Guideline Interchange Format 3.5
Technical Specification

Draft Release

December 12th, 2002

InterMed Collaboratory

Document Editors:

Mor Peleg, Aziz Boxwala, Samson Tu, Dongwen Wang, Omolola Ogunyemi, Qing Zeng

Significant contributions to the model and the document were made by (in alphabetical order):

Nachman Ash, Elmer Bernstam, Robert A. Greenes, Ronilda Lacson, Peter Mork, Edward H. Shortliffe

1. Introduction

1.1 *Purpose of document*

1.2 *What is GLIF?*

GLIF3 is a methodology that enables modeling and representation of clinical guidelines in a structured manner. Such guidelines can be used for clinical decision support applications.

Guidelines are modeled in GLIF at three levels of abstraction: a conceptual flowchart that is easy to author and comprehend, a computable specification that can be verified for logical consistency and completeness, and an implementable specification that can be incorporated into particular institutional information systems.

The GLIF3 model is object-oriented. It consists of classes, their attributes, and the relationships among the classes, which are necessary to model clinical guidelines. The model is described using Unified Modeling Language (UML) class diagrams [1]. Additional constraints on represented concepts are being specified in the Object Constraint Language (OCL), a part of the UML standard.

A top-level view of the GLIF model is shown in Figure 1. The complete class hierarchy is shown in Appendix C.

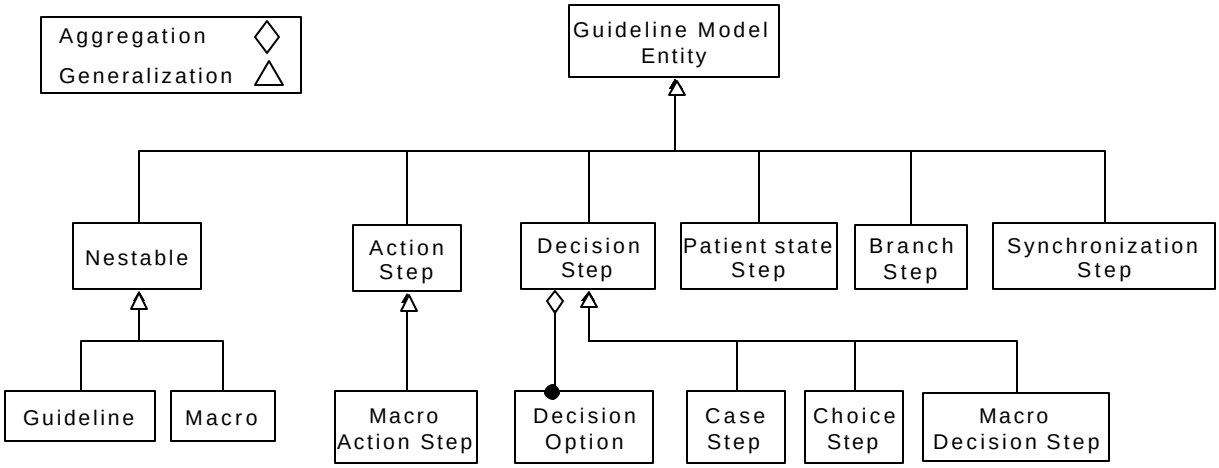


Figure 1. The GLIF model. A top-level view of the main GLIF classes

2. Overview of GLIF

2.1 *Scope of GLIF*

GLIF is intended to be used in a variety of guidelines. Guidelines may be classified according to the clinical domain, the stage of the medical problem and its management (e.g., screening, diagnosis, disease management), multiple or single encounters, setting (e.g., inpatient or outpatient clinic), time frame (emergency, acute, or chronic), and guideline computability (i.e., algorithmic, guiding, or intermediate) [[2]]. We have used GLIF to encode a variety of guidelines

including guidelines for Influenza vaccination [[3]], management of chronic cough [[4]], management of stable angina [[5]], thyroid screening [[6]], lower back pain [[7]], heart failure [[8]], and depression [[9]], as shown in Table 1 [10].

Table 1 – Classification of GLIF3-encoded Guidelines

Disease/ Condition	Stage of Problem	Encounters	Setting	Time Frame	Computability
Flu	prevention	1	outpatient	acute	algorithmic
Stable Angina	management	many	outpatient	acute/chronic	intermediate
Chronic cough	Diagnosis + management	many	outpatient	acute	intermediate
Lower back pain	Diagnosis + management	many	outpatient	acute	intermediate
Heart failure	management	many	outpatient	acute/chronic	algorithmic
Depression	management	many	outpatient	acute	algorithmic
Thyroid screening	screening	1	out	a	algorithmic

2.2 *Bird's eye view of GLIF*

In GLIF, guidelines are represented as a flowchart of temporally sequenced nodes called guideline steps. Different classes of guideline steps are used for modeling different constructs:

- The **Decision_Step** class represents decision points in the guideline. A hierarchy of decision classes provides the ability to represent different decision models.

- The **Action_Step** class is used for modeling actions to be performed. Action steps contain tasks. Two distinct types of tasks can be modeled: medically oriented actions such as a recommendation for a particular course of treatment, and programming-oriented actions such as retrieving data from an electronic patient record. Nesting of steps, discussed in Section 8, allows recursive specification of actions and decision. In other words, through nested steps, one can specify details of high-level actions and decisions as subguidelines.

- The **Branch_Step** and **Synchronization_Step** allow modeling of multiple simultaneous paths through the guideline.

- **Patient_State_Steps** serve as entry points into the guideline as well as allow for labeling patient states (e.g., a state of taking one anti-hypertensive drug).

The GLIF specification includes an expression language that is derived from the logical expression grammar of Arden Syntax. The expression language can be used for representing decision criteria, triggering events, exceptional conditions, duration expression, and states.

A medical ontology allows the use of standard controlled vocabularies and reference information models to specify linkages to individual patient data, medical knowledge, and clinical actions. This will facilitate sharing of guidelines among institutions.

The specification for supplemental material allows one to associate didactic material with the guideline itself or to some sections of it. The supplemental material can be in different formats (such as plain text and HTML) and can be for different purposes (such as rationale, further reading, patient education, etc).

2.3 *Layers of abstraction*

GLIF3 supports representing clinical guidelines in three levels of abstraction. When a guideline is first authored, a conceptual level of representation (called level A) is created. This level enables the guideline author to concentrate on conceptualizing a guideline as a flowchart. At this level of abstraction, the author is not concerned with formally specifying details, such as decision criteria, relevant patient data, and iteration information that must be provided to make the specification computable. The specification of these details is performed at the computable level of abstraction (level B), where medical concepts, patient data item definitions, logical criteria, and control flow are formally specified. We intend to create tools that will aid in the validation and simulation of guidelines that are specified at the computable level.

Before guidelines can be incorporated into an institutional information system, actions and patient data references must be mapped to institutional procedures and electronic medical records (EMRs). This mapping information is represented in the implementable level (level C). The implementable level has not yet been completed.

The different levels of abstraction are achieved by specifying values for different attributes of the GLIF classes. For example, a decision criterion (the Criterion class) has a *name* attribute that contains an English sentence that describes the criterion in free text (*e.g.*, high LDL cholesterol) and also a *specification* attribute that contains a formal definition of the criterion using the GEL expression language

(e.g., selectAttribute(“pq_value”, selectAttribute(“value”, Current_LDL_Cholesterol)) >= 160 and selectAttribute(“unit”, selectAttribute(“value”, Current_LDL_Cholesterol)) ==mg/dL)

The name attribute is specified at level A, while the specification attribute is specified at level B. When a domain expert encodes the guideline, he can first specify the level A attributes. Later on, an informatician can specify the level B attributes, after consulting with the domain expert. All GLIF classes have at least one level A attribute that lets the author describe the construct in unconstrained narrative text.

2.4 *Understanding GLIF3's medical ontology*

Logical expressions (criteria) and action specifications reference patient data items and medical concepts. These concepts are formally defined in the medical ontology, by referencing standard controlled vocabularies (e.g., UMLS [11]) and standard medical data models (e.g., HL-7's Reference Information Model version 1.0 [11]). Defining medical concepts in relationship to standard controlled medical vocabularies enables the guideline encoding in Level B to contain concepts that are not institution-dependent. The institution-dependent terms can therefore be specified only in level C, which will define the mappings between the Level B guideline terms and the institutional terms. Defining the structure of patient data items in accordance to standard medical data models is done to ease the process of mapping Level B guideline patient data items to institutional EMR codes and to facilitate the process of sharing guidelines. The support of the ontological needs for guideline modeling is separated into three layers, correlated to levels of abstraction. The first layer, **Core GLIF**, is part of the GLIF specification language. It defines a standard interface to medical data items and concepts, and to the relationships among them.

The second layer, **Reference Information Model (RIM)**, is essential for guideline execution and data sharing among different applications and different institutions. It defines the basic data model for representing medical information needed in specifying protocols and guidelines. It includes high-level classification concepts, such as medications and observations about a patient, and attributes, such as units of a measurement and dosage for a drug, that medical concepts and medical data may have. The default RIM that GLIF3 supports is HL-7's Reference Information Model (RIM) version 1, also known as the Unified Service Action Model (USAM) [12].

The third layer, **Medical Knowledge Layer** is still under development. It will be specified in terms of the methods that it should have for interfacing to the following medical knowledge sources:

1. Controlled vocabularies, like UMLS, that define medical concepts by giving them textual definitions and unique identifiers.
2. Medical knowledge bases that define medical knowledge, such as drug hierarchies, and normal ranges for test results.
3. Clinical repositories (EMRs)
4. Other clinical applications, such as order entry systems, alert/reminder systems.

When all three layers are involved, they work closely together: Core GLIF relies on the RIM to supply the attributes of the medical concepts and to represent data values. Core GLIF relies on the Medical Knowledge Layer for accessing specific medical concepts.

In the three-layered medical ontology, users have the freedom to choose a particular RIM and a particular medical knowledge layer that fits their needs. We encourage guideline authors to use a single RIM and a single controlled vocabulary to encode one guideline. This eases the process of sharing the guideline, since mapping terms that belong to different RIMs and vocabularies is a difficult task.

2.4.1 Core GLIF

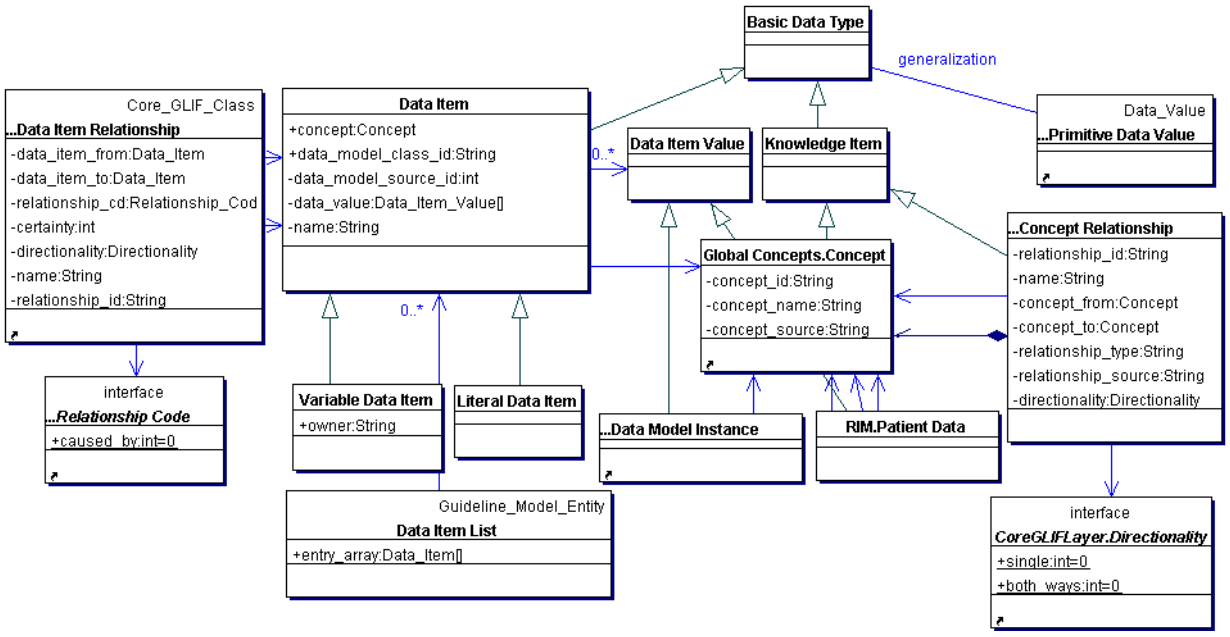


Figure 2. The Data model package, which is part of Core GLIF

Core GLIF defines the medical data model of GLIF3, which is, how medical data items are structured and how they relate to medical concepts. The specification of the Core GLIF layer is shown in Figure 2. GLIF3's **Basic_Data_Type** can be a **Primitive_Data_Item** (shown in Figure 6), **A Data_Item**, or a **Knowledge_Item**.

GLIF clinical decisions and actions refer to patient data items. Each patient **Data_Item** is defined by a medical concept, taken from some standard controlled vocabulary, and by a data model class and source. The data model class and source indicate the Reference Information Model (RIM) class and RIM model that is used for defining the data item's data structure. A data item also has a (complex) data value. A data value is a list of **Data_Item_Value**s. There are two types of data item values: the **Patient_Data** class of the default RIM, shown in Figure 9, or from a user-defined **Data_Model_Instance** class, shown in Figure 6, which is part of Core GLIF.

Core GLIF distinguishes between two types of patient data items: literals (constants) and variables. **Variable_Data_Item**s represent data that needs to be instantiated at run time from external sources (e.g., electronic medical record) when the guideline is being applied to a specific patient or to a group of patients. Patient's height, weight, gender, and age are examples of variable data items. A variable data item has an owner, to which the data value belongs. Specifying the owner of a data item is necessary because sometimes even in one guideline, data from multiple patients will be mentioned (e.g., phase-I clinical trial guidelines sometimes refer to a cohort of patients) although most of the guidelines are not applied to groups of patients. Variable data items are used when specifying decision criteria, as shown in Section 4.2.

A **Literal_Data_Item** is a data item that has a fixed value. It is similar to a constant in programming languages. Unlike a variable data item, a literal does not have an owner and its data value is modeled by a list of exactly one instance of Data_Item_Value. Congestive heart

failure, female, smoker, and TSH-test-order are all examples of literal data items. The values of literal data items are assigned at authoring time. When specifying action specifications, literal data items are used to specify a recommended action. An example is shown in Section 0. Literal data items are also used to specify literal values in decision criteria, as shown in Figure 42 of Section 4.2.

A **Data_Item_List** is a run-time object that allows referencing different data items in a single list. For example, all of the data items that are referred by a guideline are stored in a data item list (See Figure 16).

GLIF3 has two types of Knowledge_Items: Concept and Concept_Relationship. They are both an embedded part of GLIF, unlike the Reference Information Model that can be taken from various sources (e.g., HL7). A **Concept** is defined by defining its name, concept_id, the id of its source (concept_source), that is, what local vocabulary did the concept come from. Concept relationships are created using the **Concept_Relationship** class. An example is given in Figure 3.

