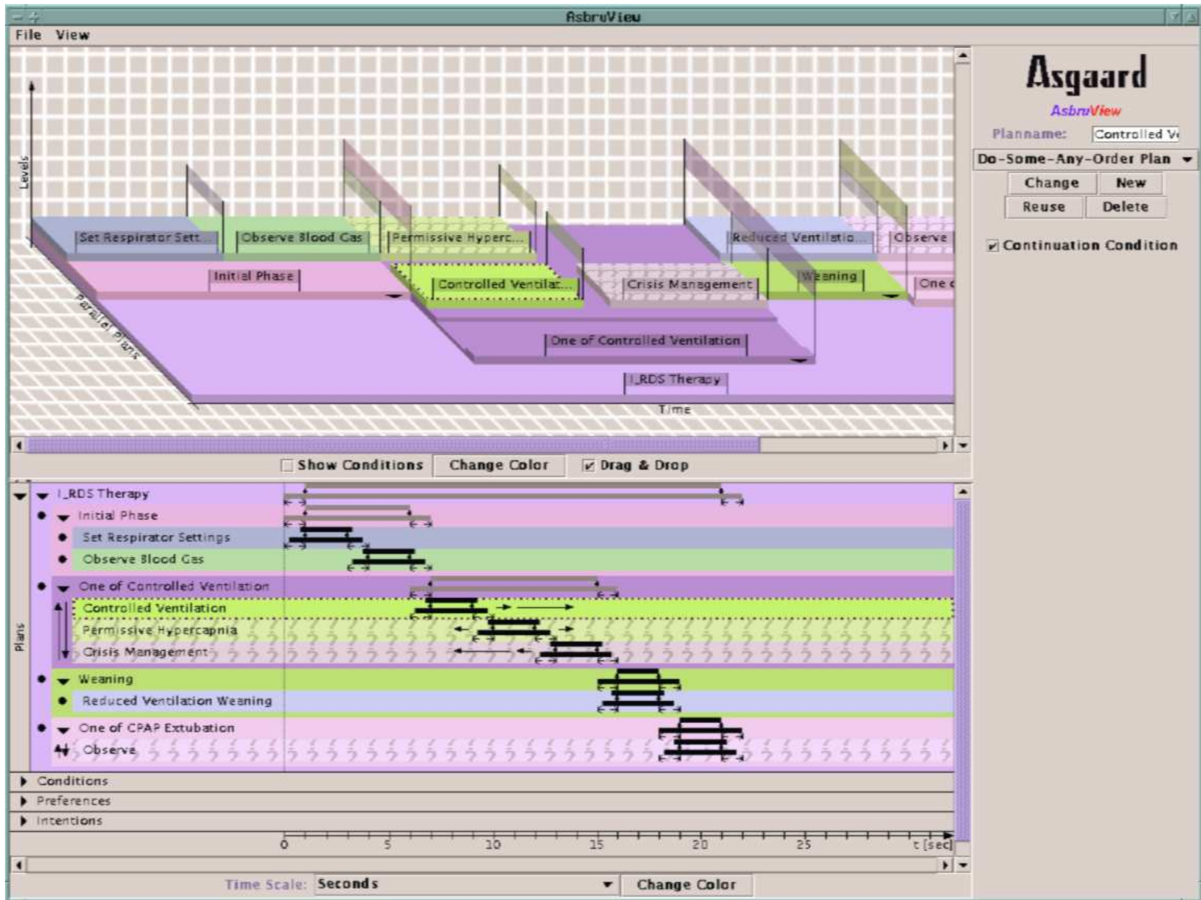

Figure 23: AsbruView user interface.

3.2.4. CareVis

An alternative to *AsbruView* for interactive visualization of protocol-based care plans is the *CareVis* project.

CareVis provides "multiple simultaneous views to cover different aspects of a complex underlying data structure of treatment plans and patient data" (22). There is a temporal view, similarly to *AsbruView*, and a logical view, which is tightly coupled with the first and presents steps on a flowchart view. This is an important feature, since physicians are accustomed to this type of graphical representation of protocols.

Additionally there is the *QuickView Panel*, where the most important patient parameters and plan variables are displayed.

Logical View

The logical view (Figure 24) uses a technique entitled *AsbruFlow*, and provides a representation of the treatment plan specification data. Information is displayed as clinical algorithm maps in the form of flowcharts. The representation is adapted to depict the characteristics of a treatment plan formulated in *Asbru*.

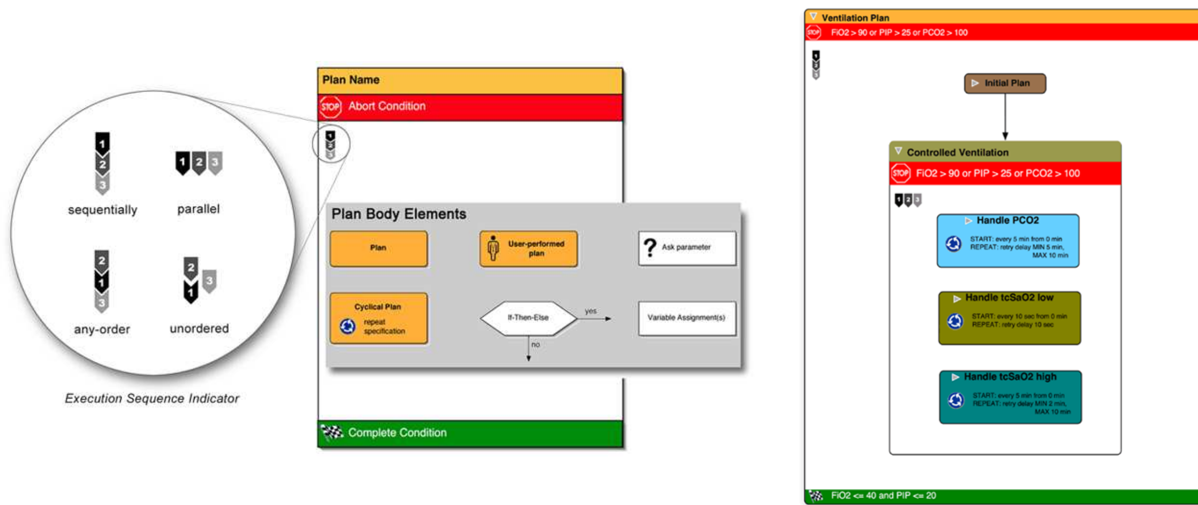

Figure 24: CareVis – Logical View.

Temporal View

The temporal representation of treatment plans is based on the idea of *LifeLines* (Figure 25). Lifelines are essentially similar to the timelines used in *AsbruView, and feature*:

- display of hierarchical decomposition as well as the complex time annotations used in *Asbru*;
- interactive representation of temporal intervals;
- hierarchical decomposition and simple element characteristics;
- encapsulated bars, representing minimum and maximum duration, bounded by two caps that represent the start and end intervals. Encapsulated bars can be shifted within the constraints of two mounted caps.

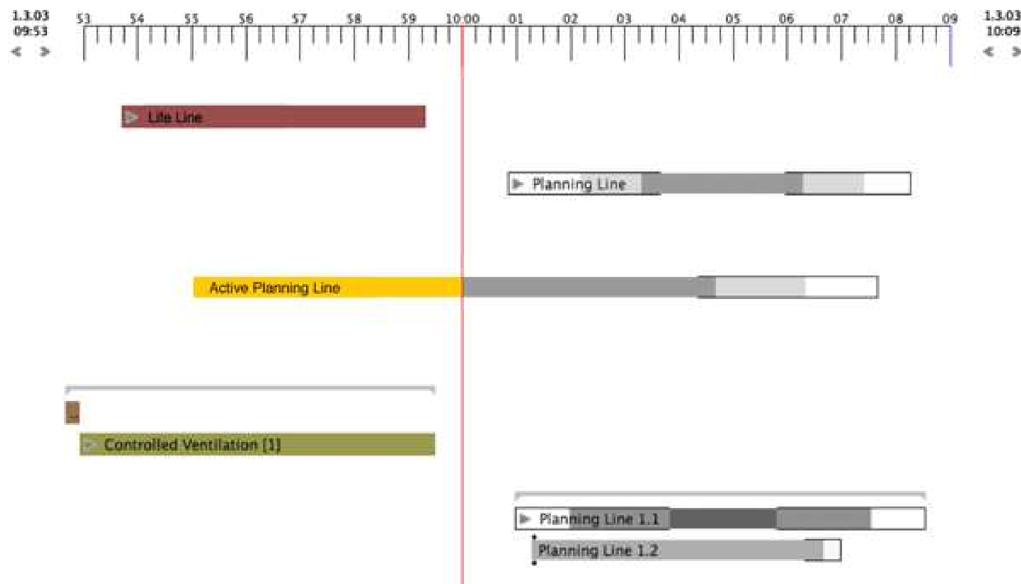

Figure 25: CareVis – Temporal View.

QuickView Panel

The QuickView panel displays currently valid variable and parameter values. The panel consists of rectangular areas that can be assigned to the available parameters and variables. For each parameter, its current value is displayed, along with its name, unit, and a trend indicator. These values are put at a prominent position, enlarged in size and without the need for displaying the complete history in an additional facet.

Figure 26 shows CareVis user interface, containing the logical, temporal and quick view panels.

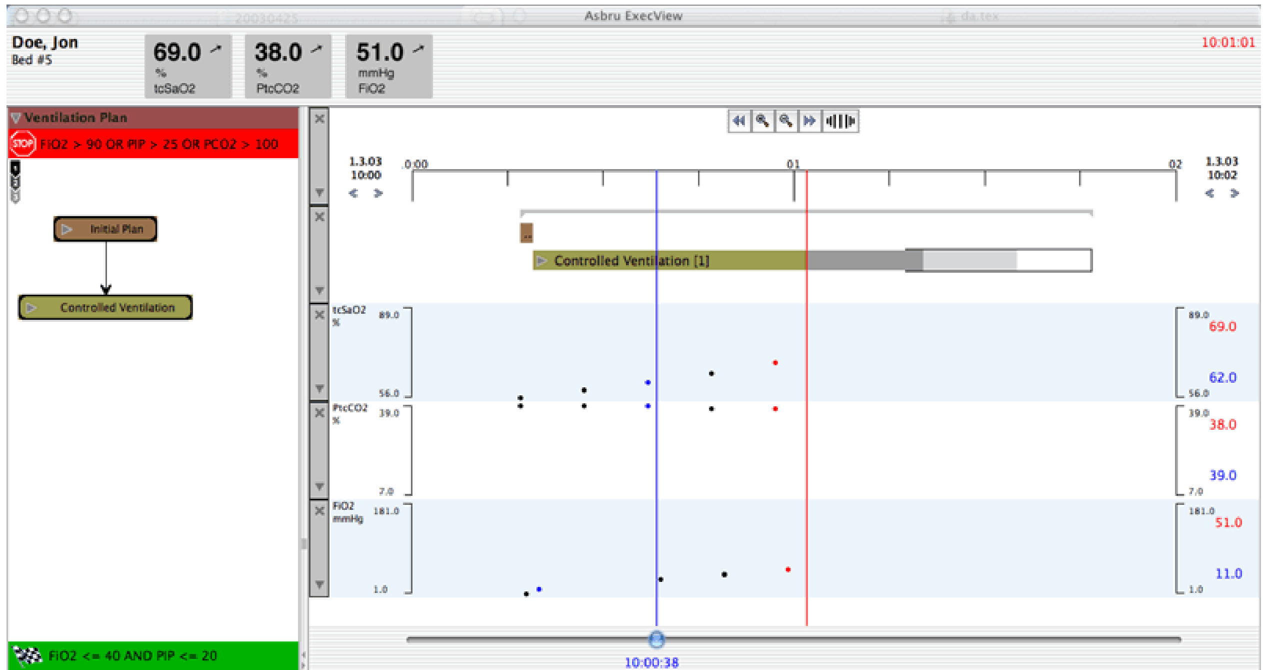

Figure 26: CareVis – user interface.

3.3. Overview

Stanford Medical Informatics researchers arranged a case-study to compare some computer interpretable guideline representation formats: ***Asbru***, ***EON***, ***GLIF***, ***GUIDE***, ***PRODIGY***, and ***PROforma***. The resulting report – *Comparing Computer-Interpretable Guideline Models: A Case-Study Approach*(14) – provides some interesting feature comparisons, revealing differences but also some similarities among formats.