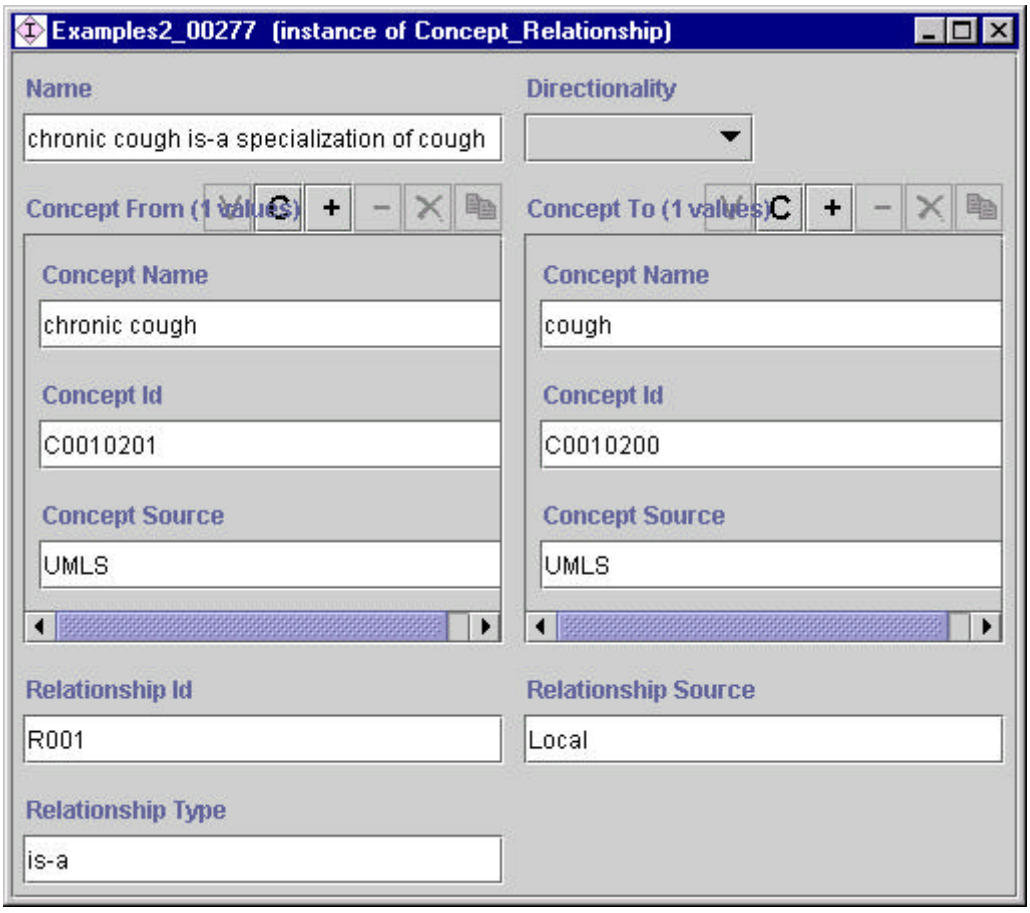


Figure 3. A concept hierarchy. The Concept_Ids are the UMLS codes for cough and chronic cough

A (patient) **Data_Item_Relationship** is a relationship between two data items (e.g., a certain patient's high body temperature was caused by a viral infection). Each data item relationship has

an associated code (e.g., "caused_by"), directionality, and a certainty attribute that expresses how sure we are that the relationship holds.

When users simply want to create a human-readable guideline, a RIM or controlled vocabulary might not be needed. When both the RIM and Medical Knowledge layers are absent, the concept, data_model_class_id, and data_model_source_id attributes of the data item are marked as "UNKNOWN". When a data item fails to be mapped to a concept, the referring concept is automatically assigned the value "UNKNOWN". When a data item does not have a data model specified by the RIM layer, the type for its data value is assigned to be a Data_Model_Instance whose *values* attribute is of the primitive type String_Value.

Sometimes, standard controlled vocabularies do not define a concept that the guideline needs to express. Similarly, a guideline may need to refer to data model classes that are not supported by the default RIM. In these cases, Core GLIF defines a way for the guideline author to define new concepts and hierarchies of concepts, new data model classes, and to map a concept to a data model class. The classes of Core GLIF that enable these functionalities are shown in Figure 4.

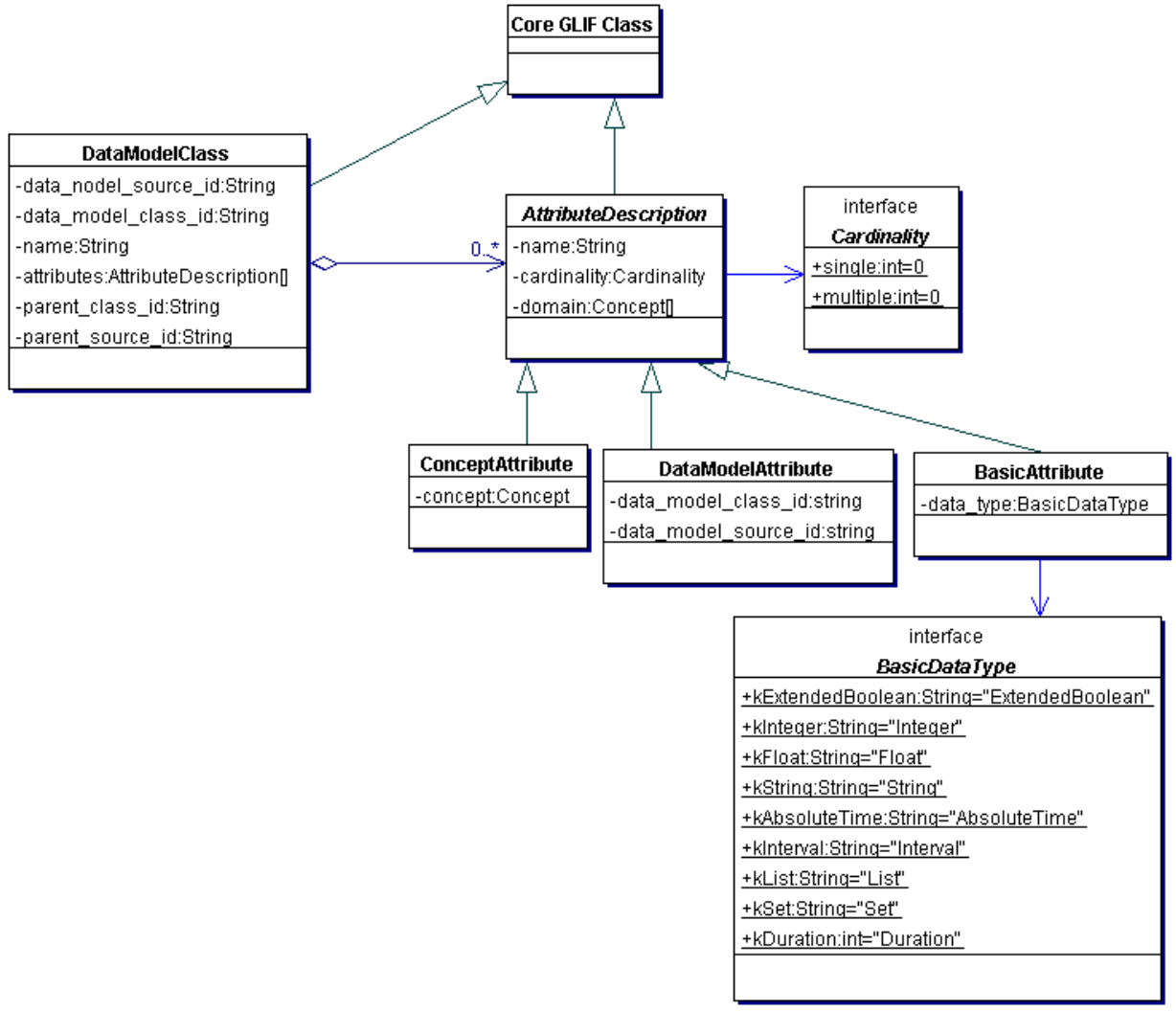


Figure 4. Core GLIF class diagram

A *Data_Model_Class* is an embedded part of GLIF, unlike the Reference Information Model that can be taken from various sources (e.g., HL7). A **Data_Model_Class** allows a user to define RIM classes that are not part of the default RIM. A data model class is defined by specifying its attributes and its parent data model class, thus supporting sub-classing. GLIF's Get_Data_Action (5.3.4) retrieves data from the EMR and presents it in a form of a Query_Result. A Query_Result assumes that each data value is associated with a primary timestamp. Therefore, on of the time-attributes of a *Data_Model_Class* will be defined in a Get_Data_Action as the primary time attribute.

The AttributeDescription class defines the attributes of a *Data_Model_Class* . Each attribute description object has a name and cardinality. There are three subclasses of **AttributeDescription.** They differ in the types of attributes that they define. The three sub-classes are:

1. **BasicAttribute**s, which are string, integer, float, Extended_Boolean, absolute_time, duration, interval, list, and set.
2. **ConceptAttribute**s, which define the range of concepts that are allowed to serve as the value of the attribute. For example, if the concept attribute refers to the concept "gender", then the allowed values for the attribute are male and female.
3. **DataModelAttribute**s, that are attributes whose type is another data model class.

When users define data model classes, they need to define the domain of each attribute of the data model class, that is, what concepts would use that data model class. Therefore, every AttributeDescription has a "domain" attribute that specifies the list of Concepts that define the domain of the attribute. (Note: This replaced GLIF 3.1's DataModelConcept_Map)

Figure 5 shows an example of a data model class.

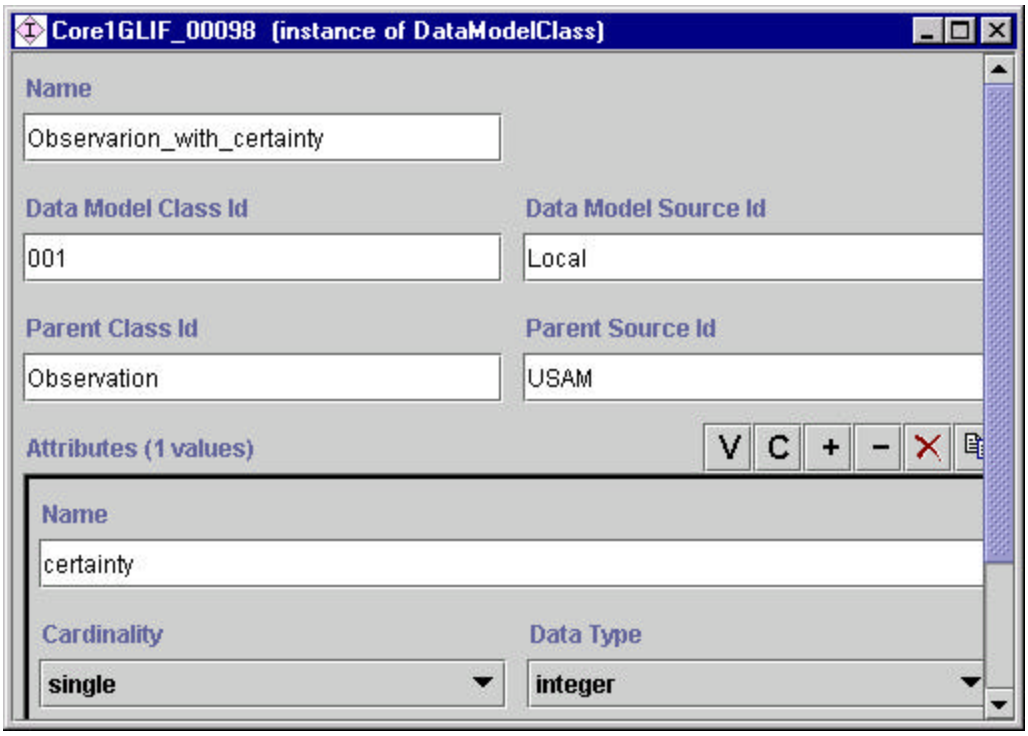


Figure 5. An example of a data model class with a basic attribute. Observation_with_certainty is derived from USAM's Observation class and extends it by adding the simple attribute "certainty", of type Integer.

Instances of user-defined (where the user is the guideline author) DataModelClass are defined using the **Data_Model_Instance** shown in Figure 6. Examples are shown in Figure 7 and Figure 8.

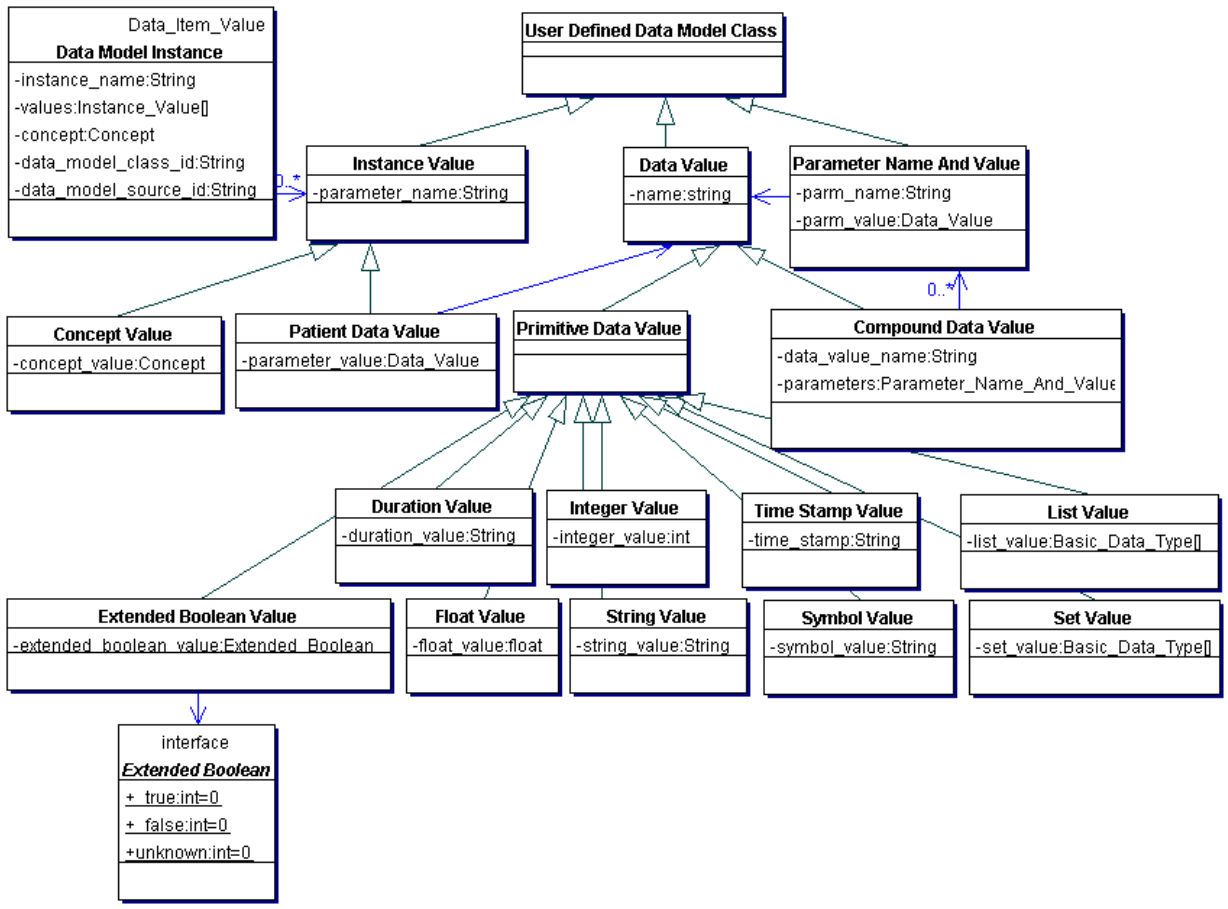


Figure 6. User Defined Instances Ontology

A **Data_Model_Instance** identifies the concept and the data_model class and source that it refers to. It also has a "values" attribute, which is of a list of Instance_Value objects.

An **Instance_Value** can be either a **Concept_Value** or a **Patient_Data_Value**. Concept values refer to concepts, as in the example shown in Figure 8. Patient data values have **Data_Values**, which can be either **Primitive_Data_Values** (i.e., **integer**, **float**, **string**, **time-stamp**, **duration**, **extended Boolean**, **symbol**, **list**, or **set**, as shown in Figure 6), or compound. **Compound_Data_Values** have parameters, of type **Parameter_Name_And_Type**, which refer to other data values.

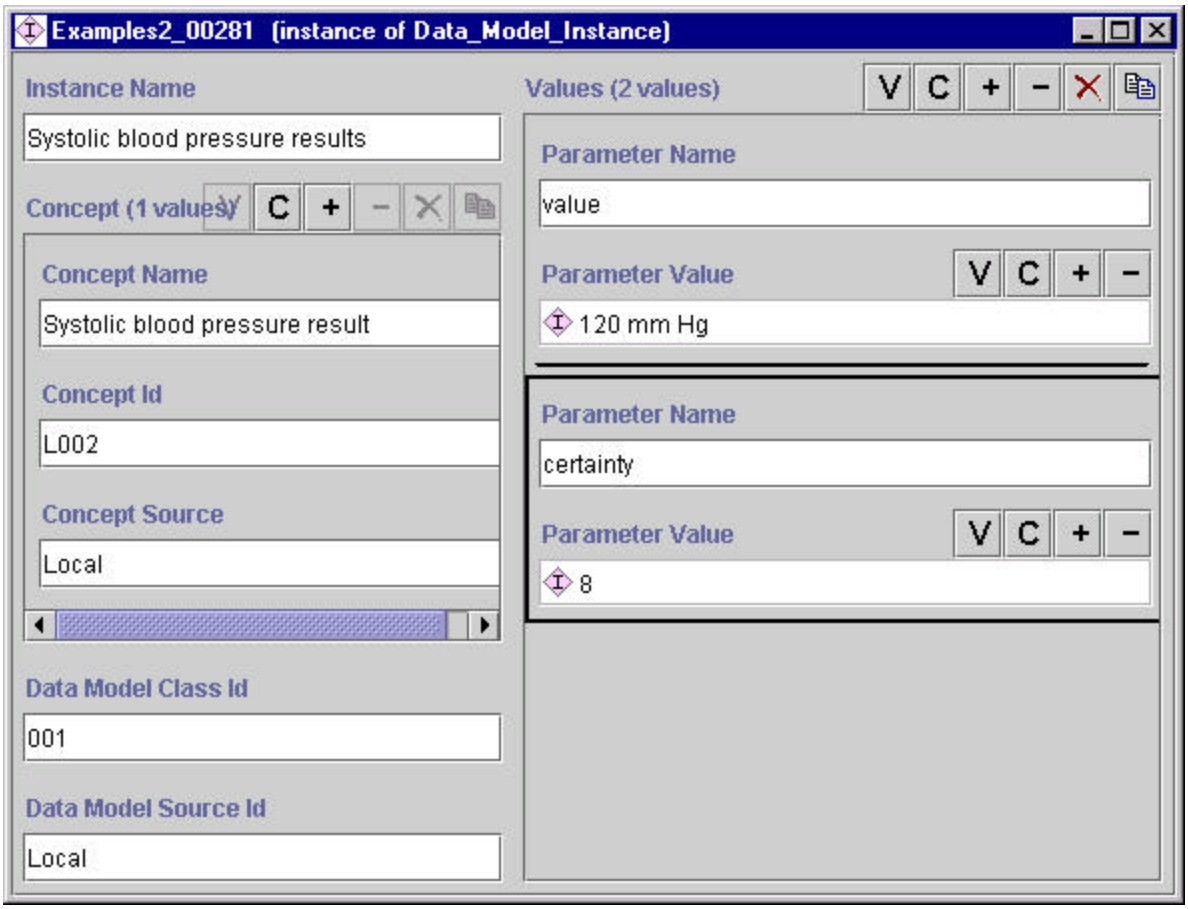


Figure 7. An instance of systolic blood pressure result, which corresponds to the locally defined Observation_with_certainty class that is shown in Figure 5.

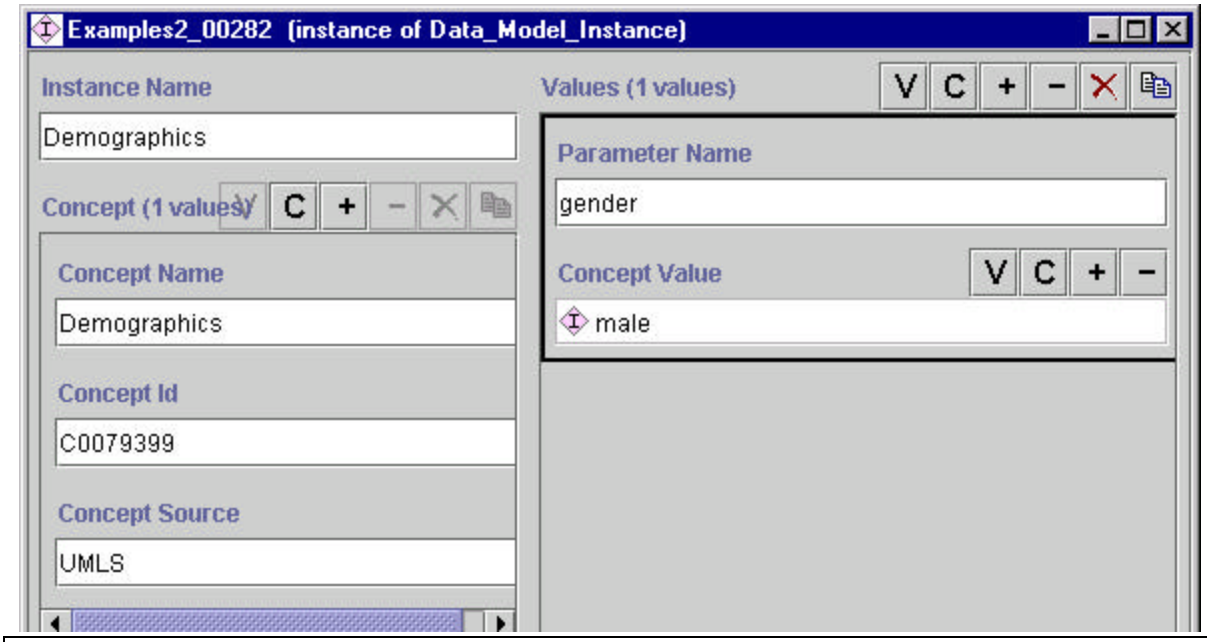


Figure 8. An instance of Demographics, which has a Concept_Value of the concept "gender"

2.4.2 Reference Information Model (RIM)

A RIM defines a class hierarchy that organizes medical concepts into classes. For each class, the RIM provides a data model that defines the attributes of the different classes. We examined several reference information models, including HL-7's RIM version 0.94, the clinical part of HL-7's RIM version 1.0 (also known as the Unified Service Action Model (USAM), foundational models in SNOMED-RT, and the National Library of Medicine's Semantic Net, which is part of UMLS.

We chose to use HL-7s Unified Service Action Model as our default reference information model, because of its generality. We use USAM's Service_Action class for representing patient data. We therefore renamed the Service_Action class as Patient_Data. We are not utilizing all of the classes and attributes that are defined in USAM, since we have a different approach for modeling them in GLIF. Specifically, we are only using the Service_Action (Patient_Data), Medication, Observation, and Procedure classes. The following changes were also made:

1. We are not representing the following attributes of the service (patient data) class: max_repeat_number, interruptible_ind, substitution_cd, priority_cd, and orderable_indication.

2. We added two attributes to the Observation class: severity, and certainty that we found lacking from USAM's Service_Action class.

3. We simplified some of the attribute data types used by USAM. For example, for activity_time and critical_time, we only allow time intervals, and not every temporal function.

The USAM reference information model class diagram is shown in Figure 9. Figure 10 shows the enumerated codes used by the USAM reference information model.

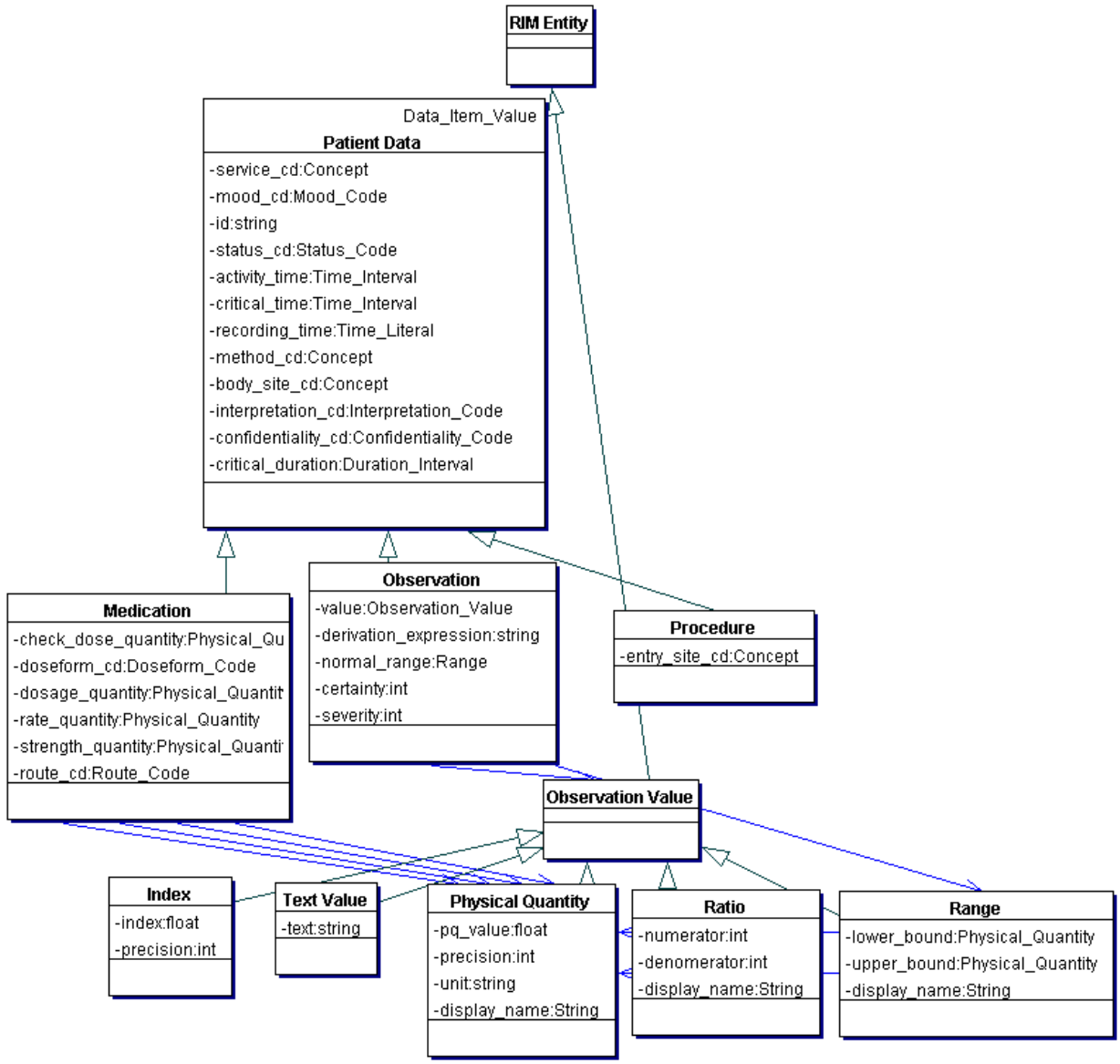


Figure 9. HL-7's USAM-RIM class diagram

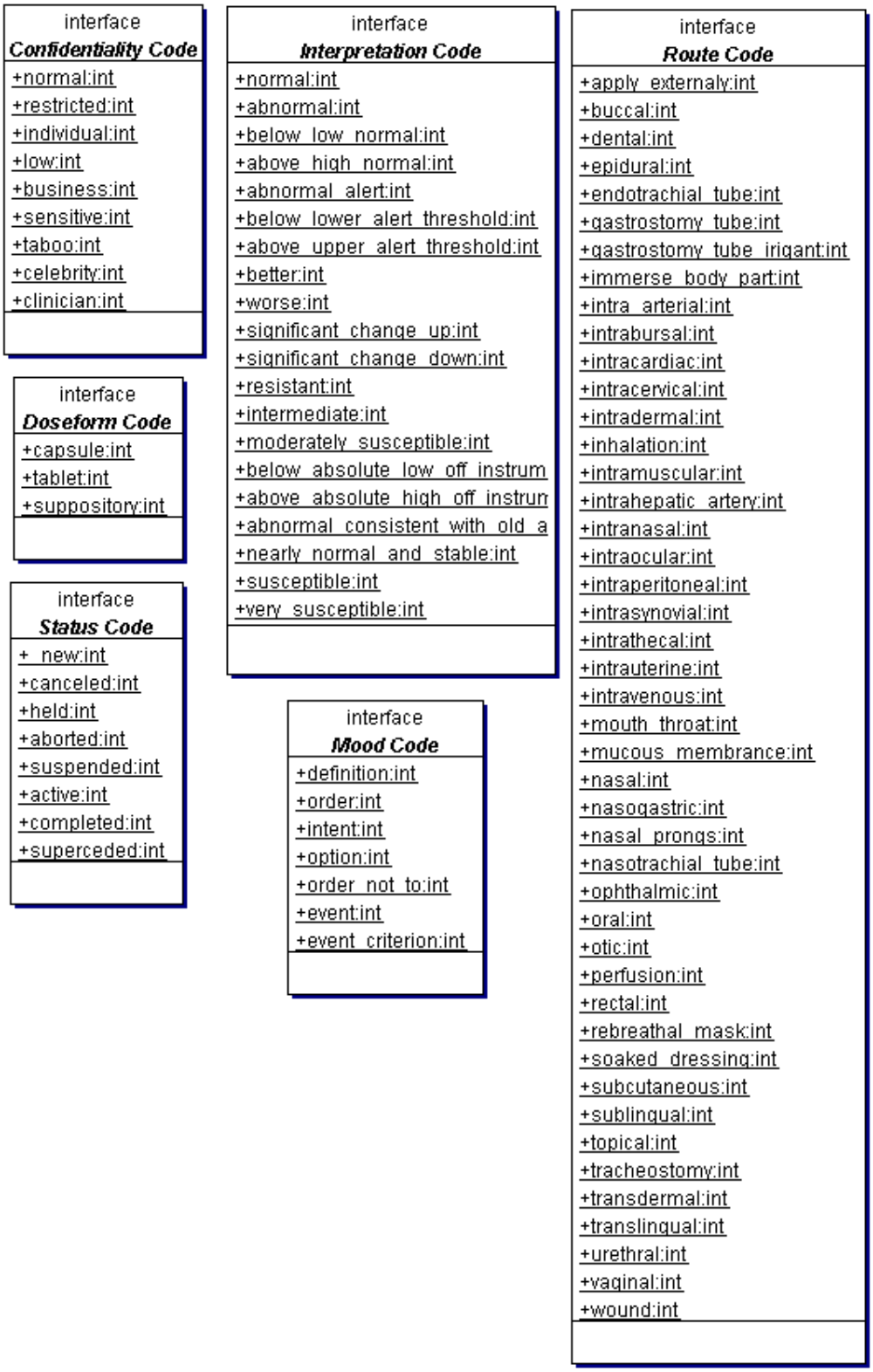


Figure 10. The codes used by the USAM-RIM ontology

Examples of USAM Observation and Medication are shown in Figure 11 and Figure 12, respectively.