In collaboration with representatives of each group that created the representation models, two test guidelines were represented in each of the languages and eight components and several aspects were compared. As a conclusion, some components were found to be consensual to all models, and good candidates to be part of a standard, while aspects such as decision models, goal representations, use of scenarios, and structured medical actions are approached differently.

Aspects that are common to the different languages include:

1. Organization of guidelines as plans that unfold over time, by linking plan components in sequence, in parallel, and in iterative and cyclic structures, thus defining control-flow.
2. Support for nesting of plans, as well as expression of temporal constraints on plan components.
3. All models would benefit with the creation of a patient information model standard.
4. A standard medical concept model would also be beneficial. However, standardization is currently out of reach since existing vocabularies have not been explicitly designed for clinical decision support and have limitations for such applications. Standardizing definitions of abstract terms would only be possible after a common expression language, patient information model, and medical concept model have been standardized.

The solution proposed for Alert® should take into account aspects 1 and 2, by supporting workflows and sub-protocols (protocols as part of other protocols).

However, and since Alert® already has a complex underlying infrastructure and internal representation methods for patient information and concept models, it would benefit minimally of the use of standards. Not disregarding interoperability, which is actually a key concern for the product, “building a protocol tool with a set of features as complete as possible” will have priority over “building a protocol tool that will easily communicate with other protocol tools”.

There’s also important feedback to be taken from the user interfaces considered. A visually simple and intuitive interface is required, as well as a graphical representation of sequential steps. Also, different components of a protocol should be visually distinguishable.

4. Solution – Clinical Protocols Module in Alert®

In general terms, the proposed solution aimed to the creation of a complete clinical protocols tool that would allow introducing existing protocols in a healthcare software solution, and enacting them according to patients' health records (Figure 27).

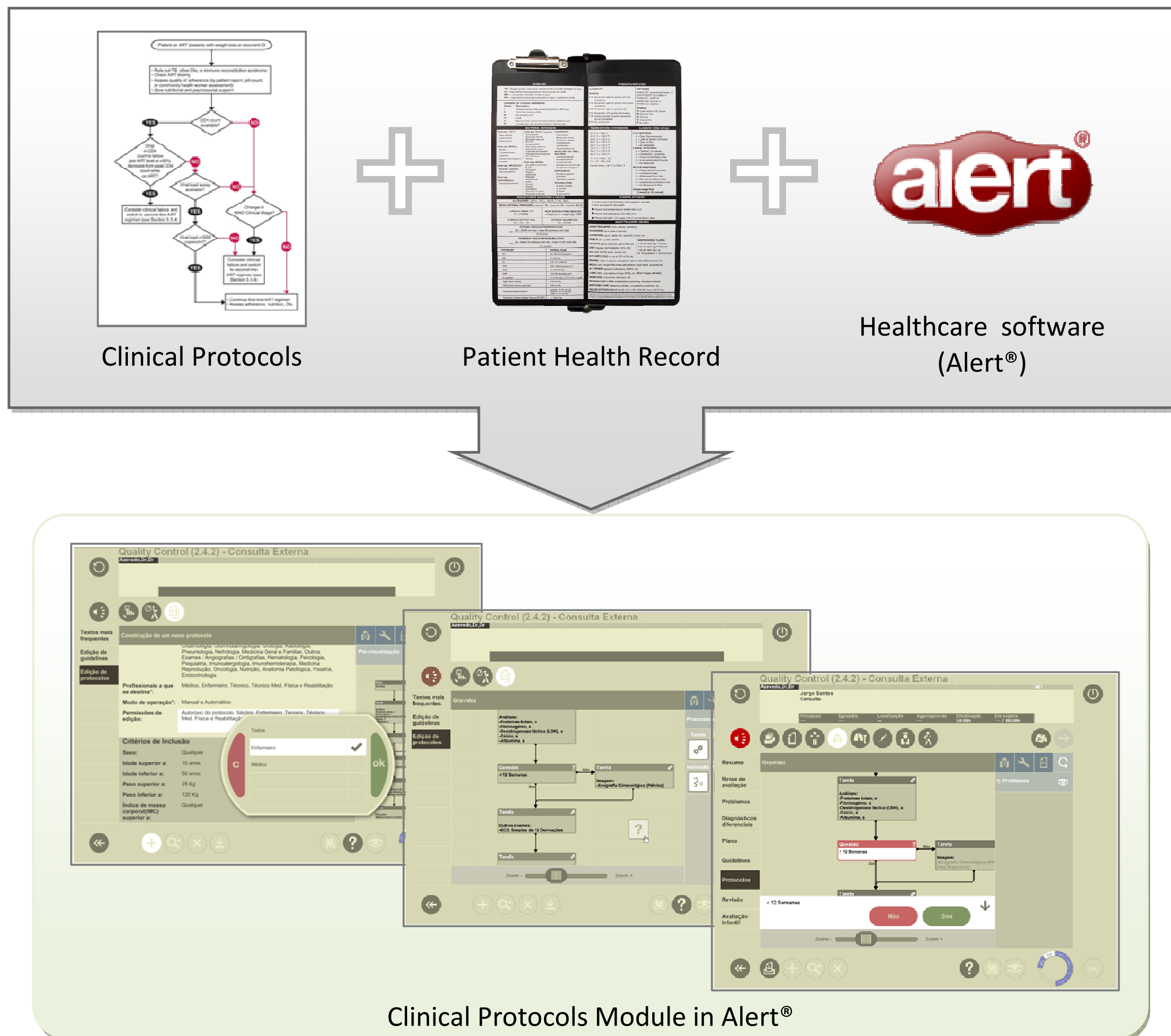

Figure 27: Proposed solution

This section describes all the stages involved on the development of the proposed solution. It starts by describing the requirements and development methodologies adopted, and contextualizing the activities carried out to implement the solution. Finally, it describes the implemented solution itself with implementation details.

4.1. Requirements

To plan the development of the protocols tool, the existing solutions were analyzed and taken into consideration, specifically those described on the State Of The Art section. Nevertheless, the purpose of this work was not to mimic other software's capabilities but fulfill these general objectives:

- create an integrated protocol authoring tool;
- integrate the tool seamlessly with Alert® design and functionalities;
- generate workflow in Alert® from protocol task requests.

Comparing to existing alternatives, some aspects have been simplified such as timeline-oriented views which were considered superfluous for the desired purpose. Other features were further explored such as integration and interactive traversing the protocols.

Developing a protocol creation tool for Alert represented a challenge to user interface development, since it required components unlike any of those already existing in Alert®. These components are described in detail on the following sections, given their importance for this project.

There has been an effort to make the components as generic and independent as possible, so that they can be reused in other contexts if necessary, and to keep the source code conveniently modularized.

Functional requirements have been separated into flow chart framework requirements (section 4.1.1) and protocol tool functional requirements (section 0).

4.1.1. Flowchart framework functional requirements

The following functionalities are expected from this tool, and must be implemented prior to protocols implementation:

Table 3: Flowchart components requirements

Requirement	Description	Priority
Add box elements (nodes) to the diagram	It should be possible to insert new text nodes in the diagram. This can be done without user interaction and through *drag & drop*.	High
Add connectors (directed lines)	It should be possible to insert connectors in the diagram. Connectors must link two box elements and may be added without user interaction and through *drag & drop*.	High
Move box	It should be possible to reposition box elements on the diagram, maintaining the incoming and outgoing connectors.	Medium
Remove box and connector elements	It should be possible to remove elements from the diagram. Removing a box element also removes its incoming/outgoing connectors.	High
Add/Edit box contents (text)	It should be possible to edit the box elements text contents.	High
Add/Edit description to connector elements	It should be possible to attach and edit a description to each connector element.	Medium
Zoom in/out on diagram	Scaling the diagram should be possible, to allow an overall view of it.	Low
Drag diagram	Dragging the diagram vertically and horizontally, allowing the creation of complex flowcharts wider than the viewing window.	Medium
Signal user interactions	Signaling user actions (e.g. press, release, type, drag, resize) is essential so that they may be detected by other components.	Medium

4.1.2. Protocol tool functional requirements

In addition to the items mentioned on section 4.1.1, this module has the following requirements regarding protocol authoring and application:

Table 4: Protocols tool requirements.

	Requirement	Description	Priority
General parameters	**Set main parameters**	Set generic parameters for the protocol: • Title (text) • Pathology (from standards list) • Type of protocol (from list of values) • Environment (from list of values) • Specialty (from list of values) • Professional to which it is destined (from list of values) • Edit Permissions (from list of values)	High
	Set inclusion criteria	Set parameters that if verified by a patient make the protocol applicable to him/her: • Gender • Minimum age limit • Maximum age limit • Maximum Body Mass Index (BMI) • Minimum height • Maximum height • Minimum systolic blood pressure • Maximum systolic blood pressure • Minimum diastolic blood pressure • Maximum diastolic blood pressure • Other Criteria	Medium
	Set exclusion criteria	Set parameters that if verified by a patient make the protocol NOT applicable to him/her. Same fields as inclusion criteria.	Medium
	Set related information	Define content that is related with the protocol: • Context help • EBM • Title • Adapted by • Media type • Author • Editor • Edition Place • Edition • Edition date • Availability and access • Original language • Image • Subtitle • Language	Low

Edit diagram	**Add different types of box elements:**	Different diagram elements should be available when building a diagram: • Title (the starting element) • Question (a decision node) • Task (a task recognized by alert) • Instruction (textual instruction) • Protocol (link to another existing protocol)	High
	Add connectors with pre-defined values	It should be possible to add a connector to the diagram, with a list of possible values to choose from	Medium
Assign protocol to patient	**Automatically**	Automatically recommend a protocol to a patient by matching criteria with patient attributes	Medium
	Manually	Assign a protocol to a patient manually, selecting it either: • Through free-text search • Browsing categories	High
	Go through protocol steps, requesting tasks	Follow protocol steps interactively, generating workflow when requesting tasks.	Medium
	View protocol details	The details of the parameters defined for a protocol, its author(s) and editing history should be made available.	Low
	View status for protocol / tasks	It should be possible to view the current and past status of each protocol applied to a patient and the corresponding tasks.	Medium

4.1.3. Restrictions and Validations

Complementing the required features, the following restrictions must be verified:

- A protocol cannot be saved if any of the mandatory fields are left unset;
- A protocol must have at least a Title element, and a second element connected to the Title;
- Multiple task selection is restricted to tasks that can be executed concurrently on the front office;
- Tasks that are part of a protocol but that are not available at the institution in which it's being applied should be distinguishable and should not be possible to request;
- Professionals should only be able to execute a task if they have the required permissions. Other tasks should be distinguishable and impossible to request.

4.1.4. Use cases

Figure 28 summarizes the expected use cases for the tool according to the different user permissions and profiles.