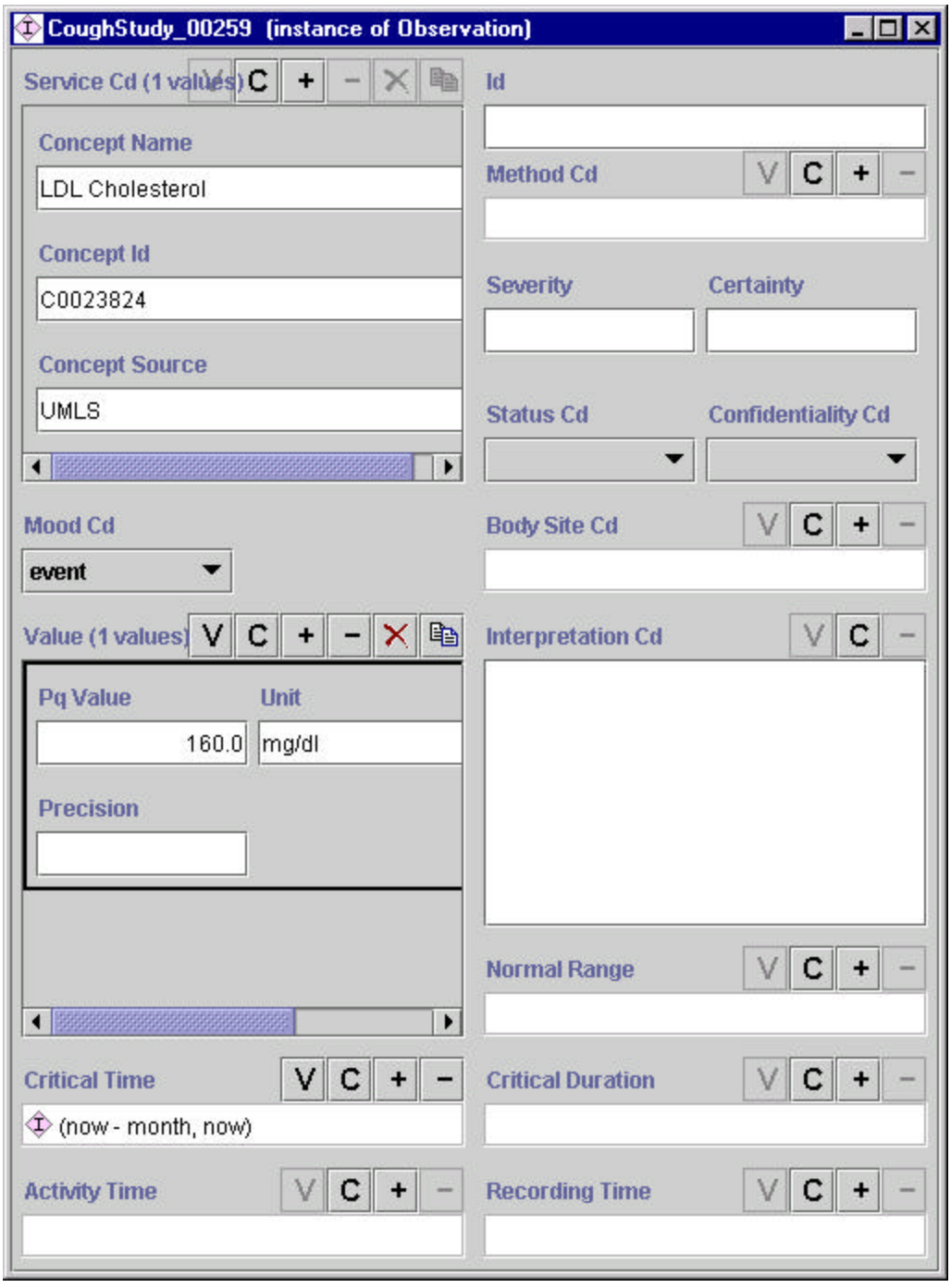


Figure 11. An example of a USAM observation showing an IDL cholesterol of 160 mg/dL that was taken within the past month

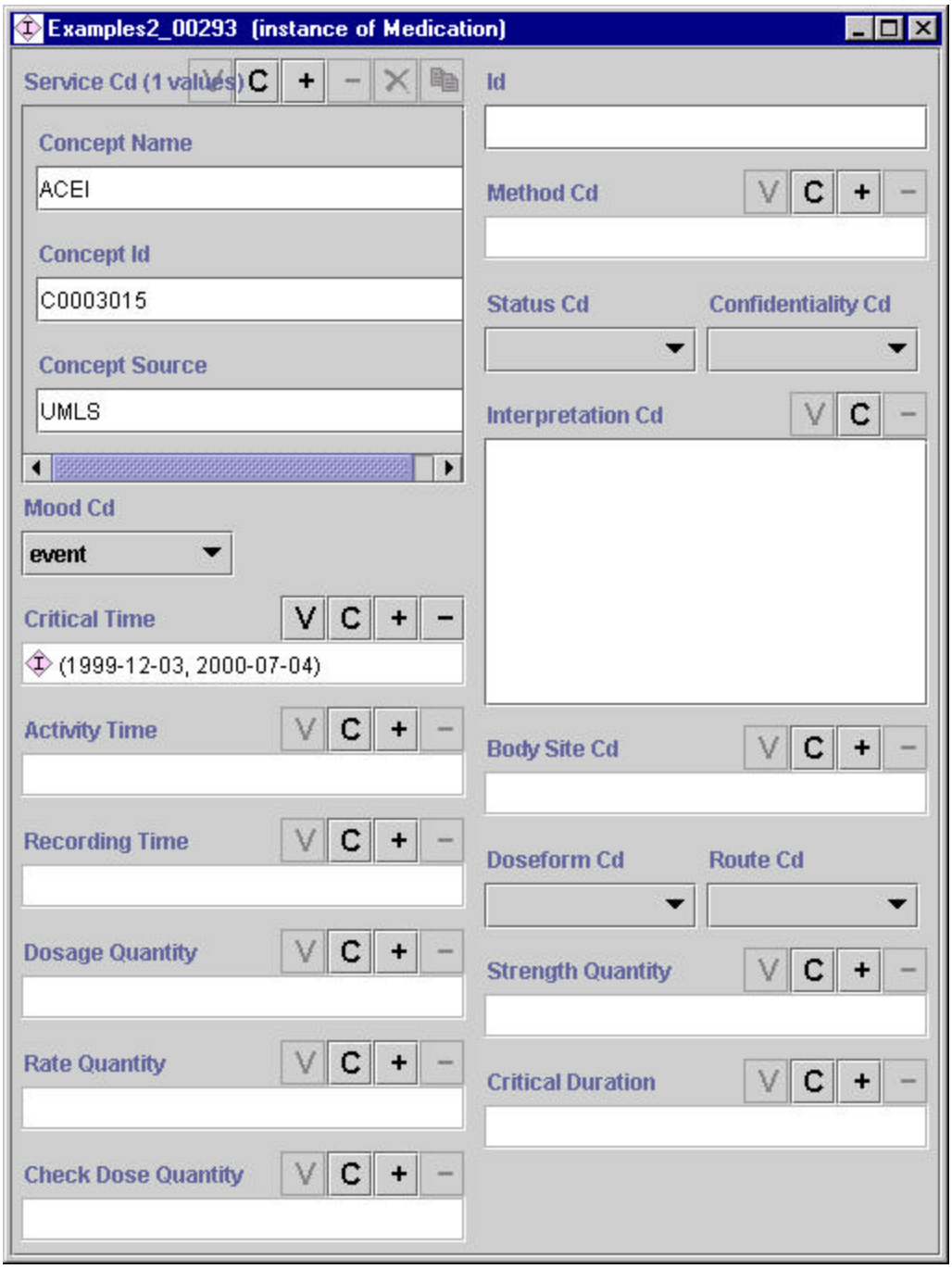


Figure 12. An example of a USAM medication showing that ACE inhibitor was used for some time interval

2.4.3 The Medical Knowledge Layer

As was mentioned before, the medical knowledge layer will contain interfaces to controlled vocabularies, medical knowledge bases, and EMRs. It will be specified in terms of the methods that it should have for interfacing to the medical knowledge sources.

The medical knowledge layer is still under development. Nonetheless, we do view this layer as a very important part of the GLIF ontology, especially for the purpose of integration into local institutional environments.

3. Creating a guideline

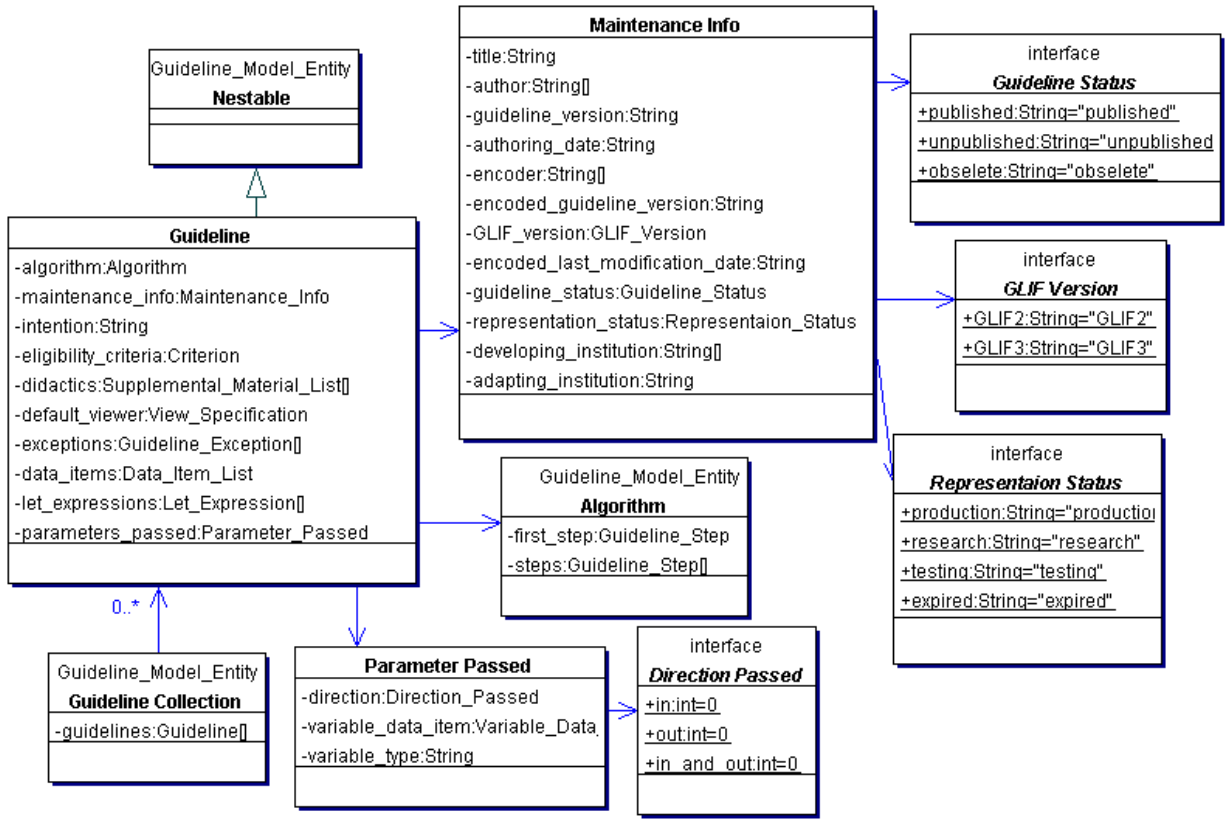


Figure 13. The Guideline Package

The **Guideline** class is used to model clinical guidelines and sub-guidelines (described in Section 5.3.1). A guideline contains an **Algorithm**, which is a flowchart of guideline steps. GLIF's guideline class specifies **Maintenance_info**rmation (such as author, guideline_status, encoded_last_modification date, and guideline_version), the intention of the guideline, eligibility criteria, didactics, and the set of exceptions that interrupt the normal flow of execution of the guideline. The guideline defines patient data items that are accessed by it and parameters that the guideline passes in and out to other sub-guidelines. A guideline also has "let expressions" that

define global definitions (see Section 3.8). For each guideline, default viewers may be specified. Since different users may be interested in different parts of a large, complex guideline, differential display capability is supported. This capability is provided through the use of filters that collapse segments of the guideline into a default view of the guideline customized to a given user, situation, etc.

A **Guideline_Collection** object identifies the primary (top-level) guidelines in a guideline file. There is only one Guideline_Collection object per guideline file. There are other guideline objects in a file. These are subguidelines.

An example of a GLIF-encoded guideline that was authored using the Protégé authoring tool is shown in Figure 14.

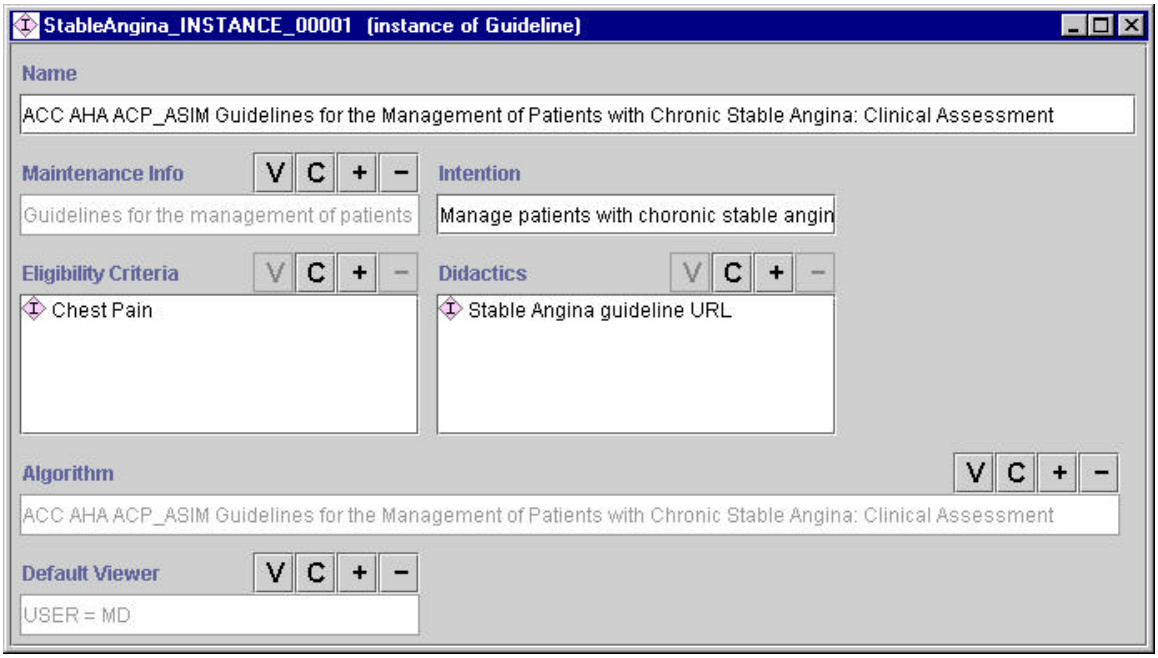


Figure 14. A Stable Angina guideline that was encoded in GLIF using the Protégé authoring tool

3.1 *Header information*

The **Maintenance_Info** class, shown in Figure 13, represents maintenance information related to guidelines. An example is shown in Figure 15.

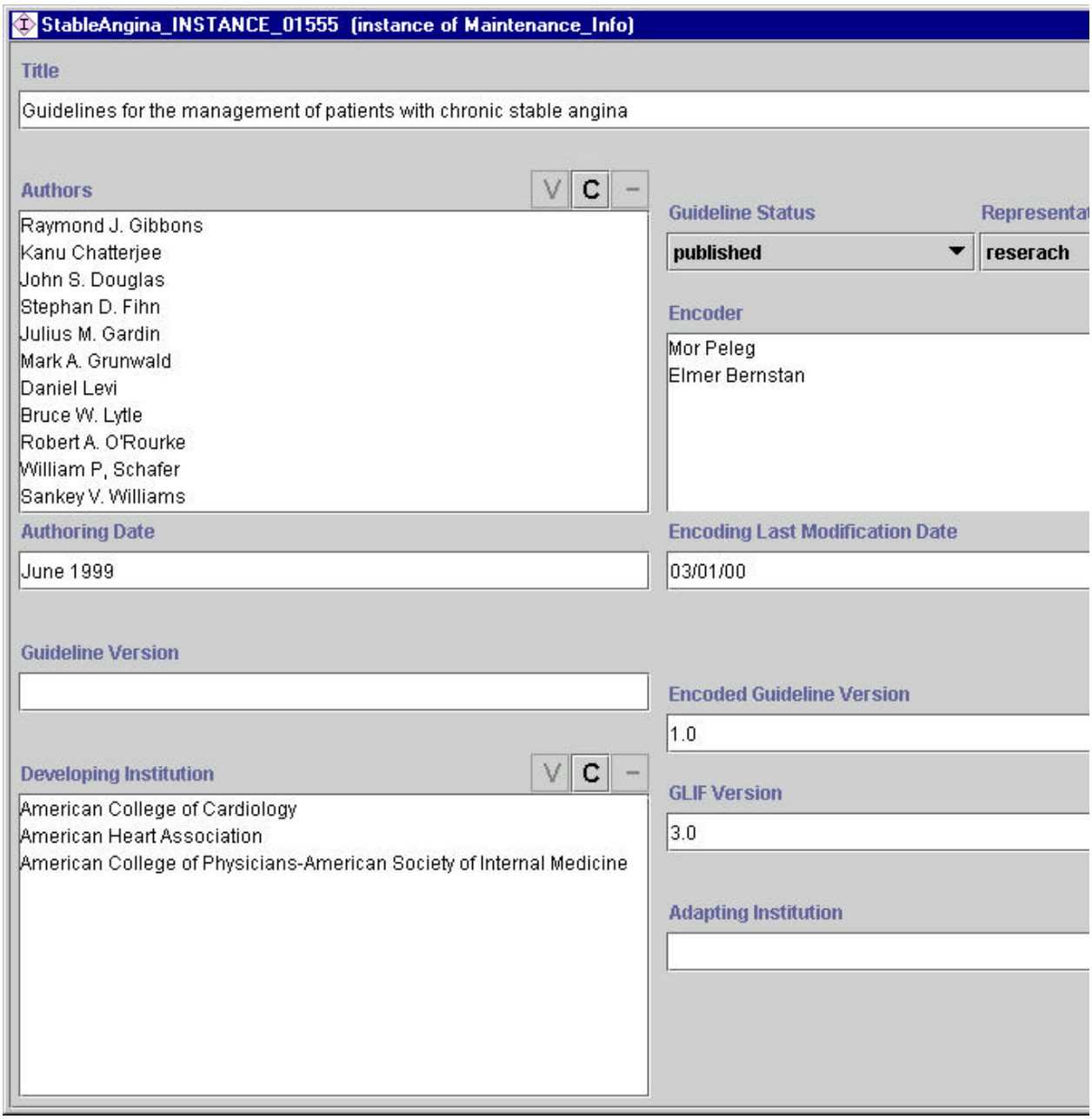


Figure 15. Maintenance information of the stable angina guideline shown in Figure 14

3.2 *Parameter passing*

By default, data items are not shared between guideline and sub-guideline. The reason for this is that guidelines and sub-guideline can be relatively independent of each other and may not be created by the same authors. Each sub-guideline has a data-items list that lists all the data items that it uses.

Sub-guideline sometimes may require some data from the calling guideline. Such needs should be explicitly declared in the form of a parameters-passed list. For each parameter in the list, the

permitted passing direction (IN, OUT, IN/OUT). IN means that the parameter value may be read but not written. OUT means that parameter value may set, but cannot acquire values from outside the sub-guideline. IN/OUT means that the parameter value may be both read from the outside, and reset. The parameters passed can be data items or variables. A pointer that points to them specifies data items. Variables are specified by indicating their name and type.

Referencing a sub-guideline transfers control from one guideline to another.

The figures below show an example of a main guideline (treatment of cough) that passes parameters in and out to a sub-guideline called "cessation of smoking/ACEI.

Guidelines need to be aware of the data items that they use (in decision criteria and action specifications). They also should define parameters that are passed to them and/or that they pass out. For example, the treatment guideline, whose algorithm is shown in Figure 17, defines several data items in its data_item_list slot, as shown in Figure 16. Some of these data items (e.g., pregnancy) are not parameters that need to be passed to other guidelines. The parameters_passed slot specifies the parameters that need to be passed in or out of other (sub) guidelines. For example, the treatment guideline has a sub-guideline called "cessation of smoking/ACEI". The sub-guideline needs to "read" the following attributes from the outer treatment guideline: ACEI, smoker, cough, and X_Ray_done. Therefore, the treatment guideline needs to export these parameters out (to the sub-guideline). The cessation of smoking/ACEI sub-guideline may perform an X-ray. It will then need to change (export out) the values of the following data items: X-Ray_done and X-Ray_result. The treatment guideline will therefore need to import (read in) these two data items. Similarly, Further_Test_Can_Be_Ordered needs to be passed in and out between the treatment and cessation of smoking/ACEI guidelines (see also Figure 18). The Time_Of_Stopping_Smoking_ACEI_Order needs to be passed out of the cessation guideline and into the treatment guideline.

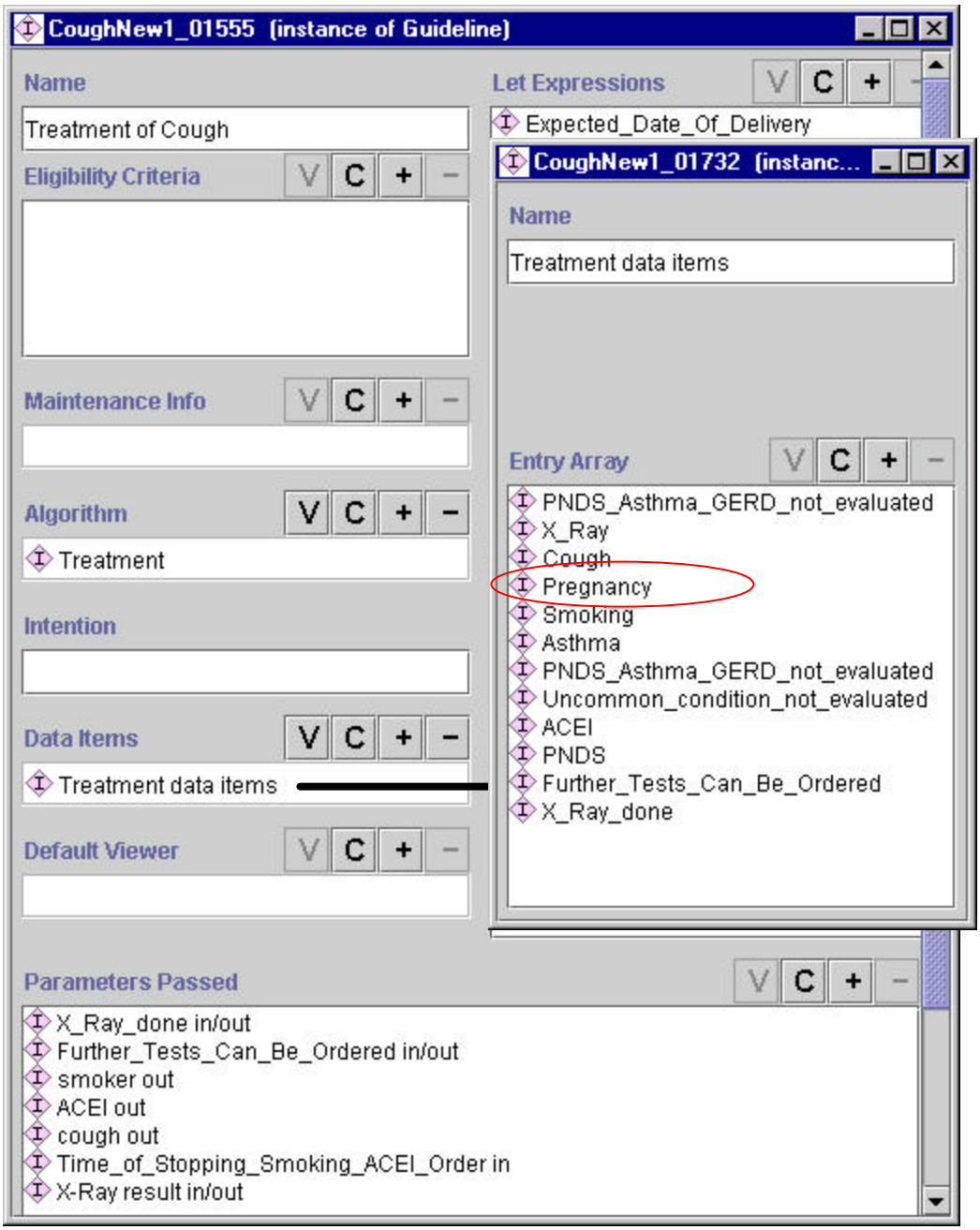


Figure 16. The "treatment of cough" guideline and the lists of data items that it uses and parameter that it passes/are passed to it by other guidelines.

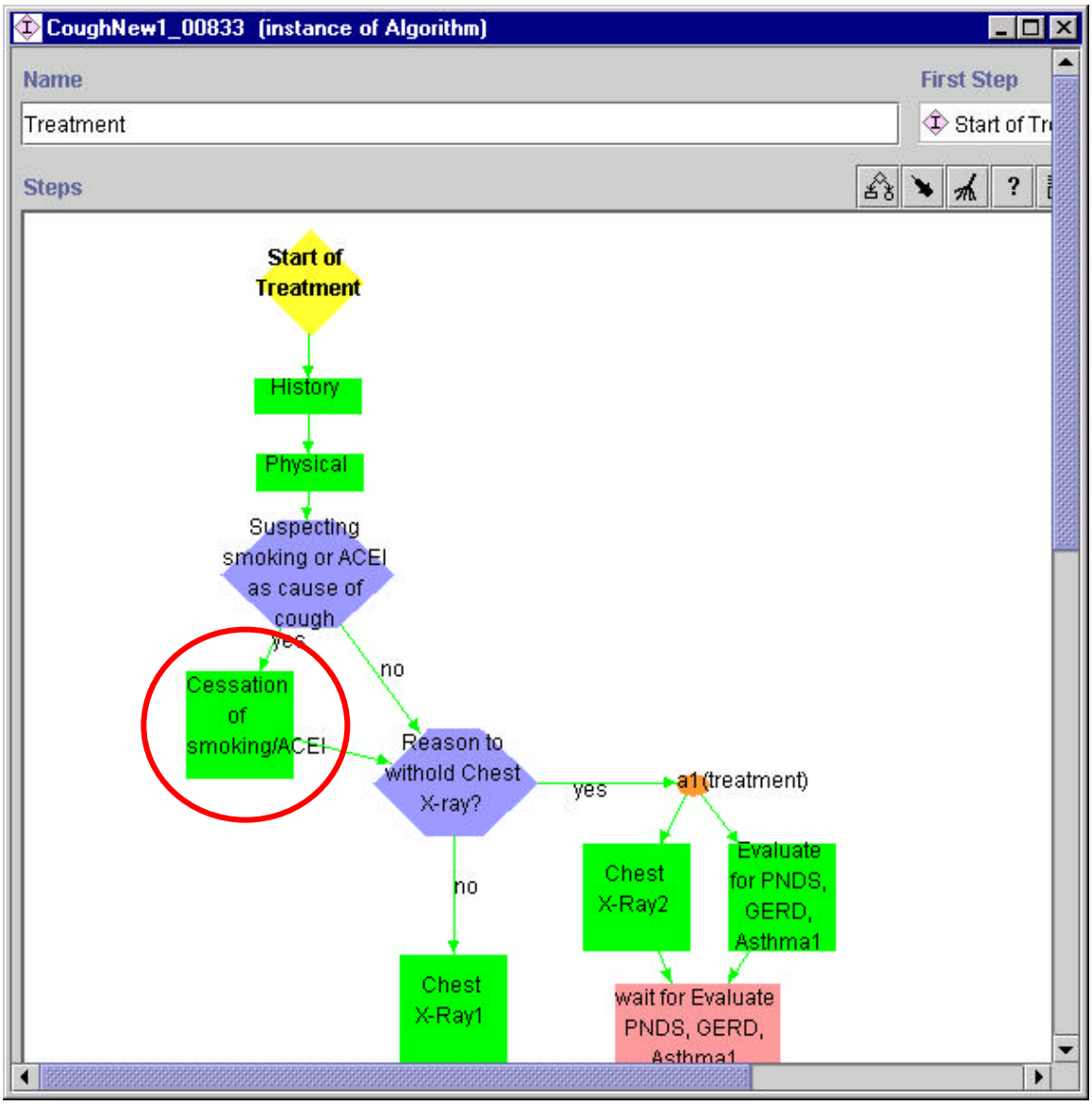


Figure 17. The treatment of cough algorithm that calls a sub-guideline called "cessation of smoking/ACEI"

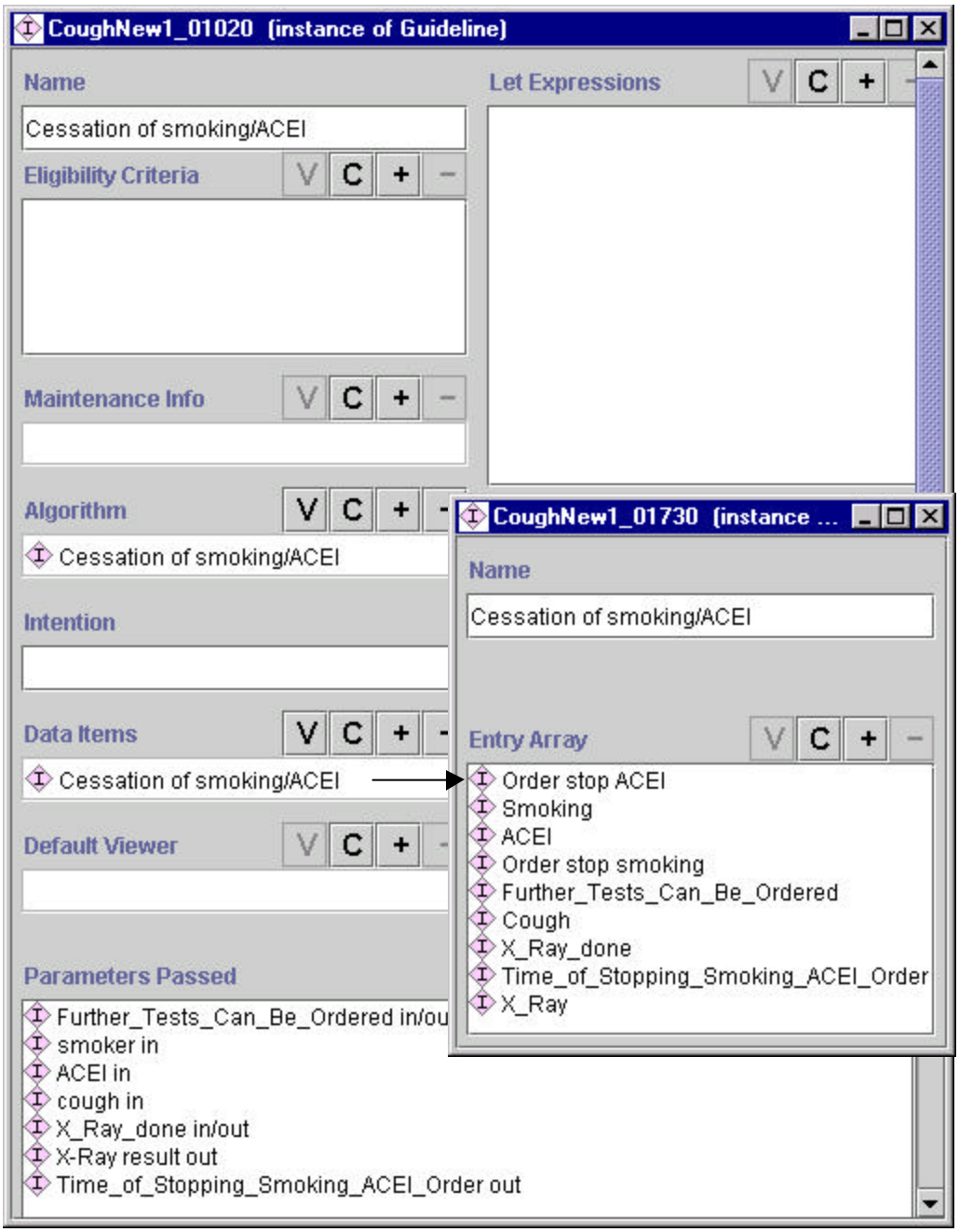


Figure 18. The "cessation of smoking/ACEI" guideline and the lists of data items that it uses and parameter that it passes/are passed to it by the treatment of cough guideline.

3.3 *Building the flowchart*

The flowchart is an instance of the Algorithm class. It may contain one or more instances from any of 5 classes of guideline steps: action, decision, branch, synchronization, and patient state (see sections 3.4, 3.5, 3.6, 3.7, and 6. The *first_step* attribute indicates the starting point of the algorithm. Next step, branches, and options attributes of the algorithm's guideline steps provide the flow among the steps of the algorithm. Examples of algorithms can be seen in Figure 19 and Figure 20. The Algorithm class diagram is shown in Figure 13.