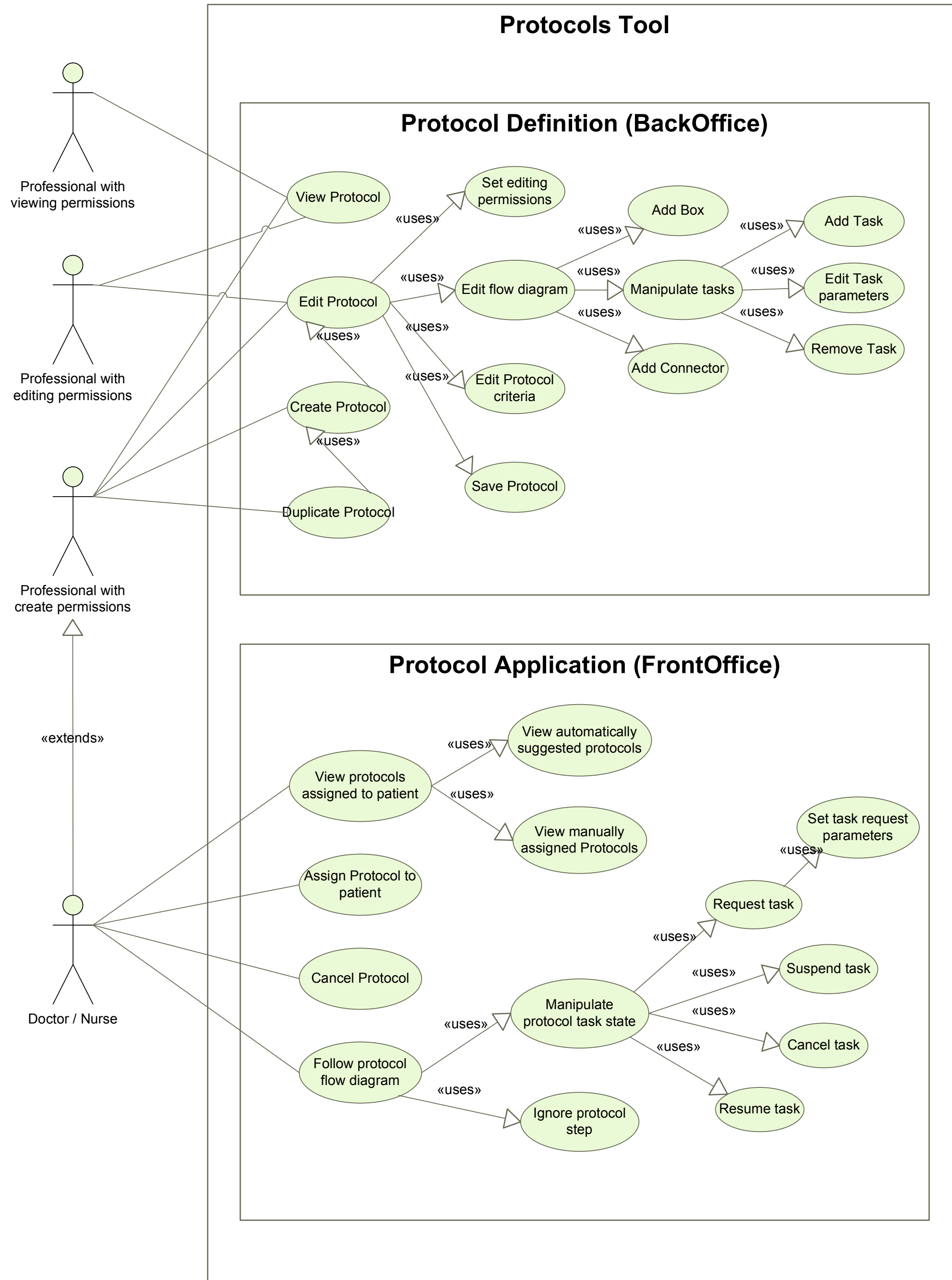

Figure 28: Protocols Use Case Diagram.

4.2. Development Methodologies

There are several processes at different levels of any Alert® product life cycle. The most general process involves a number of steps from requirements definition to product installation and maintenance, as described next.

4.2.1. Product Life Cycle

The life cycle of any Alert® product consists of a sequential set of phases as illustrated on Figure 29 (2).

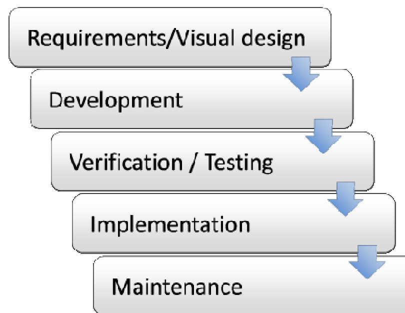

Figure 29: Product Life Cycle.

These steps are executed sequentially and each step requires prior completion of the preceding one.

Requirements / Visual Design

All development originates from a set of requirements, which can have distinct proveniences:

- End-users requests for features. These are usually compiled by the requirements team;
- Identification of potentially useful features from inside the company (e.g. Product Managers);
- Necessities imposed by quality standards.

Requirements from all these sources are analyzed, filtered, compiled and prioritized, resulting on a list of features and use-cases to be implemented, with associated priority levels.

The design team is responsible for taking the requirements list and creating documentation defining all style related issues, namely:

- screen layouts;
- colors, sizes, positions of components;
- component functionality and interaction;
- icons to be used.

These items are compiled on a set of drawings with associated comments to clarify functionalities. These sets are divided in different documents, typically one for each functionality.

Development

On this stage, development teams take the drawings produced by the design team and implement the infrastructure that will provide the required functionalities. This process involves developers from all

three layers: Interface, Java and Database. The Database team creates (or updates) the data model to accommodate the new features and creates the needed data access functions. Java developers then generate the corresponding packages and Interface developers create the Flash structures that will use those packages. This is typically processed iteratively and in this order, due to the dependencies between layers (Figure 30).

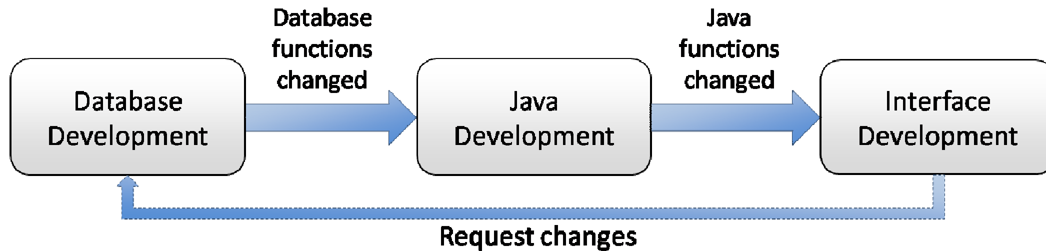

Figure 30: Development workflow for the 3 software layers.

Although the new components will undergo integration tests on the verification stage, it's the developers' job to unit test the new features. This will greatly reduce further problems and unexpected behavior.

Most of the work described on this document took place on this stage of the development, more specifically on the Interface development phase.

Verification / Testing

In spite of the great effort from the development team to minimize errors, minor *bugs*[10] and malfunctions are likely to occur after the development stage finishes. It's the Tests department responsibility to thoroughly test the developed features, to guarantee that the application functions as expected before it's deployed.

This is a vital stage, with special importance in software as critical as medical frameworks.

Implementation

Implementation consists in all activities related to the installation of Alert® on a client institution. It involves software and hardware configuration, and the setup of all the necessary interfaces with existing solutions already in use at the institution.

Maintenance

After a version of the software is developed and tested, it's released and implemented at the client institution. Ideally, no further action should be required from the development team, at least until the following version is released. The clients can however provide feedback and suggestions for improvements, or report errors they run into. These are reported back to the requirements team, who will prioritize the reported issues and start the cycle over again with the new necessities

[10] a fault or defect in a system or machine

4.2.2. *Development Process*

This project took place on the development stage of the product life cycle. At this stage, and since the Alert® suite is currently a large scale product made of a diversity of modules, development is made iteratively according to the Evolution Model described on Figure 31.

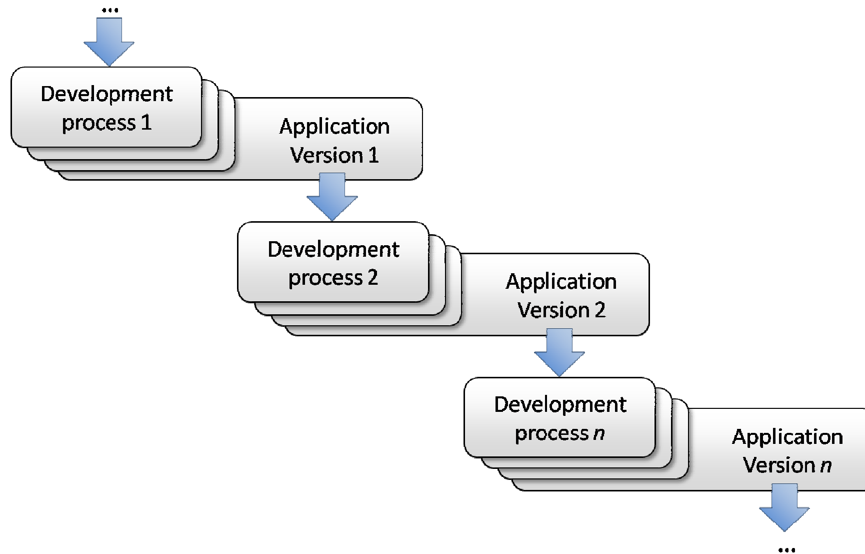

Figure 31: Global product development process – Evolution Model.

After a version of the product is released, development for the next version starts. Each team works independently on the module of their responsibility, and maintains its own development process. Each team's adopted development model is decided by the respective product manager, and depends mainly on the nature of the modules being developed, and on the available team elements.

For the development of the Protocols module, the Content team followed the model on Figure 32, which is based on the Waterfall Model. It involves a set of well defined steps, from requirements definition to product deployment and stabilization, with different alternative workflows.

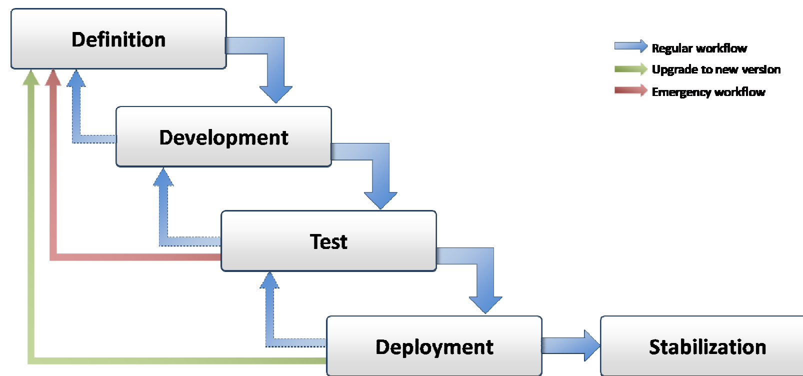

Figure 32: Content project development process – modified Waterfall Model.

This development module is essentially the well known Waterfall Model where each phase starts after the previous is complete, ideally with no need to return to a previous stage. However, in situations where agile development is needed, or when minor conflicts arise on the Implementation stage that don't affect software design, the emergency workflow (red arrows) is used.

4.3. Context of the developed work

The Content Development team assigned to these project consisted of four programmers, two of them Oracle/PLSQL programmers, and other two Flash/Actionscript programmers. Since the Java layer is semi-automatically updated when changes are made to the database, there wasn't a permanent Java programmer in the team. Instead, a request was sent to the Java team each time changes were necessary. The team proved to be very cohesive and organized, maintaining good communication, which made development easier and faster.

The work developed was focused on the Development stage, specifically on User Interface (Flash/ActionScript) programming (Figure 33). This section describes mainly the Interface related procedures, since database is kept transparent to the User interface.

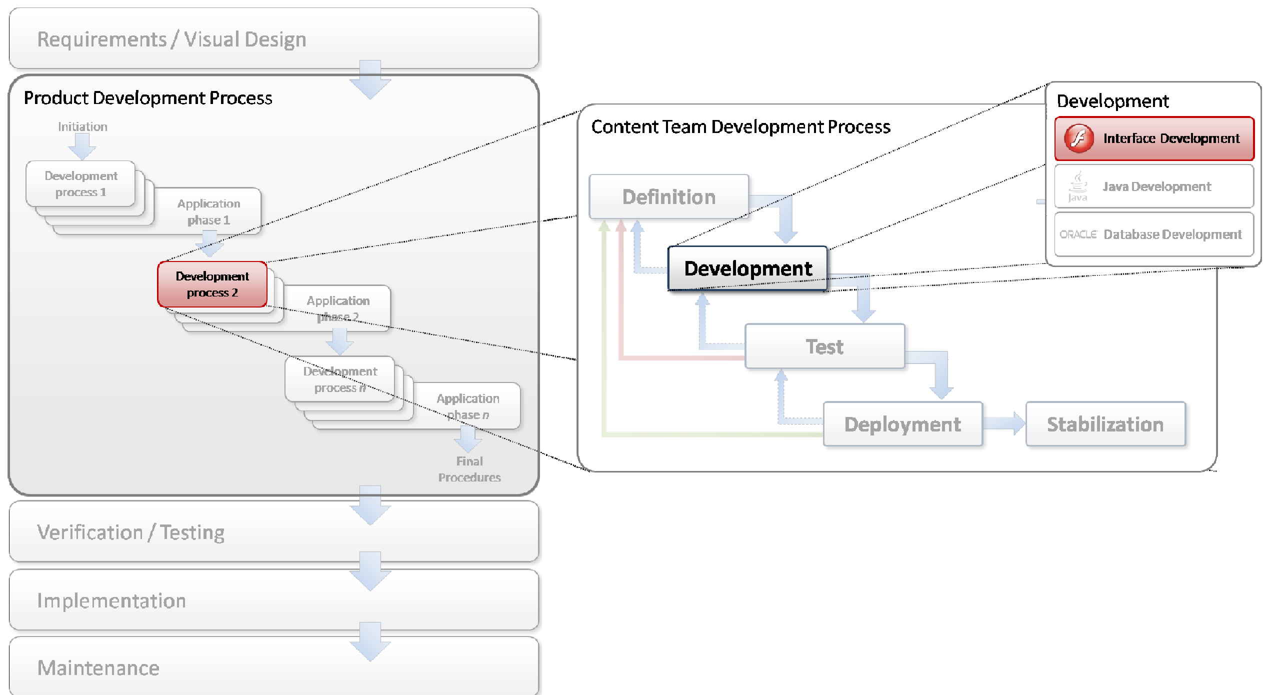

Figure 33: Context of the project.

4.4. Flowchart Development

The development of a flowchart framework consisted essentially on the creation of several independent components that interrelate with each other. Section 4.4.1 explains the process inherent to component development, while 4.4.2 presents details of the developed components themselves.

4.4.1. Component development process[11]

A *component* is a movie clip with parameters that are set during authoring in Macromedia Flash, and with ActionScript methods, properties, and events that allow you to customize the component's appearance and behavior at runtime. Components are designed to allow developers to reuse and share code, and to encapsulate complex functionality that designers can use and customize without using ActionScript. Flash comes bundled with a set of customizable components such as Buttons and radio boxes, and allows the creation of reusable user-built components.

A component typically consists of a Flash (FLA) file and an ActionScript (AS) file. The FLA file contains a movie clip symbol that must be linked to the AS file in both the Linkage Properties and the Component Definition dialog boxes.

The movie clip symbol has two frames and two layers. The first layer is an Actions layer and has a stop() global function on Frame 1. The second layer is an Assets layer with two keyframes: Frame 1 contains a bounding box; Frame 2 contains all other assets, including graphics and base classes, used by the component.

[11] Abridged from Flash Documentation (25)

The ActionScript code specifying the properties and methods for the component is in a separate ActionScript class file. This class file also declares which, if any, classes the component extends. The name of the AS class file is the name of the component plus the ".as" extension. For example, MyComponent.as contains the source code for the MyComponent component.

A component's AS class must contain some mandatory methods and parameters

```
1.  import mx.core.UIComponent;
2.
3.  class Dial extends UIComponent
4.  {
5.      // Components must declare these to be proper
6.      // components in the components framework.
7.      static var symbolName:String = "Dial";
8.      static var symbolOwner:Object = Dial;
9.      var className:String = "Dial";
10.
11.      // The private member variable "__value" is publicly
12.      // accessible through implicit getter/setter methods,
13.      private var __value:Number = 0;
14.
15.     // Constructor;
16.     // Constructor must be empty with zero arguments.
17.     // All initialization takes place in a required init()
18.     // method after the class instance has been constructed.
19.     function Dial() {
20.     }
21.
22.     // Initialization code:
23.     // This method is required for components extending UIComponent.
24.     // It must call its parent class init() method with super.init().
25.     function init():Void {
26.         super.init();
27.         // other initializations here
28.     }
29.
30.     // Create children objects needed at start up:
31.     // Required for components extending UIComponent.
32.     public function createChildren():Void {
33.         size();
34.     }
35.
36.     // The draw() method is required for v2 components.
37.     // It is invoked after the component has been
38.     // invalidated by someone calling invalidate().
39.     // This is better than redrawing from within the set() function
40.     // for value, because if there are other properties, it's
41.     // better to batch up the changes into one redraw, rather
42.     // than doing them all individually. This approach leads
43.     // to more efficiency and better centralization of code.
44.     function draw():Void {
45.         super.draw();
46.     }
47.
48.     // The size() method is invoked when the component's size
49.     // changes. This is an opportunity to resize the children.
50.     // This method is required for components extending UIComponent.
```

```
51.         function size():Void {
52.             super.size();
53.             invalidate();
54.         }
55.
56.         // Getters and setters for component properties
57.         // The [Inspectable] metadata makes the property appear
58.         // in the Property inspector. Calling invalidate
59.         // forces the component to redraw, when the value is changed.
60.         [Bindable]
61.         [ChangeEvent("change")]
62.         [Inspectable(defaultValue=0)]
63.         function set value (val:Number)
64.         {
65.             __value = val;
66.             invalidate();
67.         }
68.
69.         function get value ():Number
70.         {
71.             return __value;
72.         }
73.
74.     }
```

4.4.2. Flowchart component development

To implement flowchart capabilities in Alert, five components were created:

- **Box**
- **Connector**
- **Stage**
- **ProtocolElement**
- **ZoomBar**

These components are independent but interact together to create and display flowchart diagrams.

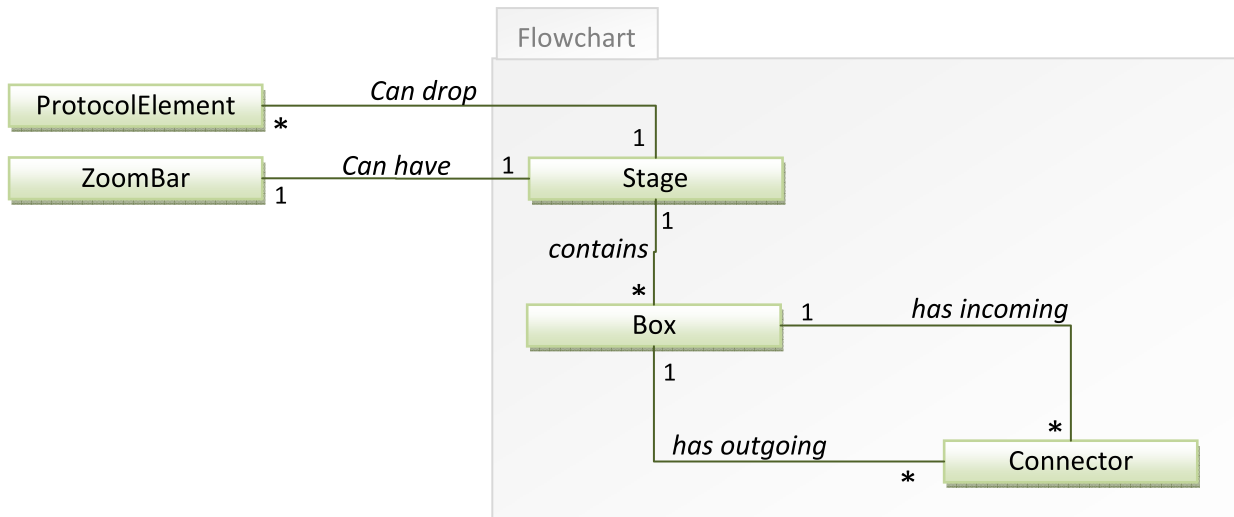

Figure 34: Flowchart components interaction.

Box

The *Box* element is a central component of the flowchart. It represents a node in the diagram to which connectors can be attached to create a directed graph. In a protocol context, this element represents a stage of the protocol that may pose a question to determine which path to follow, or contain instructions or tasks that need to be performed before proceeding.

Apart from the background graphics defining the header background color, border color and box background color, these component contains the following elements:

- A **title text field**;
- An **editable text area**, for user input information;
- A **static text area**, to display non-editable contents;
- A customizable **icon**.

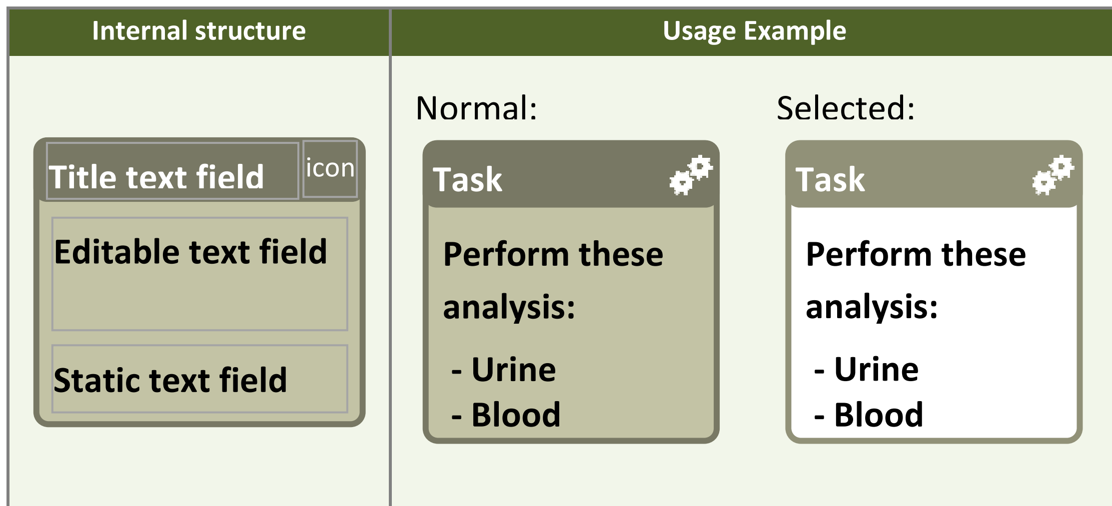

Figure 35: Box component.

This component exposes a series of properties that control its appearance and behavior. These may be set at compile time, through the authoring tool, and/or changed at runtime in response to user interactions.

Table 5 lists the editable properties for the Box component.

Table 5: Box parameters.

Parameter	Type	Description	Default value
static_content	String	Sets the text on the static text field. The box height is adjusted as necessary.	<empty string>
content	String	Sets the editable text field contents. Unless editable is set to false, this content can be changed by direct user text input.	<empty string>
selected	Boolean	Gets or sets selected state on the box.	false
borderColor	Number	Defines color for the border and title background (when box is not selected)	0x787864
borderSelectedColor	Number	Defines color for the border and title background when box is selected.	0x919178
editable	Boolean	Gets or sets editable state of the box. If true, box content may be edited through user inputs.	False

Other components need to be aware of changes performed to the Box component, e.g. to readjust size/position or trigger other actions. The Box element dispatches a series of events that may be caught and externally processed, as shown on Table 6.

Table 6: Box element events.

Event name	Description
boxPressed	Dispatched when a mouse button is pressed over the box's visible area.
boxPressedTwice	Dispatched when a mouse button is pressed twice over the box's visible area, within a 500 millisecond interval (double-click).
boxResized	Dispatched when the visible area of the box changes.
boxUnselected	Dispatched when a box loses focus (becomes unselected)
boxSelected	Dispatched when a box gains focus (becomes selected)

Connector

The connector element is a directed line that will define possible flows for the diagram. A connector must always be attached to two Box elements, one at each end, except when it's being drawn. The source and destination box may be one and the same.

The connector automatically "reroutes" itself according to the relative positions of its adjacent boxes. It's however unaware of the existence of other Connectors or Boxes on the same diagram, and therefore does not avoid collisions with other objects. This has been marked as a future improvement to the

component, but at the moment the user must take into account the Boxes positioning to avoid connector overlays. The fact that Boxes may be easily reorganized through *drag & drop* eases this task.

Figure 36 shows all possible relative box positions, and illustrates the way connectors are drawn on each of the cases.

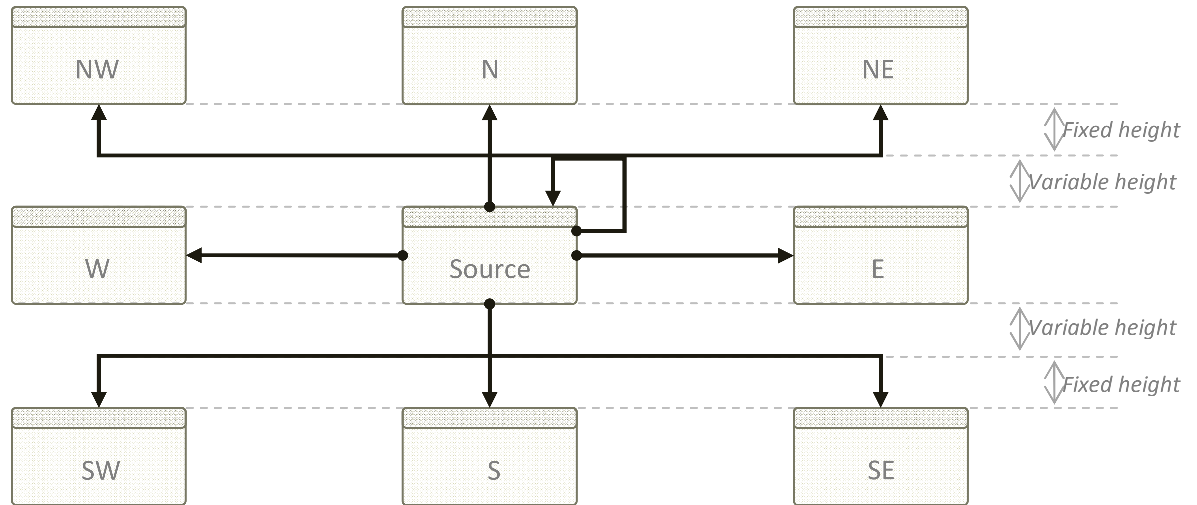

Figure 36: Connectors.