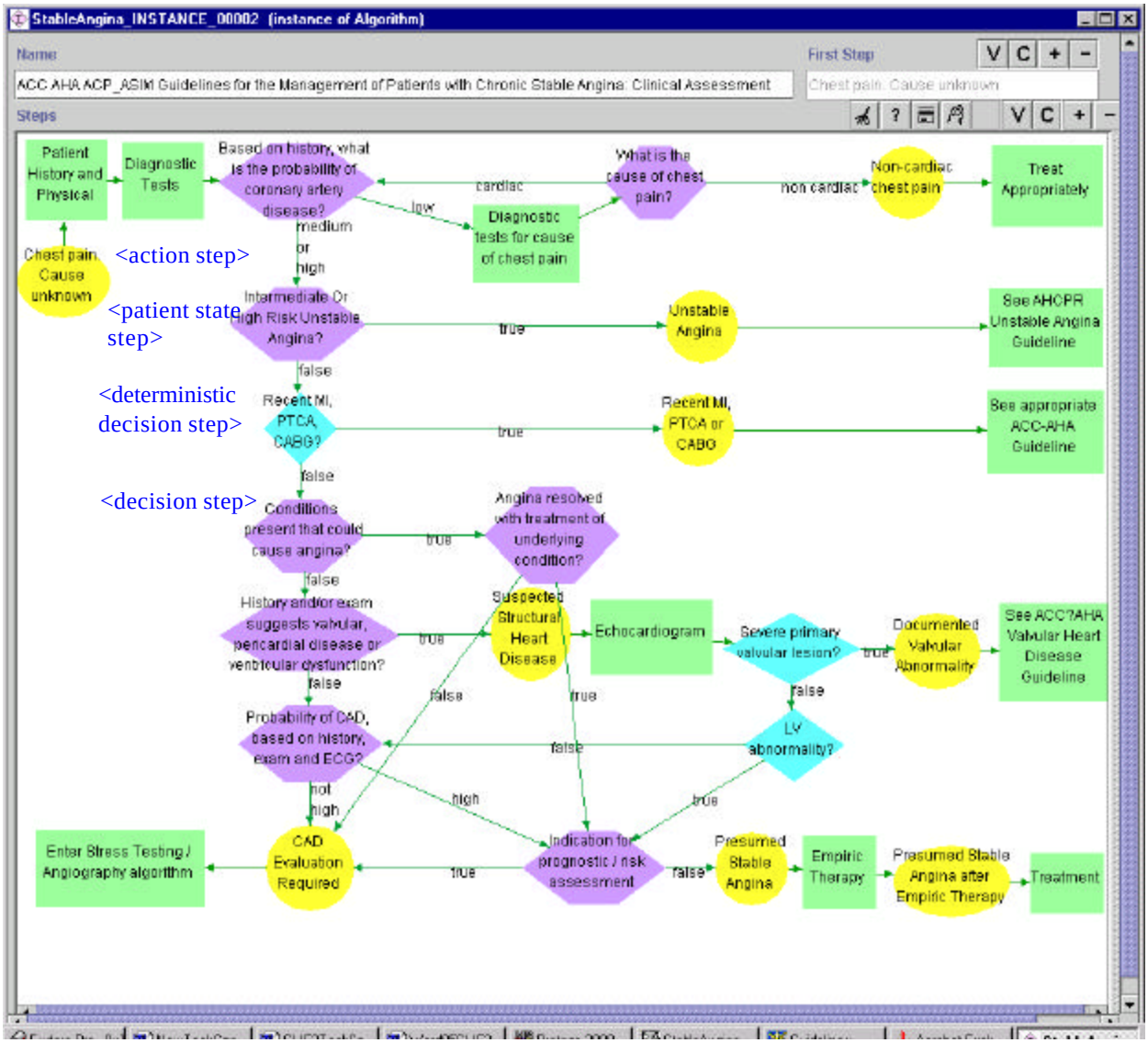


Figure 19. An algorithm for the stable angina guideline shown in Figure 14

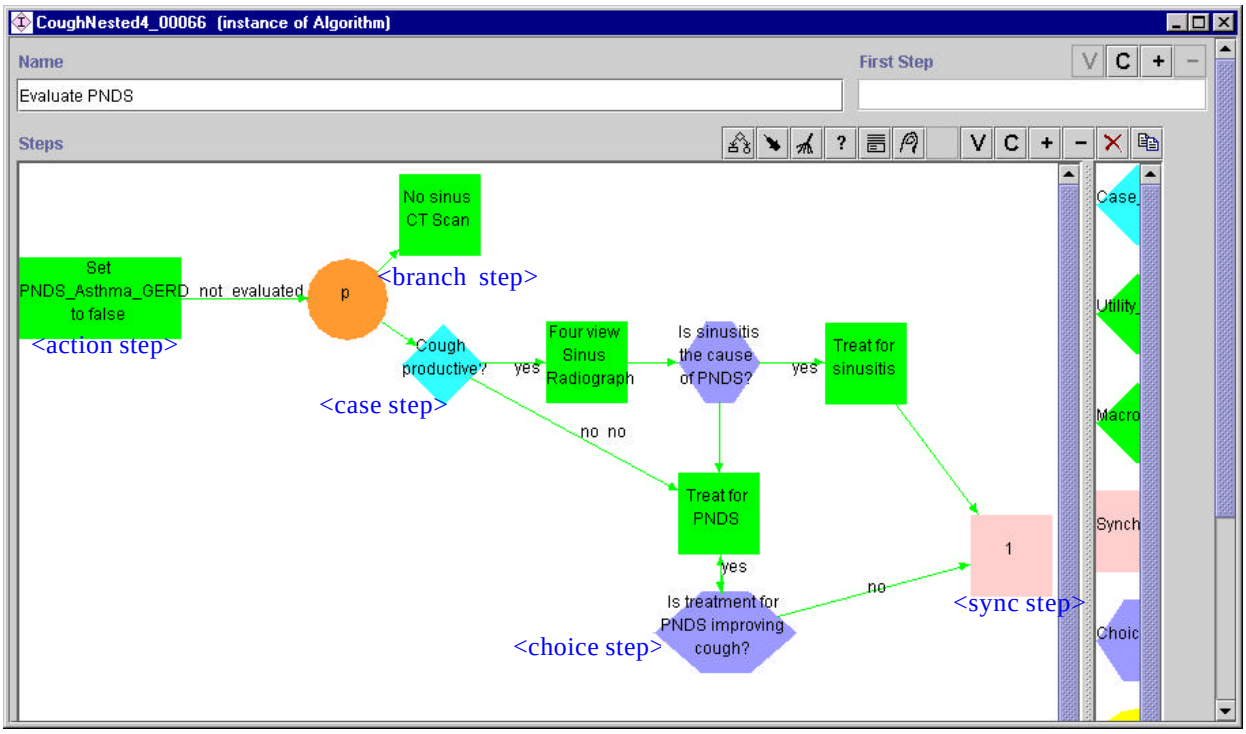


Figure 20. The algorithm for evaluating Post Nasal Drip Syndrome (PNDS) as the cause of chronic cough in immunocompetent adults.

As described earlier, the steps of the algorithm are subclasses of the Guideline_Step class. Each subclass is used for a step with a different purpose. Each step has a name and associated didactics. The hierarchy of guideline steps is shown in Figure 22.

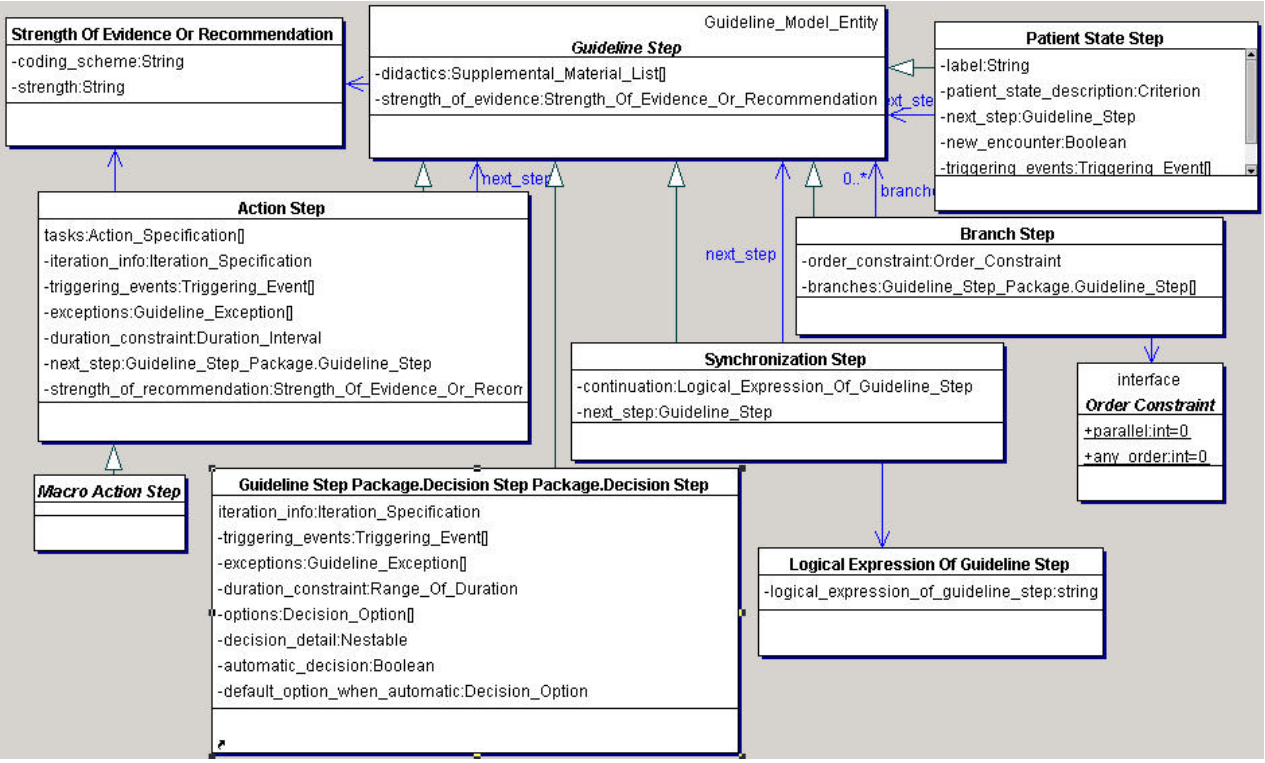


Figure 22. The Guideline_step class hierarchy

3.4 *Action Steps*

Action Steps specify clinical actions that are to be performed in the patient-care process. An action step specifies a set of tasks (Action_Specifications, discussed in Section 5.3) that need to be performed. The order in which the tasks are executed is not specified. The action step has attributes that specify its strength of recommendation, strength of evidence[1], didactics, iteration information, duration range, triggering events, and associated exceptions (events and exceptions are discussed in Section 3 of Appendix A. Action Steps can be refined by including a task of Subguideline_Action type in the step. The Subguideline_Action task has a (sub)guideline attribute that contains the nested subguideline. An action step has a next step attribute that is used to specify the step to go to once this step has finished execution. When a guideline step has finished its execution and the control flow is about to pass to the next step, then, if the next step has associated triggering events, then this next step is executed only after one of its triggering event occurred. An example of an action step is shown in Figure 23. The class diagram of the action step is shown in Figure 22.

[1] Strength of evidence marks the way the guideline authors evaluate the strength of evidence that supports a recommendation. Strength of recommendation indicates whether the guideline authors want the physician to follow the recommendation in every case, or do they relax the recommendation

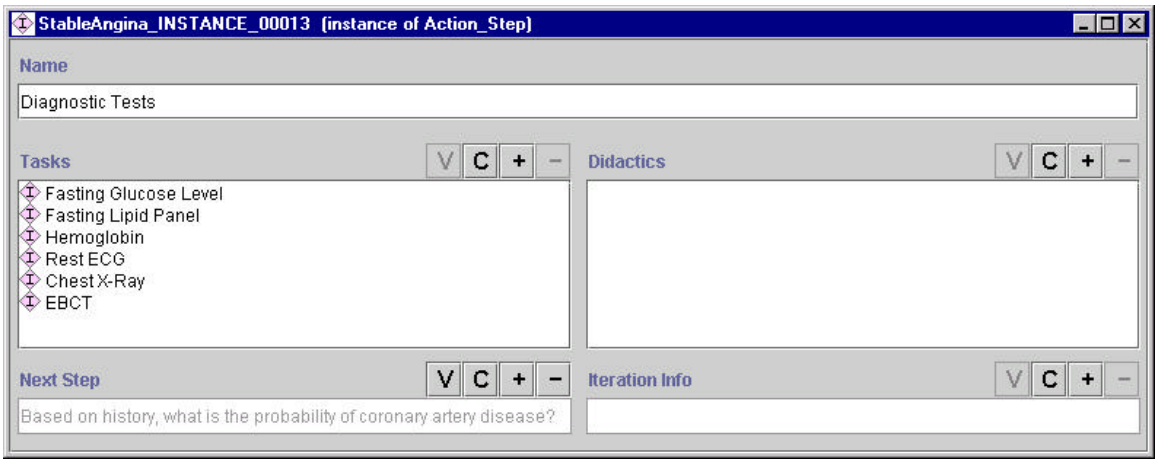


Figure 23. An example of an action step

3.5 *Decision Steps*

Decision steps, shown in Figure 31, *conditionally* direct flow from one guideline step to another. GLIF provides a flexible decision model through a hierarchy of decision step classes. The Decision Step allows specification of deterministic as well as non-deterministic decisions. Examples of decision steps are shown in Figure 20. The decision hierarchy can be extended in the future to model decisions that consider uncertainty or patient preferences. The hierarchy might be extended to support different decision models.

Decision Steps are nested by specifying a (sub)guideline in the decision_detail attribute of the step. This subguideline is executed before the decision criterion for that step is evaluated. The subguideline would modify or create new variable data items and assign them values. The use of these variables in the decision criteria makes the decision nested. An example of a nested decision step is shown in Figure 54 and Figure 55. Like the action step, a decision step has attributes that specify its strength of recommendation, strength of evidence, didactics, iteration information, duration range, triggering events, and associated exceptions (events and exceptions are discussed in Section 3 of Appendix A.

The decision hierarchy is discussed in greater detail in Section 4.1.

3.6 *Branch Steps*

The branch step is used to model concurrent guideline steps. Branch steps direct flow to multiple guideline steps. All of these guideline steps must occur in parallel. A branch step may link a guideline step to any other guideline step. An example of a branch step is shown in Figure 24. The class diagram of a branch step is shown in Figure 22.

The selection method (e.g., “one of”) that characterized the branch step in GLIF2 was removed so that the branch step would not semantically overlap the case step.

Like every other guideline step, branch steps have didactics and strength of evidence.

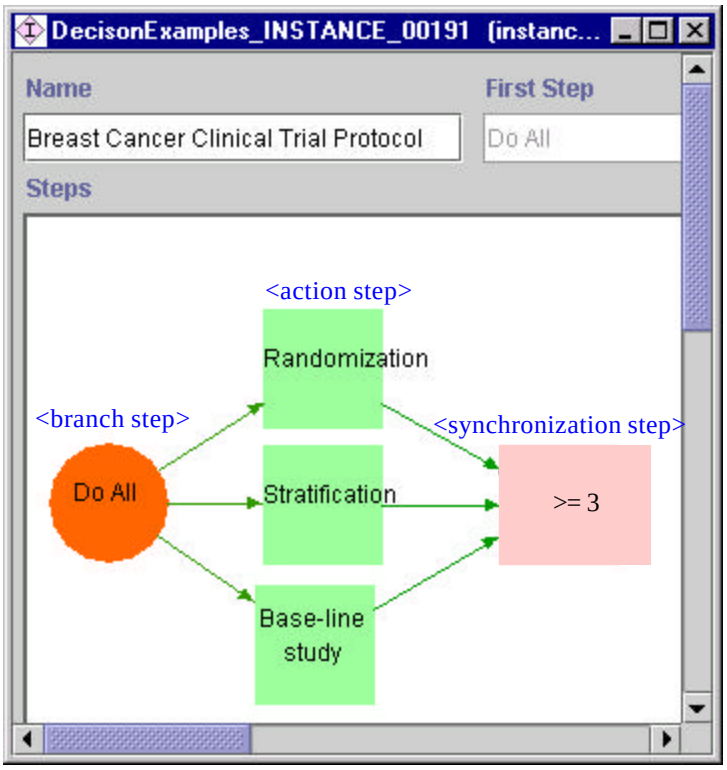


Figure 24. An example of branch and synchronization steps

3.7 *Synchronization Steps*

Synchronization steps are used in conjunction with branch steps. When multiple guideline steps follow a branch step, the flow of control can eventually converge in a single step. Each branch may lead to a series of steps, resulting in a set of branching paths. The step at which the paths converge is the synchronization step. When the flow of control reaches the synchronization step, a continuation attribute specifies whether all, some, or one of the preceding steps must have been completed before control can move to the next step. The continuation is expressed as a logical expression of guideline steps (e.g., ((Step_A or Step_B) indicates that flow must continue once either Step A or Step B are completed). The syntax of the expressions for specifying continuation is as follows:

Logical_expression_of_guideline_steps: **Guideline_Step** |
(Logical_expression_of_guideline_steps**)** | **not** Logical_expression_of_guideline_steps |
Logical_expression_of_guideline_steps **and** Logical_expression_of_guideline_steps |
Logical_expression_of_guideline_steps **or** Logical_expression_of_guideline_steps | >= Integer

Like every other guideline step, synchronization steps have didactics and strength of evidence.