As the Box component, a Connector may be manipulated through the properties it exposes, as shown on Table 7.

Table 7: Connector parameters.

Parameter	Type	Description	Default value
`text`	String	Gets or sets the connector description.	`<empty string>`
`startObject`	Object	Gets or sets the element where the connector starts.	`null`
`endObject`	Object	Gets or sets the element where the connector ends.	`null`
`selectable`	Boolean	Gets or sets selectable property, indicating if the connector can be selected (catch focus).	`false`
`editable`	Boolean	Gets or sets editable property, indicating if the connector text can be edited by the user.	`false`

Because it's necessary to detect when a user selects a connector (by pressing it), and since visually this component is a thin line, there is an area surrounding the connector that is not visible to the user but which is responsible for capturing mouse clicks, as shown on Figure 37.

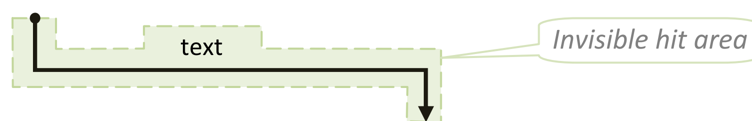

Figure 37: The connector hit area.

Events dispatched by the Connector simply signal if it was pressed or released (Table 8).

Table 8: Connector element events.

Event name	Description
`connectorPressed`	Dispatched when a mouse button is pressed within the area surrounding the connector or its text.
`connectorReleased`	Dispatched when a mouse button is released within the area surrounding the connector or its text.

Stage

The Stage component works as a container for the flowchart boxes and respective connectors. It's used to arrange the elements on a regular grid, thus keeping the diagram organized. Elements can be added to the stage in one of the following ways:

- Without user intervention, through invocation of the appropriate ActionScript method, e.g. when a previously built diagram is open;
- Dynamically, e.g. dragging a ProtocolElement into the appropriate position (See ProtocolElement details on section 0).

Zooming and Panning operations performed on the diagram are interpreted by the Stage, which is responsible for the required transformations.

Upon creation, the Stage is configured by setting variables such as the boxes width (the same for all boxes), minimum height and spacing. Although the visual appearance of the Stage component is just a solid color, the component class defines an underlying structure that is used for element positioning as shown on Figure 38.

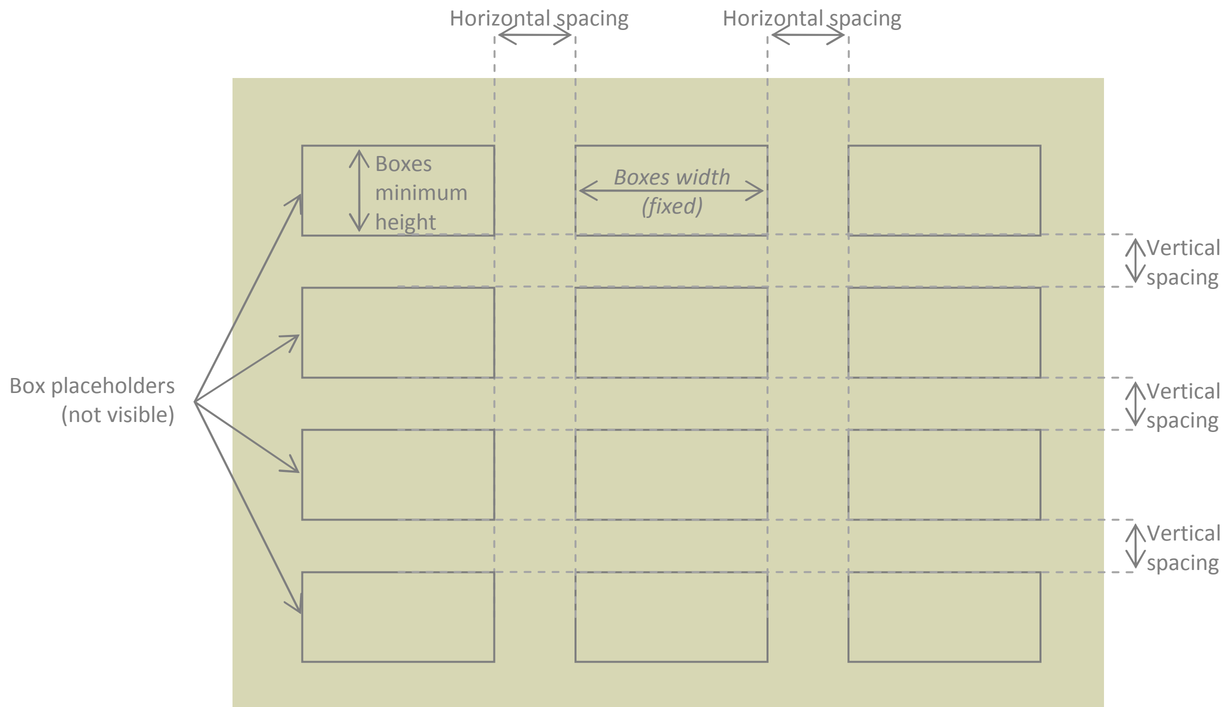

Figure 38: Stage component.

As the other components, the Stage has its own properties that make it configurable (Table 9). Some Box parameters are common to all Box components contained on the stage. Thus, parameters like `boxTitleTextFormat`, `boxesWidth` can be set on the Stage component and be reflected on the Boxes it contains or those subsequently added.

Table 9: Stage parameters.

Parameter	Type	Description	Default value
editable	Boolean	Gets or sets editable property, indicating if the stage contents can be edited by the user.	`false`
boxTitleTextFormat	TextFormat	Gets or sets the text format for the box title text.	
boxContentTextFormat	TextFormat	Gets or sets the text format for the box contents.	
backgroundColor	Number	Gets or sets the stage background color.	`0xC3C3A5`
borderColor	Number	Gets or sets the stage border color.	`0xEBEBC8`
boxesWidth	Number	Gets or sets the width for the boxes in the stage.	`128`
boxesMinimumHeight	Number	Gets or sets the minimum height for the boxes in the stage. The actual height of each box depends on its contents but is never below this parameter value.	`64`
stageHeight	Number	The total height of the stage in pixels.	`768`
stageWidth	Number	The total width of the stage in pixels.	`576`

The Stage dispatches its own event notifications, such as `backgroundPressed` or `elementReleasedOnStage` but it also forwards Events received from underlying objects, such as `boxSelected` or `connectorPressed` , which may be useful for whatever screen contains the flowchart (Table 10).

Table 10: Stage element events.

Event name	Description
`boxSelected`	Dispatched when some box contained in the stage is selected.
`boxUnselected`	Dispatched when some box contained in the stage is unselected.
`connectorPressed`	Dispatched when any connector contained in the stage is pressed
`connectorInserted`	Dispatched when a connector is inserted on the stage
`elementReleasedOnStage`	Dispatched when a ProtocolElement is dragged and dropped on the Stage.
`backgroundPressed`	Dispatched when the background of the stage is pressed.

Apart from the background color, the stage has no other visible components. Box placeholders are dynamically resized and used internally for box positioning but are not displayed. Figure 39 shows the stage component containing some example box and connector components.

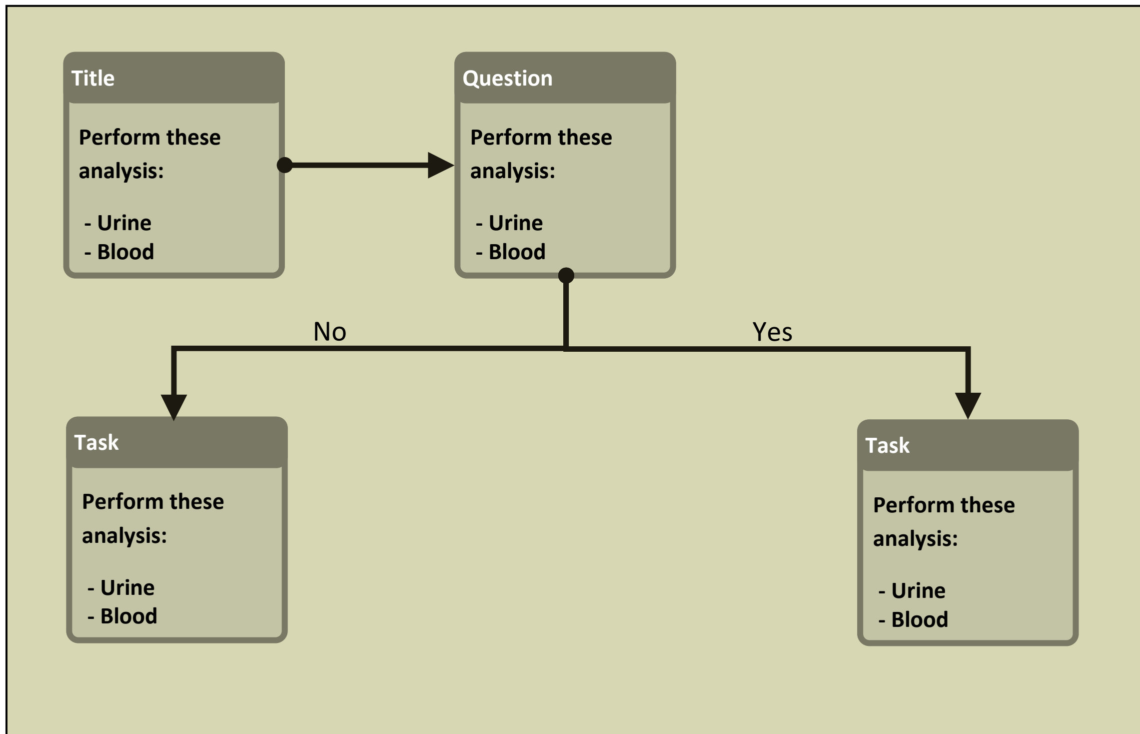

Figure 39: Stage with some Boxes and Connectors in place.

ZoomBar

A flowchart can become extensive and as a result difficult to visualize as a whole. On the other hand, trying to view the entire diagram by reducing the components' size would make text unreadable. For these reasons, a zooming tool is required to provide a flexible way of visualizing the diagrams.

Other areas in Alert® already used a zooming widget for the same purpose, and this component has been adapted so that it could be used with the flowchart components. Graphically, the component resembles a horizontal scrollbar, where scrolling left will zoom out, and scrolling right zooms in (Figure 40).

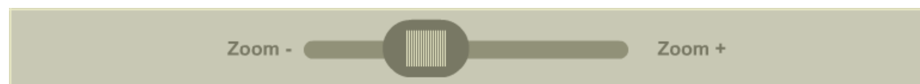

Figure 40: ZoomBar component.

This component takes an object (Table 11) whose `_xscale` and `_yscale`[12] properties will be manipulated proportionally, as the scroll is dragged left or right.

[12] `_xscale` and `_yscale` are `MovieClip` properties that define its horizontal and vertical scaling, respectively.

Table 11: ZoomBar parameters.

Parameter	Type	Description	Default value
zoomObject	Object	The object to zoom when the scroll is dragged	`null`

The ZoomBar dispatches an event signaling that the zooming factor has changed, when the scroll is dragged (Table 12).

Table 12: ZoomBar events.

Event name	Description
`zoomChanged`	Dispatched when the zoom factor on the object is changed (by dragging the scroll).

4.5. Protocols Tool Development

The developed modules are part of Alert® core functionalities, as schematized on Figure 41.

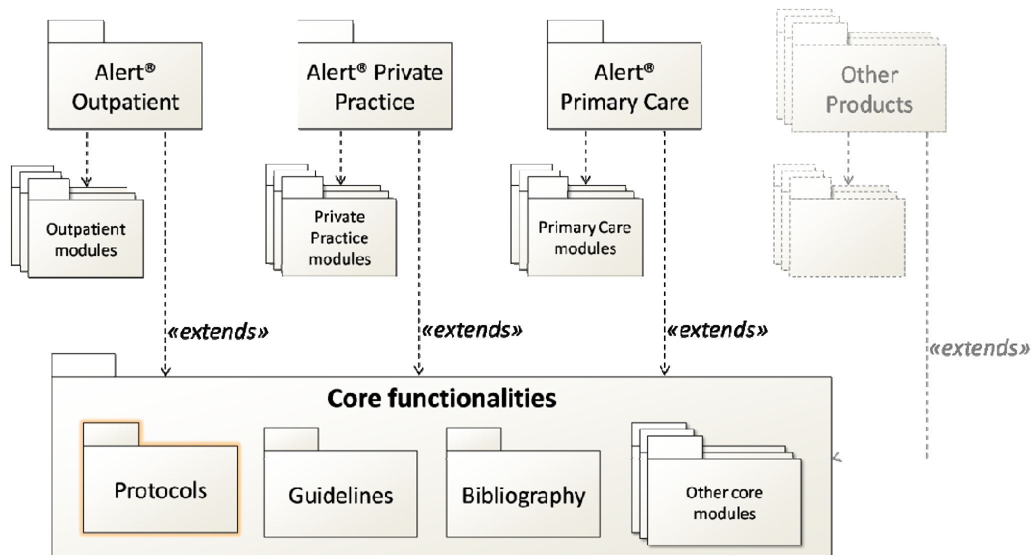

Figure 41: Alert® package diagram.

From the user interface perspective, each package represents a set of screens that fit together coherently. Some screens are specific to particular products, others, such as the Protocols screens, are shared by all or some of the Alert® products. In spite of being core functionalities, Protocols are parameterized to be used by Alert® Outpatient, Alert® Private Practice and Alert® Primary Care. This customization is made on the database, and no changes will be required to the interface to make these modules available to other products in the future.

Alert®'s User Interface is made up of SWF files, which interrelate in different ways. After successful authentication and selection of the institution (a user may have access to multiple institutions), the main screen for the selected environment (Outpatient, Inpatient, ER, etc…) is loaded.

Once logged in, most of Alert® screens have the layout shown on Figure 42, and share components such as the Header, DeepNav and Action Buttons.

Figure 42: Alert® screen components.

Every different screen in Alert® has a corresponding SWF file, which defines the main screen area contents. This file is loaded by the application controller upon request. The *Viewer* is a separate SWF file that interacts directly with the main screen area, showing relevant information. It also provides access to utilities without the need to change the main screen.

The *header* shows summarized patient information. The application buttons are shortcuts to different areas of the application and the *DeepNav* allows navigation through the different sections of those areas. Action buttons perform actions over the screen contents such as record insertion and deletion, printing or viewing context help.

Of the described elements, the most relevant and the ones who were directly affected by the work developed are the *main screen area* and the *Viewer*. Each of the modules developed consist on a set of different application screens that are available in all Alert® software solutions.

4.5.1. The Screen Creation Process

Creating a screen for Alert® (2) is not different from creating any other flash movie. The tutorial-like steps described next are the basis of all the more complex programming tasks that were part of this work. They are included as an illustration of how interface logic and design are linked in Flash.

1. **Creating a new Flash document (*.FLA) on the Flash authoring environment.**

The *.FLA file contains all the components that will make up the screen. It also defines the target directory and file name for the compiled .SWF file.

2. **Importing the necessary pre-compiled components and movie clips to the library.**

The library holds all the objects, components and movie clips that can be used on the screen (Figure 43). They can be either dragged into the movie canvas to create instances or dynamically imported using ActionScript commands.

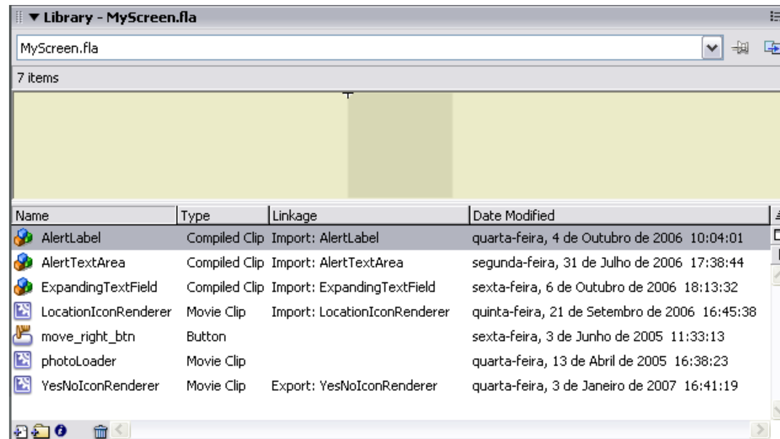

Figure 43: Flash Authoring Application: the Object Library.

3. Inserting the components in the document

Most elements used in Alert® are pre-compiled components and static, so there's no reason to create them at runtime[13]. Instead, they are placed and resized as appropriate within the authoring environment. These visually added elements can still be accessed from ActionScript and edited in runtime.

The objects dragged into the screen can be organized in different layers (Figure 44), which can be hidden, locked or masked with other layers as desired. Layers are useful to keep elements visually organized and modularized. However, the existence of layers is transparent to ActionScript. It accesses elements by their instance name, and objects created in runtime from ActionScript classes are placed on a common layer.

It's a common practice, although not mandatory, to create a layer (conventionally named *actions*) exclusively for ActionScript code.

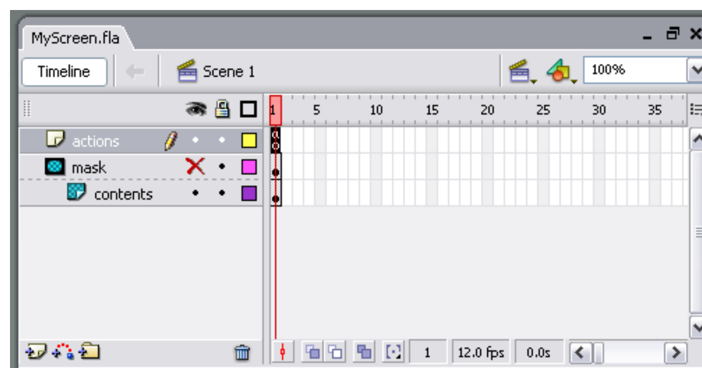

Figure 44: Flash Authoring Application : layers.

[13] the period of time during which a program is executing

4. Creating an ActionScript class with the same name as the *.FLA document.

Associated with the .FLA authoring file will be an ActionScript class with the .AS extension. For organization purposes, this file commonly has the same name as the authoring file. It will contain all the logic inherent to the screen, and uses functions, variables and event handlers in an object oriented approach. A very simple class file can have these contents:

```
66: /* Imports */
67: import mx.utils.Delegate;
68:
69: class MyScreen extends MovieClip {
70:
71:    /* Class Variables */
72:    private var label_tf : TextField;
73:
74:    /* Class Constructor */
75:    public function MyScreenClass() {
76:
77:            /* create references to objects on stage */
78:            label_tf = _root["myLabelInstanceName"];
79:
80:            /* Add event listeners */
81:            label_tf.onRelease=Delegate.create(this,myFunc);
82:    }
83:
84:    /* Other functions and event handlers */
85:    private function myFunc () {
86:            Label_tf.text = "New text for the label";
87:    }
88:
89: }
```

This example represents a basic structure for a class file. The first instructions on the file are the necessary imports. On this example we import the `Delegate` class from Flash core libraries. Then we declare the class itself and any classes it inherits (on this case the `MovieClip` class). Class variables should be declared before everything else. On line `72:` we declare an object of the type `TextField`, which will reference a text field that we previously inserted on the stage.

The constructor function has the same name as the class, and is executed when a new instance of this class is created, by invoking:

```
new MyScreen();
```

Inside the constructor we make any initializations that are necessary, like assigning stage objects to class variables and attaching event listeners to objects as exemplified on lines `78:` and `81:`, respectively.

Event handling functions, and any other functions used by the screen can be defined inside the class. They can be private (available only to functions within the same class) or public (accessible from other classes) and have access to the class variables previously defined, as lines 85: to 87: show.

There are other functions that are commonly found in every Alert® screen:

- `buttonClick()` : This function executes whenever an action button is pressed, and receives the code for the button. This code is then used to decide which action to take.

- `onUnloadScreen()` : This function is called when the user leaves the current screen. It contains instructions to delete unnecessary references and form variables[14]
- `buttonUpdate()` : This function concentrates all action button states logic. It's explicitly invoked when an action that can affect button states is performed. It then sets every button as active, inactive or not available according to a set of pre-verified conditions.

5. Importing the new class, creating a new instance

The association of the screen to the corresponding class is accomplished by importing the class file and creating a reference to it. For example, to include the ActionScript MyScreen, the following instructions would be added to the actions layer of **MyScreen.FLA** :

```
1:  import mypackage.screens.MyScreen;
2:
3:  new MyScreen();
```

The class is first imported, and then instantiated by invoking it's constructor. This code is executed when the MyScreen.SWF (see step 6) file is loaded.

6. Compiling the document to create a SWF file.

The final step is compiling the screen (compile is also called *publish* in Flash). This will produce a compressed and uneditable file with the SWF extension (Figure 45). This object can then be viewed with a Shockwave player, load and be loaded by other SWF files.

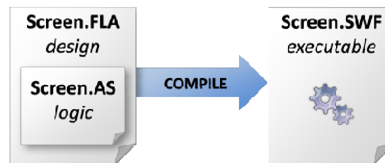

Figure 45: Compiling Flash files.

[14] Form variables are kept in memory. Once set, with setFormVar(var_name, value), they become accessible from any screen and are deleted only when exiting Alert or when explicitly removed with deleteFormVar(var_name). Form variables are used to pass information between different screen classes.

4.5.2. *Protocol BackOffice development*

Protocols are created and edited on the application BackOffice. There, parameters such as general protocol details, inclusion and exclusion criteria and other related information are set on a first screen (Table 13).

Table 13: Protocol parameters.

Protocol Parameters		
Protocol Details	**Inclusion/exclusion Criteria**	**Other Information**
Title Pathology Type of protocol Environment Specialty Professional to which it is destined Edit Permissions	Gender Minimum age limit Maximum age limit Maximum Body Mass Index (BMI) Minimum height Maximum height Minimum systolic blood pressure Maximum systolic blood pressure Minimum diastolic blood pressure Maximum diastolic blood pressure Other Criteria	Context help EBM Title Adapted by Media type Author Editor Edition Place Edition Edition date Availability and access Original language Image Subtitle Language

Moving on to the next screen, the flow diagram associated with the protocol can be created. This flow chart editing screen gathers all the components mentioned on section 4.4.2 to provide a diagram authoring tool. There, different types of "nodes" can be added to the diagram, by dragging the corresponding ProtocolElement objects:

- **Question:** A node of this type is meant to contain a question that is input as free text. Connectors originating from a Question node represent the possible answers to it.
- **Warning:** A Warning node is meant to have some kind of relevant notice to the user. This warning is input as free text.
- **Instruction:** This type of node, also input with free text, provides some instruction that does not fit within any of the available tasks.
- **Protocol:** This node allows the user to insert a protocol within the protocol being edited at the moment. The user can select among the previously defined protocols, which will appear as static text on the box. It's also possible to append a free text description to the node.

- **Task:** of all the types of nodes this is perhaps the most relevant when building a protocol. On a Task node, the user may append one or more tasks, of a certain task type. The available task types include:
    - Analysis
    - Appointments
    - Image Exams
    - Procedures
    - Nurse Teachings
    - Nurse interventions
    - Other Exams
    - Local medication
    - External medication
    - Opinion

  The selected tasks descriptions will be shown as static text on the box, and some free text description can be appended.

ProtocolElement

Each ProtocolElement component provides an object that may be dragged into the Stage. This object has an associated visual representation and a name. Figure 46 shows the ProtocolElements that are available when building a protocol. The parameters for these components and the dispatched events are listed on Table 14 and Table 15, respectively.

Figure 46: ProtocolElement components.

Table 14: ProtocolElement parameters.

Parameter	Type	Description	Default value
iconColor	Number	The color of the icon to display.	`null`
iconName	Text	The name of the icon MovieClip.	`<empty string>`
iconWidth	Number	The width in pixels of the icon.	`64`
text	Text	The text to display.	`<empty string>`

Table 15: ProtocolElement events.

Event name	Description
`elementReleased`	Dispatched the object is released, after it has been dragged.

Local structures

Typically, client side data of Alert® (the user computer) is not persistent. This means that when a user navigates through different screens in Alert®, the data that is required for a given screen is requested to the database every time that screen is loaded, and saved to database when exiting the screen. When editing a protocol diagram, and after selecting to attach a task, the user is forwarded to the task selection screen, returning afterwards to the diagram editing screen. This would mean having to load the diagram data twice just for this simple operation. A different approach has been adopted for the protocols.