3.8 *First look at expressions*

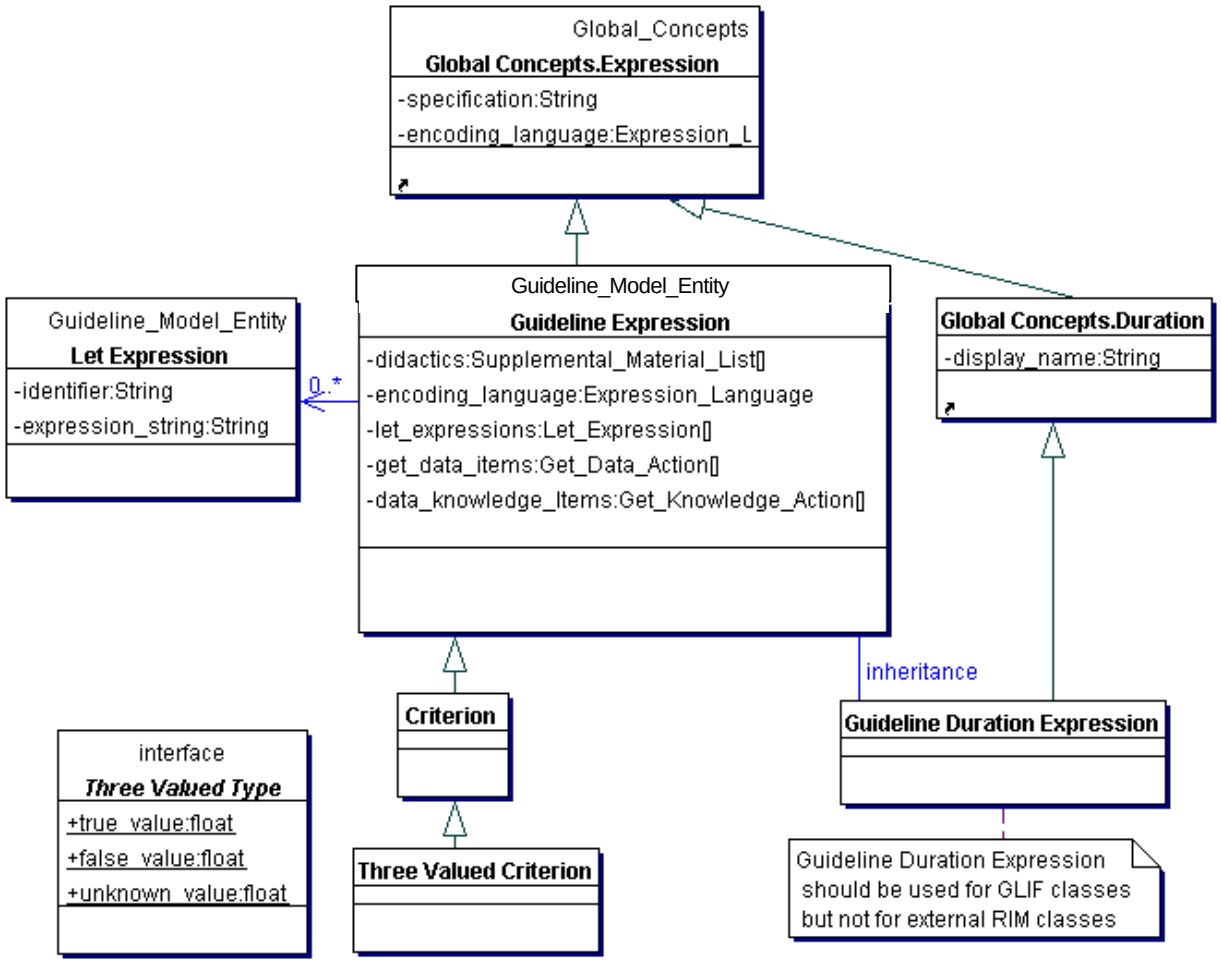


Figure 25. The Guideline_Expression hierarchy

The Guideline_Expression class is a parent class for all expressions, whether they are (logical) criteria (e.g., Age > 32), or simply expressions (e.g., Age). Expressions may have arithmetical or text data items, can contain temporal information, and can refer to single variable data items or to lists of data items. Examples of different expressions and criteria are shown in Figure 26. Different expression languages can be used. Currently, the expression classes make use of an expression language called Guideline Expression Language (GEL) [13] that is based on Arden Syntax. A BNF grammar for GEL as well as a list of operators that are part of GEL but are not present in Arden Syntax and vice versa are presented in Appendix B. The **Get_Data_Action_Specification** (see Section 5.3.4) is used to retrieve data item values from EMRs. The retrieved data is presented in the form of a Query_Result that can be use by GEL expressions or criteria.

Arithmetic expressions:

heart_beats_per_min / 60
5 + 6 * 7 / 8

Criterion that involves string literal data item:

test_name == "Serum_Potassium" (test_name is a variable data item)

Criterion that involves a list of variable data items

Cough is in Problem_list (where Cough is a concept, and Problem_list is a list of concepts)

Criteria that involves a single data item

Latest_LDL_Cholesterol_Test_Result < 160 mg/dL

selectAttribute("pq_value", selectAttribute("value", Current_LDL_Cholesterol)) >= 160 and
selectAttribute("unit", selectAttribute("value", Current_LDL_Cholesterol)) ==mg/dL)

Criteria that contain temporal operators

(smoking_end_time >= now and chronic_cough_end_time >= now)
latest_LDL_Cholesterol_Test_Result_recording_time is before 1998-12-20
latest_LDL_Cholesterol_Test_Result_recording_time is after week_3_of_pregnancy
latest_LDL_Cholesterol_Test_Result_recording_time is within past 15 days
latest_LDL_Cholesterol_Test_Result_recording_time is within 1999-12-03T20:46:01 to 1999-12-10T20:46:01
blood_pressure_reading occurs at 1995-03-20T18:30:15
previous_chemotherapy is not within past 2 years

Figure 26. Examples of GLIF_ARDEN expressions and criteria

Evaluating criteria

Currently, GLIF supports only three-valued criteria. In the future, probabilistic criteria might be added. The temporal criterion

(smoking_end_time >= now and chronic_cough_end_time >= now)

evaluates to "true" if the patient is a smoker and has a chronic cough. It evaluates to "false" if the patient is not a smoker, or does not have a chronic cough, or is neither a smoker nor has a chronic cough. It evaluates to either "unknown" or "false" if it is unclear whether the patient is a smoker or has a chronic cough. The interpretation of a non-existing value as false or unknown should be defined by the implementation and should depend on the data item.

Referring to time-literals

Time literals in GEL involve a specific instance in time (expressed as yyyy-mm-ddThh:mm:ss.millisec(Z|+/-hh:mm) based on Arden syntax notation, which, in turn, is based on the ISO standard 8601:1988. [14]. Z is the abbreviation used for Coordinated Universal time,

also known as the "zero meridian" time. When Z is not used, local time is assumed. The string +hh:mm can be added to the time to indicate that the used local time zone is hh hours and mm minutes ahead of UTC. For time zones west of the zero meridian, which are behind UTC, the notation -hh:mm is used instead. For example, Central European Time (CET) is +0100 and U.S./Canadian Eastern Standard Time (EST) is -0500. Examples of time literals are "1999-11-22T08:30:00", "2 days before 1999-11-22T08:30:00", "2000-09-19T12:31:42.435-04:00", and "2000-09-19T12:31:40.125Z".

Let Expression: Let expressions are used to define global definitions. At execution time, the identifier of a Let Expression is replaced by the expression_string of the Let Expression, just like in a macro substitution of programming languages. This occurs every time the identifier of the Let Expression is encountered. Let Expressions enable guideline authors to represent definitions that they can later on refer to. The example of Figure 27 shows that Age is defined as the current time "now" minus the date of birth (DOB), where DOB is a global variable data item and "now" is a globally defined temporal operator.

A Let Expression can be used to define global definitions or local definitions. If the let expression is defined as an attribute of a guideline object, then the let defines a global definition. If the let expression is defined as an attribute of a Guideline_Expression object, then the let defines a local definition.

Let Expressions are similar to Assignment Action Specifications, discussed in Section 5.3.2. The difference is that an Assignment Action assigns an identifier the result of the expression_string once. After that, the identifier's value remains constant throughout the execution of the guideline and is not reevaluated every time the identifier is encountered.

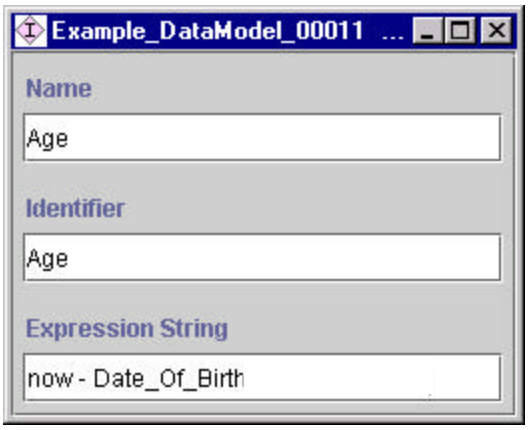


Figure 27. An example of a Let Expression

3.9 *Documenting the guideline*

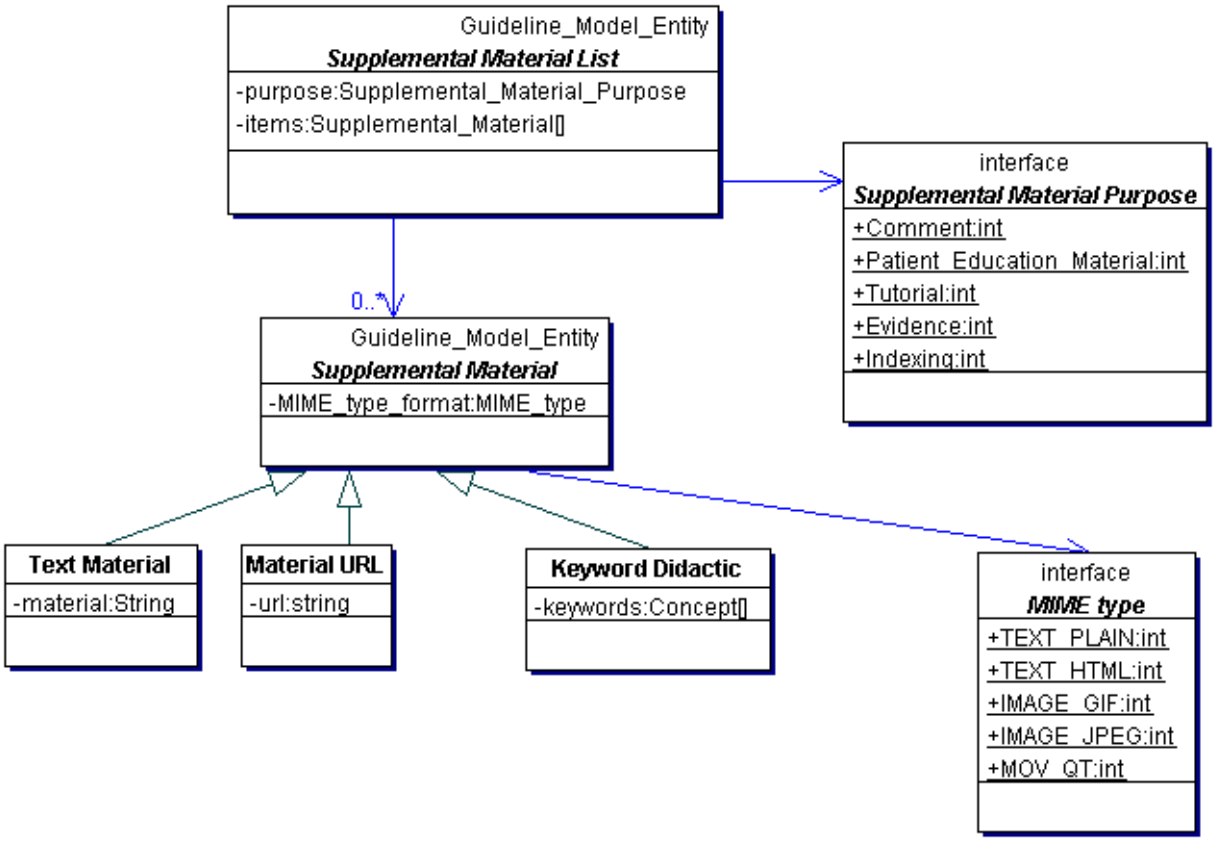


Figure 28. The supplemental material package class diagram.

Supplemental material can be used to include additional documentation for a guideline. Supplemental material can be of different formats such as text material, URLs, and keywords. The Supplemental_Material_List class is used to package a number of different supplemental material objects that serve the same purpose. The purpose of the Supplemental_Material_List class can be selected from the enumerated type Supplemental_Material_Purpose.

All the different formats of supplemental material are sub-classes of the Supplemental_Material class. All supplemental materials define their format through the Mime_Type_format attribute. The domain of this attribute is a Multipurpose Internet Mail Extensions (MIME) type such as text/plain, text/html, image/gif, and mov/qt.

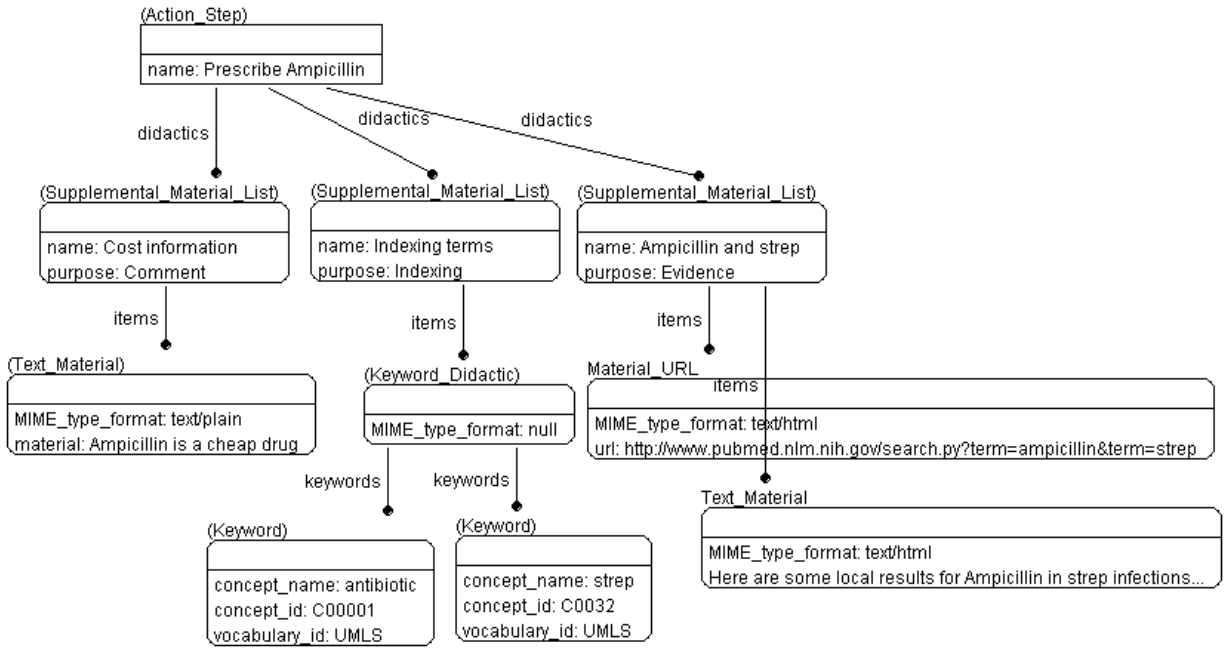


Figure 29. An example of supplemental material packages.