When the user chooses to edit a protocol, the corresponding diagram and tasks' details are loaded from database. This data is mapped into an internal structure that will be updated as the user inserts, removes or edits elements, or any time a task is added. This avoids having to set all the data associated with the protocol every time any change is made. After the user finishes editing, and confirms the intent to save the changes made, the modified structure is then "uploaded", replacing the previous one (Figure 47).

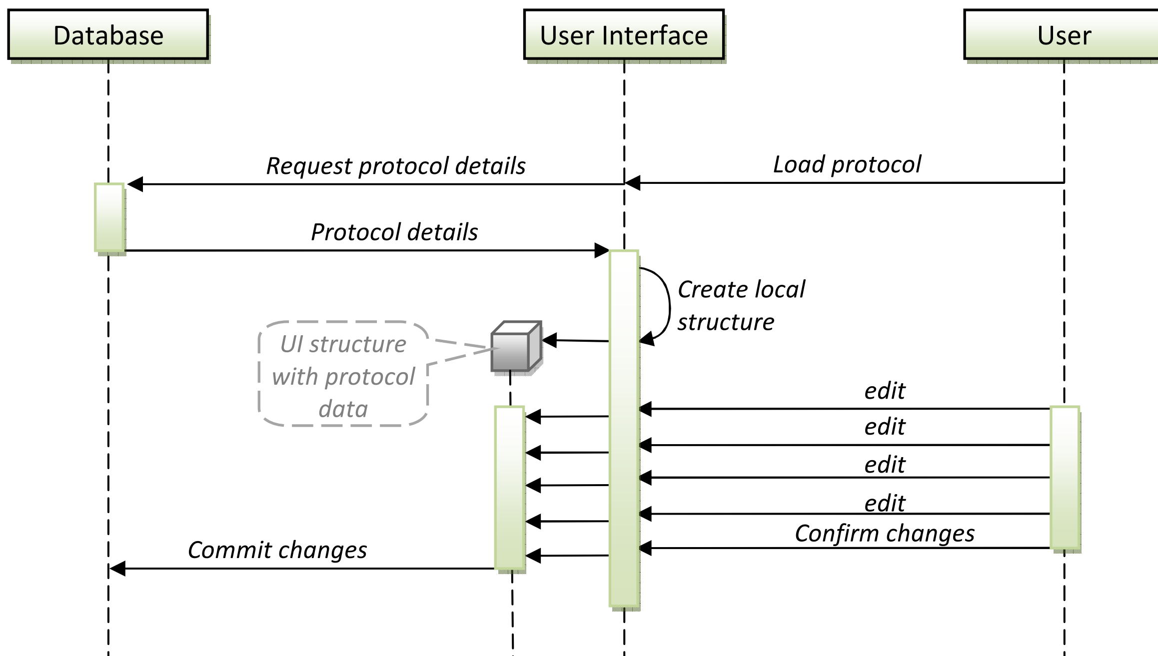

Figure 47: Sequence diagram illustrating use of local protocol data across screens.

This approach saves time and network traffic, since it eliminates the need for some (often redundant) instructions to exchange data with the server. Additionally, it enables the "help-save" feature as described next.

“Help-save” feature

The fact that all the data related with a protocol is always known to the interface while it’s being edited (and not just within the flowchart screen) allowed the implementation of a save confirmation dialog when a user exits abnormally.

If a user selects a different application area while editing a protocol, he is prompted to commit or discard his changes. While he is not able to go back and continue editing from where he left, due to limitations on Alert®’s core framework, he has an option to save the changes made up to that point. This is only possible if all mandatory fields were set and all requirements to save are verified, otherwise changes will be discarded. This workflow is illustrated on Figure 48.

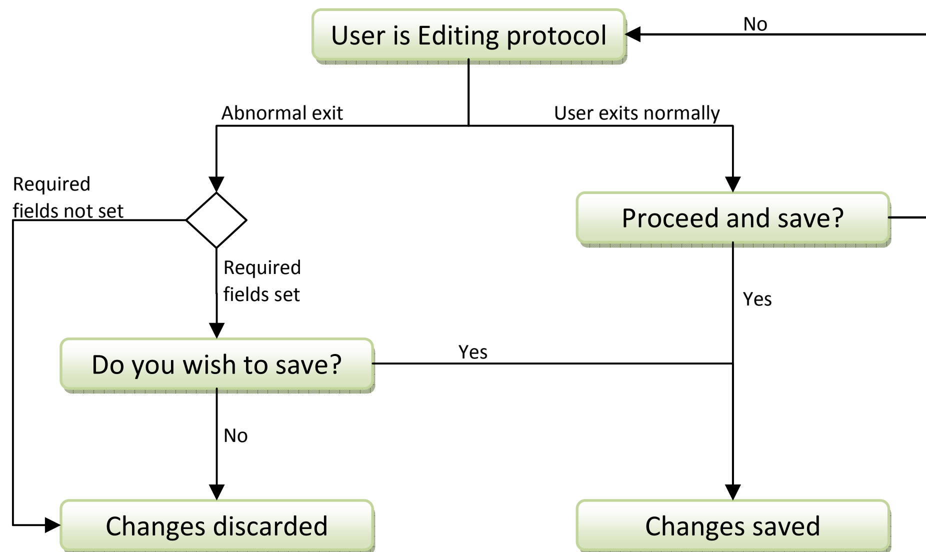

Figure 48: Protocol saving workflow.

4.5.1. Protocol Front Office development

Protocols are accessible within the **patient clinical record**, under the **Clinical Decision** application area. Selecting the **Protocols DeepNav**, all the protocols previously assigned to the patient, or automatically recommended are listed. If the patient has no assigned or recommended protocols, the advanced search screen is displayed instead, where the user can select from existing protocols.

Selecting a protocol, its associated diagram is displayed, and focus is set on the element immediately after the title box, where the protocol flow starts. When executing a protocol, there is always an active element at any given moment. There is a set of actions that may be performed according to the active element type (Figure 49):

- **Question:** The options displayed correspond to the connectors originating from this question. The next active element is decided according to the selected option.
- **Task:** The user can *IGNORE* the task and proceed, or choose to *PERFORM* the task, in which case he will be forwarded to the appropriate requisition screen. This will generate workflow to the responsible professionals. Performing a task may not be possible if the user does not have enough privileges or if the task is not available within that particular software or institution.
- **Protocol:** The user can *IGNORE* the protocol and proceed or *PERFORM* the protocol. In the latter case, he will be forwarded to the diagram of the "nested" protocol, where it can be followed.
- **Instruction:** The user may select *IGNORE* or *EXECUTE* the instruction, proceeding in both cases to the next element. None of these options will generate workflow, since an instruction is simply a textual indication of some task(s) that cannot be indicated in a Task element.
- **Warning:** The user may confirm acknowledgement of the warning, selecting *READ*.

Figure 49: Applying a protocol to a patient.

Although the protocol flow should be followed according to the pathways defined by the connectors' arrow, the user has the option to override this flow by activating any box element on the protocol at any given time.

4.6. Testing and debugging

Although no specific test framework has been adopted during development, all code produced and adapted was thoroughly tested to ensure that no user action can lead to undesired consequences. These tests included (23):

4.6.1. Black box tests[15]

When functions are created or modified for the user interface, or new classes are implemented, the affected functionalities are "black box tested". Correct and erroneous input values are passed to these components, and the output produced is then analyzed to trace errors or misbehavior of the components. When no visual output is produced by these objects, they are forced to output trace messages that show if the result in the end is the one expected. For viewing these runtime messages, the Flash console is used when possible. When testing on Alert® environment, the XRay tool (see section 0) was used to capture outputs.

It's also very common to invoke a database function and receive back some output values which can be evaluated and compared to the expected results. This testing that is performed on the database from the user interface point of view is also a good example of black box testing.

4.6.2. White box tests[16]

For white box tests, XRay has been used to explore flash objects and their parameters. XRay provides the state of all objects on screen in real-time, providing an internal view of the objects and thus allowing testing for abnormal conditions.

[15] **Black box testing** takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure. (25)

[16] **White box testing** (a.k.a. clear box testing, glass box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate output. (25)

5. Conclusions and Future Work

The final result is extremely positive and the developed tool is in line to what was proposed as the project objectives. The analysis of existing technologies on this area has provided valuable information that was used to define a pathway for development of Alert®'s protocol tool.

Section 5.1 provides a comparison between the developed solution and the technologies exposed on the State Of The Art section. On section 5.2 the project highlights are summarized, as well as some setbacks and debatable issues. Section 5.3 presents some aspects that should be addressed in future developments for this solution.

5.1. Results

Comparing the developed tool with the most relevant technologies in the field, there has been a somewhat different approach towards the problem. This is mostly due to the context for which the programs were developed, their purpose and their priorities.

Table 16 presents a comparison of the technologies chosen to implement the different tools.

Table 16: Technologies comparison.

	Alert® protocols	Asbru		PROforma	
		+ AsbruView	+ CareVis	Arezzo	Tallis
Interface	Flash	Java	Java	Java	Java
Storage	Oracle database	XML	XML	Internal storage format	Internal storage format
Plan Focus	Task Oriented	Time Oriented	Time Oriented	Task Oriented	Task Oriented
Graphical representation	Flowchart	Timeline + Topological	Timeline + Flowchart	Flowchart	Tree + Flowchart
Platform	Cross-platform	Cross-platform	Cross-platform	Windows	Cross-platform

As previously explained, while the focus of both *Asbru* and *PROforma* based software is interoperability and standardization of a guideline representation language, Alert® protocols are intended to integrate fully into Alert®. This is reflected on the chosen technologies. Even though Alert® is cross-platform (and consequently Alert® protocols as well), it relies on Flash and Oracle, both commercial and proprietary technologies. The other tools rely on Java for graphical interface implementation and *Asbru* uses XML for data storage, which eases interoperability.

Alert® protocols are task-oriented, meaning their main focus is on tasks themselves, their requisitions and generated workflows, and not as much on time constraints or temporal factors, even though it's possible to define scheduling and time parameters on protocols' tasks.

Regarding graphical representation of protocols, flowcharts were given priority over other methods such as timelines or tree views. This type of representation was considered more intuitive and essential for most protocols, and thus was the only implemented view. Other visualization methods are not excluded however, and may be considered for later releases.

5.2. Project overview and highlights

Even though the project was focused on the development of a graphical interface, there has been a significant research component. It also demanded an effort to innovate on the way components are built into Alert®, and to maintain them appropriately organized, structured and documented, which contributed to develop some good working and programming practices.

5.2.1. Research

When a new feature aims to convert between different formats, in this case from *paper-based schematics* to a *computer representation*, it's essential to have a profound knowledge of both. Furthermore, the type of content that makes up the source format – paper based protocols – must be taken into account, so that the developed feature may fully respond to the user needs.

This work called for a close contact with the Content department on a first stage to clarify what was intended with this tool (as exposed on section 4.1), and later to validate that the developed tool corresponded to users' necessities.

5.2.2. Innovative Challenges

The developed protocols module is in its essence an innovation in Alert®, especially concerning user interface, as explained before. It demanded new methods of interaction for the user that represented a challenge to user interface development. This added extra value to the project, since there was some space to think *out of the box* and not to just having activities focused on creating yet another set of Alert® screens.

5.2.3. Component reuse

The developed interface components were built with reusability in mind. Even though they were created as an infrastructure to protocols, they were designed in a way that allows their use for other applications. The flow chart components could be used to represent workflows of some sort, or individual components could be given completely different uses. The Connector object for example, may be used to visually interconnect two interface objects that relate to each other in some way (e.g. associate a medication on a grid to a problem on another grid).

This focus on reusability motivated the development of robust, yet flexible modules, emphasizing programming and documentation best practices.

5.2.4. Setbacks

Being a relatively unexplored field, protocol automation is a task prone to cause uncertainties and setbacks. This work placed some pertinent questions, of both ethical and technical nature, some with answers that are not consensual.

What can be automated and how?

It's not easy to determine an ideal method to define and store protocol information. Storing information in free text maximizes flexibility but on the other hand reduces interoperability and is error prone, degenerating data quality considerably. Using predefined discrete sets of contents improves usability, standardization, integration and interoperability, but also reduces input flexibility for the user and if contents are not carefully chosen may narrow the amount of information that can be introduced.

A solution that has been implemented on the developed tool was to combine both information originating from medical standards (disease, medication and treatment nomenclatures, among others) and information freely entered by the user. This is meant to combine the best of both worlds, but still doesn't account for all possible situations (e.g. one can't prescribe something that was entered manually by the user, since this information can't be interpreted by the application)

Where to stop automation?

This is a question that divides opinions. Assuming that the required patient data is known to the application, and that treatment tasks can be requested autonomously, should it still be done without human interference? How much trust should be put in a machine?

While perhaps most people will agree that a clinical professional must **at least** validate requests before they are made by the system, opinions diverge on whether the system should even suggest a treatment automatically (possibly misleading the user instead of aiding).

In Alert®, it has been decided not to automate task requests (even though it would be possible), but to present them as a suggestion to the user, requiring an explicit action from the professional confirming the request. Protocols are automatically suggested to patients that fit within the criteria, but merely as suggestions, and this is stressed out to the end users when presenting the product.

How to guarantee credible and current information?

This is a pertinent question nowadays, where digital content is shared worldwide often with little, uncertain or no information about its origins. While at a smaller scale, this problem can be found within Alert®. Given that a large number of people can freely create protocols for other people to use, information credibility must be questioned. Information on who created and edited a protocol is available to the user, but there is no reliable method of guaranteeing protocols contents' validity. A possible way to improve this issue would be to centralize protocol validation on a restrict set of people with the required knowledge.

How to "navigate" a Protocol?

There are several concerns to take into account regarding the way a user is directed through a protocol.

- The next step should be made available after the previous has completed, or after it was requested, independently of whether it was actually performed?
- Should it be possible to ignore a given step of a protocol?
- Should the user be allowed to "override" the natural sequence of the protocol and select/perform previous or subsequent steps? What should happen if a step is performed multiple times, or if it's performed but later on canceled?

There is no ideal solution for these issues; they're decisions that must be taken according to what brings the most benefits and flexibility to the user, without compromising usability and safety. With this in mind, and since a protocol is viewed as a suggested pathway, Alert® allows the users to override the natural sequence of a protocol, enabling the user to ignore steps and to "jump" to other out of context steps. The protocol flows immediately after a task is requested, whether or not it has been completed.

Interoperability vs completeness?

As previously mentioned, there is usually a compromise between the differentiating features supported by an application and its interoperability capabilities. As previously explained, developments were focused on creating an integrated and complete tool and interoperability with other systems was given a lower priority.

5.3. Future work

This work has filled an existing gap on Alert® products, and although it's usable and robust as it is, there is much space for future improvements, which will surely increase as feedback from costumers becomes available. Features that are planned as future work at the moment include

Improve connector routing

Connectors currently use a somewhat primitive method for directing themselves. Their position is decided independently of other components' positions, often becoming overlaid or confusing. In most cases this can be solved by repositioning the Boxes as appropriate, but a possible future improvement is shown on Figure 50, where connectors avoid crossing with boxes and "jump" over other Connectors by displaying a slight curvature.

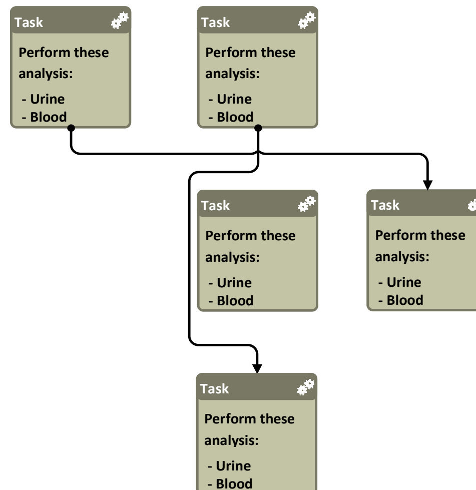

Figure 50: Connector routing improvements

Allow multiple task types on same element

Because each type of tasks has its own parameters, and since they must be set before any request, it's impossible to request tasks of different types at the same time. For this reason, the user can't place tasks of different types on a same Box element. This is definitely a worthwhile improvement that could be accomplished by setting default values for the tasks' parameters when creating the protocol and/or presenting a summary screen where the user can set parameters for each selected task, when executing the protocol.

Automatic path suggestion based on patient data

As previously mentioned, automation may not be desired and/or recommended. Suggestion of paths to follow might be done automatically by the system if the required patient information is known. This may be a possible future improvement.

Other protocol views (timeline, tree view, text view)

The developed tool provides a flow chart view of the protocol diagram, which was considered the most meaningful. Some of the analyzed protocol software programs provide different perspectives on a protocol flow, such as timeline views, tree views and even textual versions of the diagram. Although superfluous, these alternative perspectives would bring value and flexibility to the functionality.

Study alternative applications for the developed tools

The flowchart components implementation was done in a way that will allow their use within other contexts. They may be adopted in the future e.g. to build screen workflows, to create relations between interface elements, or even to document functionalities.

Also, it's worth considering the commercialization of the protocols tool as a standalone feature (an independent Alert® product), given that this feature can be made completely independent from the rest of the applications.

This work contributed to take some weight off of clinical professionals, by using technology to accurately store and share knowledge, and leaving professionals responsible for judging it.

> *"It's ridiculous to live 100 years and only be able to remember 30 million bytes. You know less than a compact disc. The human condition is really becoming more obsolete every minute."* (Marvin Minsky[17])

[17] Professor and researcher at MIT Media Lab and MIT AI Lab

References

1. **Coiera, Enrico.** *Guide to Health Informatics.* s.l. : Hodder Arnold, 2003.

2. **Rodrigues, Hugo.** *Decision Support Systems in Alert® – Integrating Guidelines and Bibliography.* Porto : Edições FEUP, 2007.

3. **Wikipedia.** Evidence-Based Medicine. *Wikipedia.* [Online] 02 09, 2008. [Cited: 02 21, 2008.] http://en.wikipedia.org/wiki/Evidence-based_medicine.

4. **DL, Sackett, et al.** *Evidence based medicine: what it is and what it isn't: It's about integrating individual clinical expertise and the best external evidence.* 1996.

5. **School of Rural Health, University of Sydney.** What is Evidence Basesd Medicine. *Medical Informatics@SRHDubbo.* [Online] [Cited: 02 28, 2008.] http://srhdb.com/ebm_whatis.htm.

6. **Stolba, Nevena and Tjoa, A Min.** *An Approach Towards the Fulfilment of Security Requirements for Decision Support Systems in the Field of Evidence Based Healthcare.* 2006.

7. **Oracle.** Oracle Homepage. [Online] http://www.oracle.com/.

8. —. Oracle Forms. [Online] [Cited: 04 01, 2008.] http://www.oracle.com/technology/products/forms/index.html.

9. **Microsoft Corporation.** WindowsClient.net. [Online] http://windowsclient.net/.

10. **Java.** Applets. [Online] [Cited: 04 01, 2008.] http://java.sun.com/applets/.

11. **Wikipedia.** Ajax (programming). [Online] [Cited: 04 01, 2008.] http://en.wikipedia.org/wiki/Ajax_(programming).

12. **Netscape.** ECMAScript 4 Netscape Proposal. [Online] 06 30, 2003. [Cited: 04 01, 2008.] www.mozilla.org/js/language/es4.

13. **Wikipedia.** Java. *Wikipedia.* [Online] [Cited: 04 01, 2008.] http://en.wikipedia.org/wiki/Java_(programming_language).

14. **Peleg, Mor, et al.** *Comparing Computer-Interpretable Guideline Models: A Case-Study Approach.* Stanford, : Stanford Medical Informatics, 2003.

15. **Aigner, Wolfgang and Miksch, Silvia.** *Communicating the Logic of a Treatment Plan Formulated in Asbru to Domain Experts.* Vienna : s.n., 2004.

16. **Tu, S.W. and Musen, M.A.** *A flexible approach to guideline modeling.* 1999.

17. **Peleg, M., Boxwala, A. and Ogunyemi, O.** *GLIF3: The Evolution of a Guideline Representation Format.* 2000.

18. **Quaglini, S., et al.** *Flexible Guideline-based Patient Careflow Systems.* 2001.

19. **Johnson, P.D., et al.** *Using Scenarios in Chronic Disease Management Guidelines for Primary Care.* 2000.

20. Tallis Training. *COSSAC - IRC in Cognitive Science and Systems Engineering.* [Online] 9 28, 2007. [Cited: 02 22, 2008.] http://www.cossac.org/tallis/.

21. **Faculty of Informatics of the Vienna University of Technology.** AsbruView. *The ASGAARD Project :: Plan Authoring.* [Online] 03 10, 2006. [Cited: 02 21, 2008.] http://www.asgaard.tuwien.ac.at/asbruview/index.html.

22. **Aigne, Wolfgang and Miksch, Silvia.** The CareVis Project. *Information Engineering Group - Vienna University of Technology.* [Online] 04 27, 2005. [Cited: 02 22, 2008.] http://ieg.ifs.tuwien.ac.at/projects/carevis/.

23. **Wikipedia.** Software Testing. [Online] [Cited: 03 18, 2008.] http://en.wikipedia.org/wiki/Software_testing.

24. —. Black Box Testing. [Online] 08 10, 2007. [Cited: 08 28, 2007.] http://en.wikipedia.org/wiki/Black_box_testing.

25. **Adobe.** *Flash 8 LiveDocs.* [Online] 3 30, 2007. [Cited: 12 05, 2007.] http://livedocs.adobe.com/flash/8/.

26. **Wikipedia.** White Box Testing. [Online] 07 10, 2007. [Cited: 04 01, 2008.] http://en.wikipedia.org/wiki/White_box_testing.

27. **Hans-Ulrich Prokosch, Joachim Dudeck.** *Hospital Information Systems: Design and Development Characteristics, Impact and Future Architecture.* s.l. : Elsevier, 1995.

28. **Kosara, Robert and Miksch, Silvia.** *Metaphors of Movement: A Visualization and User Interface for Time-Oriented, Skeletal Plans.* Vienna : s.n., 2001.

Glossary / Index of Terms

Alert LSC Alert Life Sciences Computing, the company where the internship took place.

AS ActionScript, or ActionScript file format, the object-oriented scripting language based on ECMAScript, used for Flash-based software development.

Care Plan See *Plan*.

Clinical Protocol A Clinical Protocol, or just Protocol, is a streamlined plan for carrying out a scientific study or a patient's treatment regimen. In Alert®, Clinical Protocols are looked at as Guidelines with a decision tree associated. In clinical contexts, the term Protocol is often used when referring to Guidelines or Care Plans.

Datagrid A Flash component used to display information on screen organized in a scrollable and customizable grid.

DSS Decision Support System(s). Frameworks that aid medical professionals in diagnosing and general decision making.

EBM Evidence Based Medicine, an attempt to apply the evidence gained from the scientific method to certain aspects of medical practice. Assess the quality of evidence relevant to the risks and benefits of treatments (including lack of treatment). According to the Centre for Evidence-Based Medicine, "Evidence-based medicine is the conscientious, explicit and judicious use of current best evidence in making decisions about the care of individual patients".

FLA Flash file, contains source material for the Flash application. Flash authoring software can edit FLA files and compile them into .SWF files. Proprietary to Adobe.

Guideline A Guideline or Clinical Practice Guideline is a systematically developed statement used to assist practitioners and patients in making decisions about appropriate health care specific clinical circumstances. Has a global perspective that must be interpreted at local level. In clinical contexts, the term Guideline is often used when referring to Protocols or Care Plans.

HL7 Health Level Seven, is an organization involved in development of international healthcare standards. "HL7" is also used to refer to some of the standards created by the organization (i.e. HL7 RIM etc.).

IT Information Technology, the study, design, development, implementation, support or management of computer-based information systems, particularly software applications and computer hardware.

OOP Object Oriented Programming. A programming paradigm that uses "objects" to design applications and computer programs. Uses techniques like inheritance, modularity, polymorphism, and encapsulation.

PHR Personal Health Record. Provides the health and medical history of an individual.

PL/SQL Procedural Language/Structured Query Language. Oracle Corporation's proprietary server-based procedural extension to the SQL database language.

Plan A Plan or Care Plan is a “roadmap” to guide those involved with a patient’s care. The term is often used when referring to clinical Protocols or Guidelines.

PM Product Manager. The person who deals with the product planning at all stages of the product lifecycle.

Protocol See *Clinical Protocol*.

SNOMED CT Systematized Nomenclature of Medicine Clinical Terms, is a computerized clinical terminology covering clinical data for diseases, clinical findings, and procedures developed by the College of American Pathologists.

SQL Structured Query Language. A computer language designed for the retrieval and management of data in relational database management systems.

SVN Subversion, a version control system “inspired” on CVS (Concurrent Versions System).

SWF *Small Web Format* file (also commonly called *Swiff* file), is the compressed and uneditable file produced by Flash after compiling a **.fla** file.

UI User Interface