3.10 *The Global Concepts*

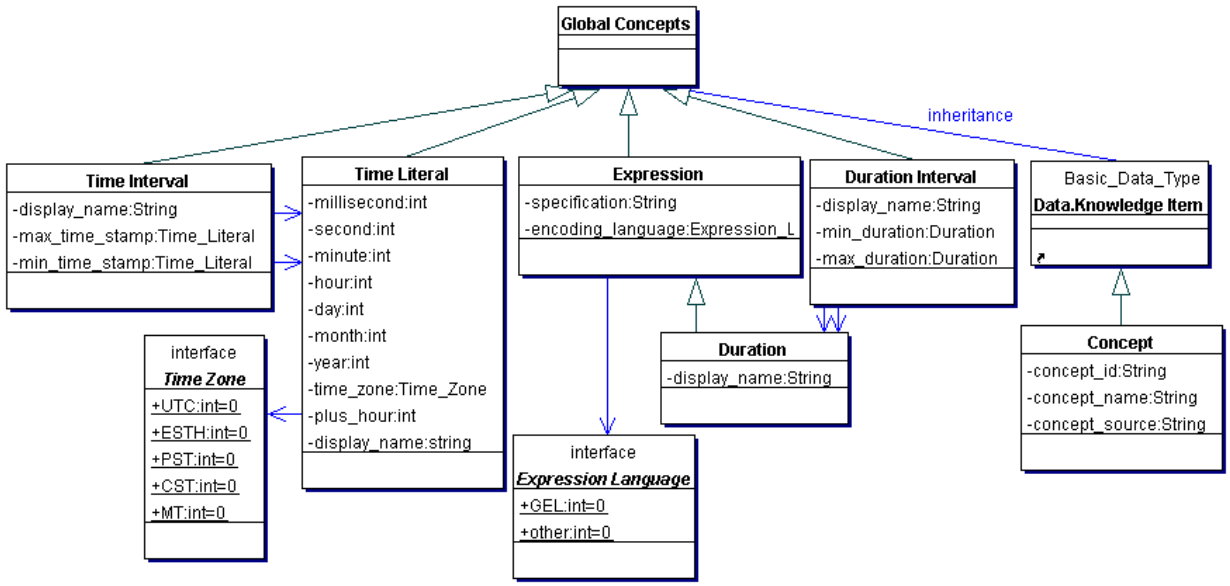


Figure 30. The global concepts package

In this package we define concepts that are applicable to many parts of the guideline model. For example, a medical concept is part of the RIM, the data model, and of supplemental material. Temporal constructs are part of the RIM and of iterations.

4. Specifying decisions

4.1 *Different types of decision steps*

Decision steps, shown in Figure 31, represent decision points where a choice has to be made among competitive, mutually exclusive alternatives (decision options). In automatic decisions, if the criteria specified in the decision option are met, then the control should flow to the step specified in that decision option. If there is no match, then the control flows to a default step indicated by the attribute *default_option_when_automatic*.

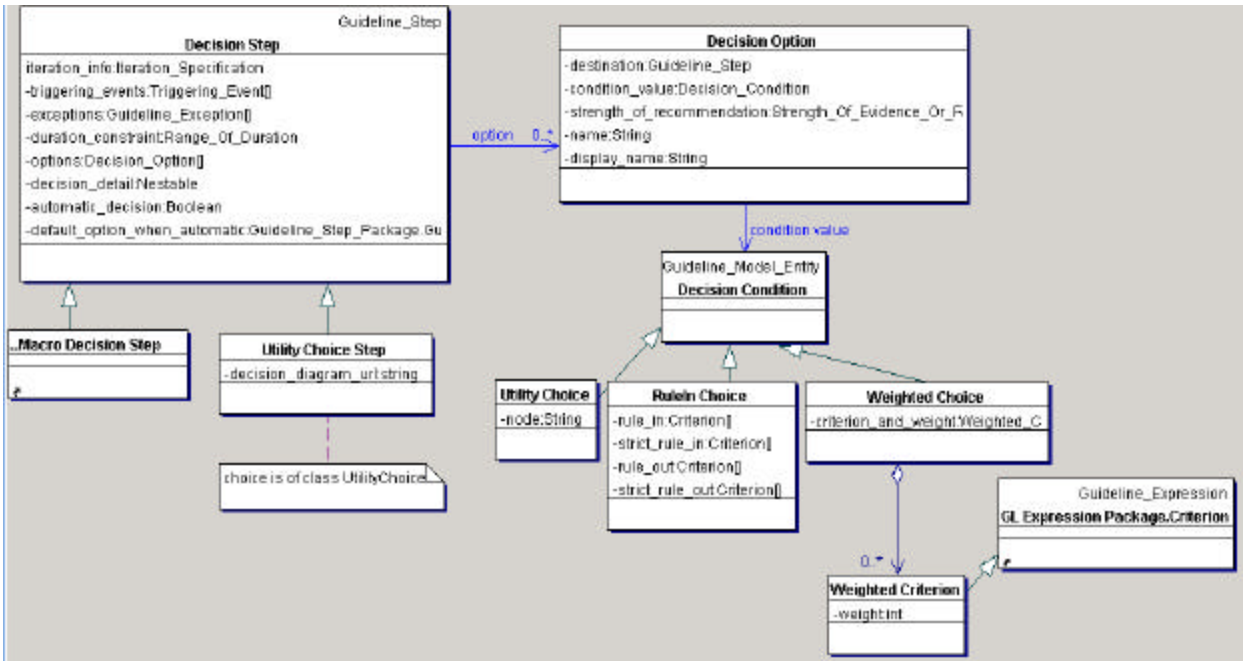


Figure 31. The decision step hierarchy

4.1.1 *Modeling deterministic one-of decisions (Previously known as Case Steps)*

Decision_Step can be used as a means to represent conditional selection of one and only one path from among several alternatives. This replaces GLIF2' s conditional step class, which used a Boolean model. The Boolean model made it cumbersome and error-prone to represent criteria that do not have a true-or-false result (e.g., selection based on the condition "patient's age category" has several options: neonate, infant, toddler, child, adolescence, adult, elderly).

To represent deterministic one-of, a decision step is linked to several decision options. The strict_rule_in attribute of each decision option is used to specify a decision condition that could be computed automatically. If a strict_rule_in evaluates to true, then the control flows to the guideline step that is specified by that decision option's destination.

The decision options' criteria in a case should be mutually exclusive. However, the responsibility of ensuring mutual exclusiveness is left to the guideline author. If these criteria are not mutually exclusive, and more than one decision option criteria are met, then only one decision option is chosen, arbitrarily. The GLIF specification does not define which of the options is selected in case of more than decision criterion being true.

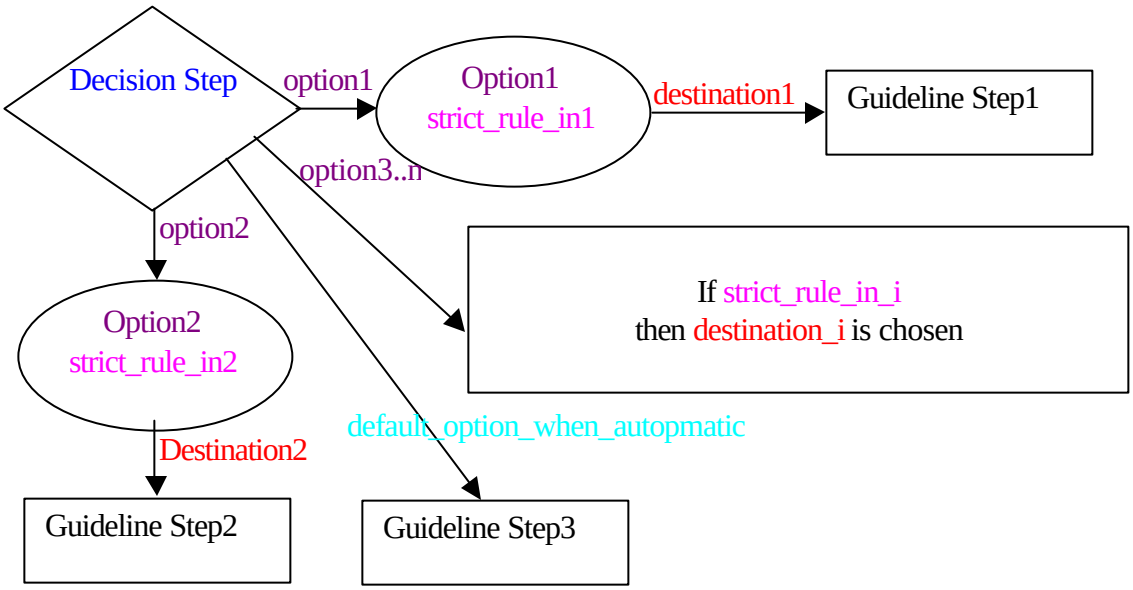


Figure 32. The way in which deterministic one-of decisions are modeled in GLIF3

Note that the decision options are not guideline steps. When using Protégé as an authoring tool for GLIF3, decision options are not graphically depicted as flowchart nodes. Instead, they are depicted as connectors, as shown in Figure 33.

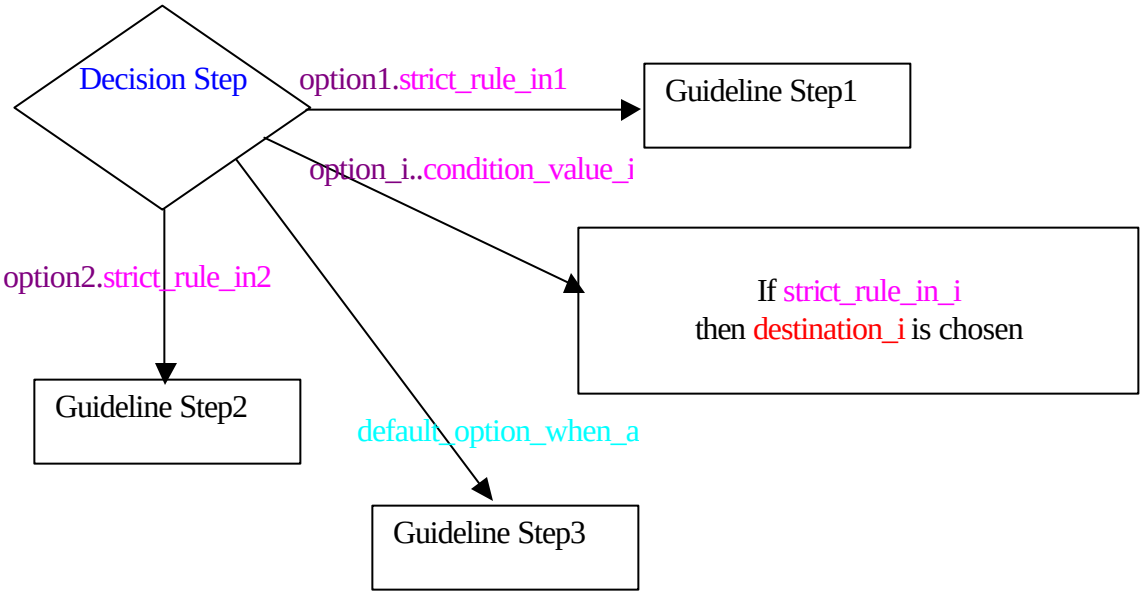


Figure 33. The way decision steps are graphically displayed by the Protégé GLIF authoring tool

An example of a case step is shown in Figure 35 through Figure 38.

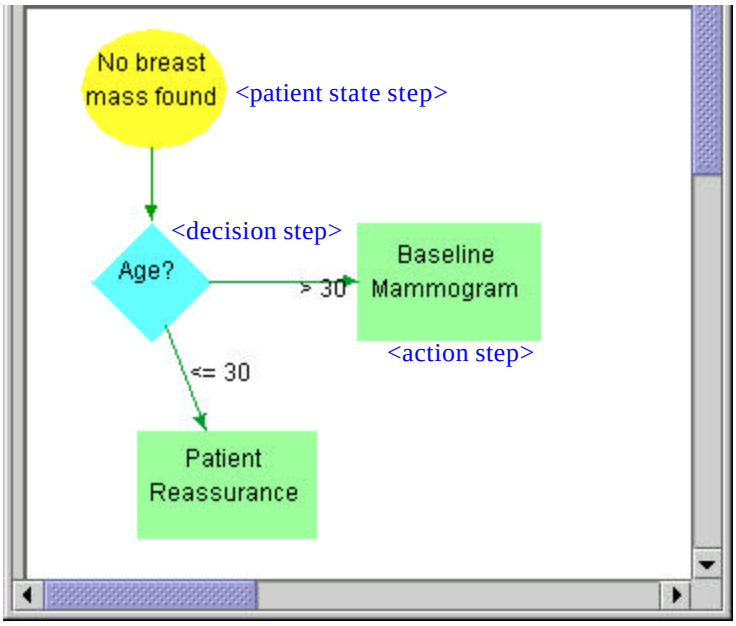


Figure 34. Deterministic one-of decision step used in the Breast Mass Workup algorithm. Only part of the algorithm is shown.

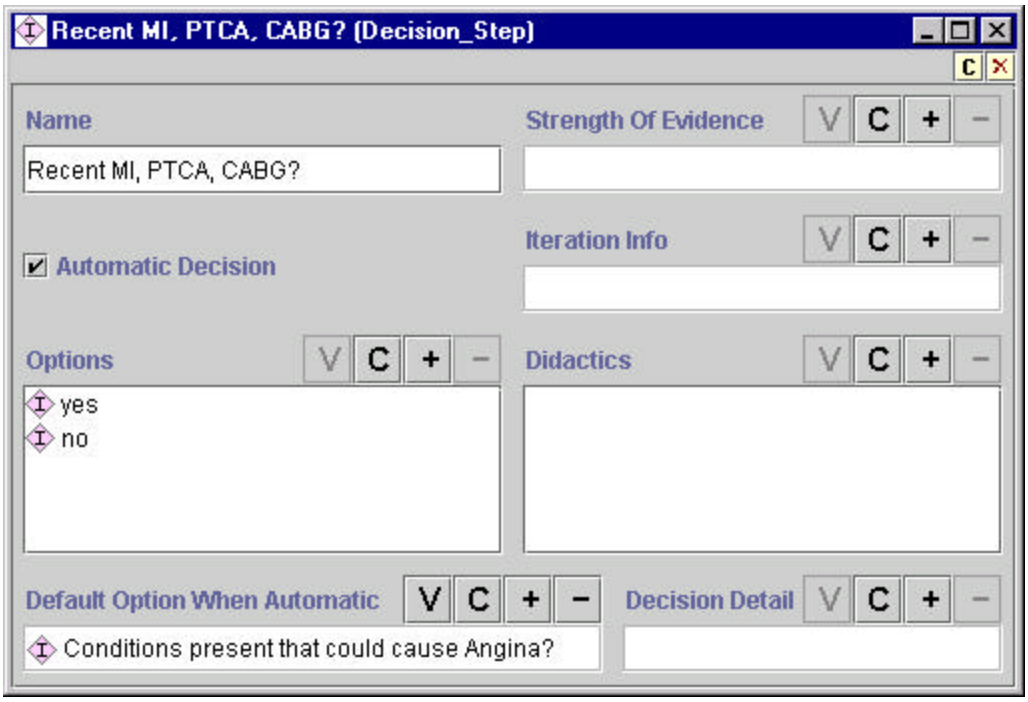


Figure 35. An example of a case step. This is one of the case steps shown in Figure 19.

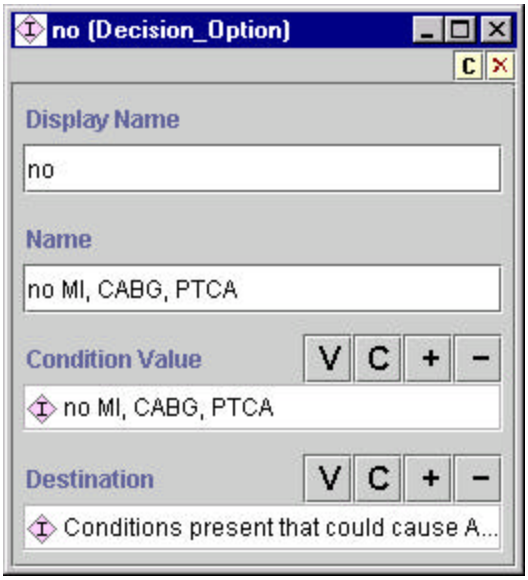


Figure 36. The details of the "no" option shown in Figure 35. When the expression "RecentMPC?", shown in Figure 35 evaluates to the condition value "No Recent MPC" control flows to the destination step "Conditions present that could cause angina?"

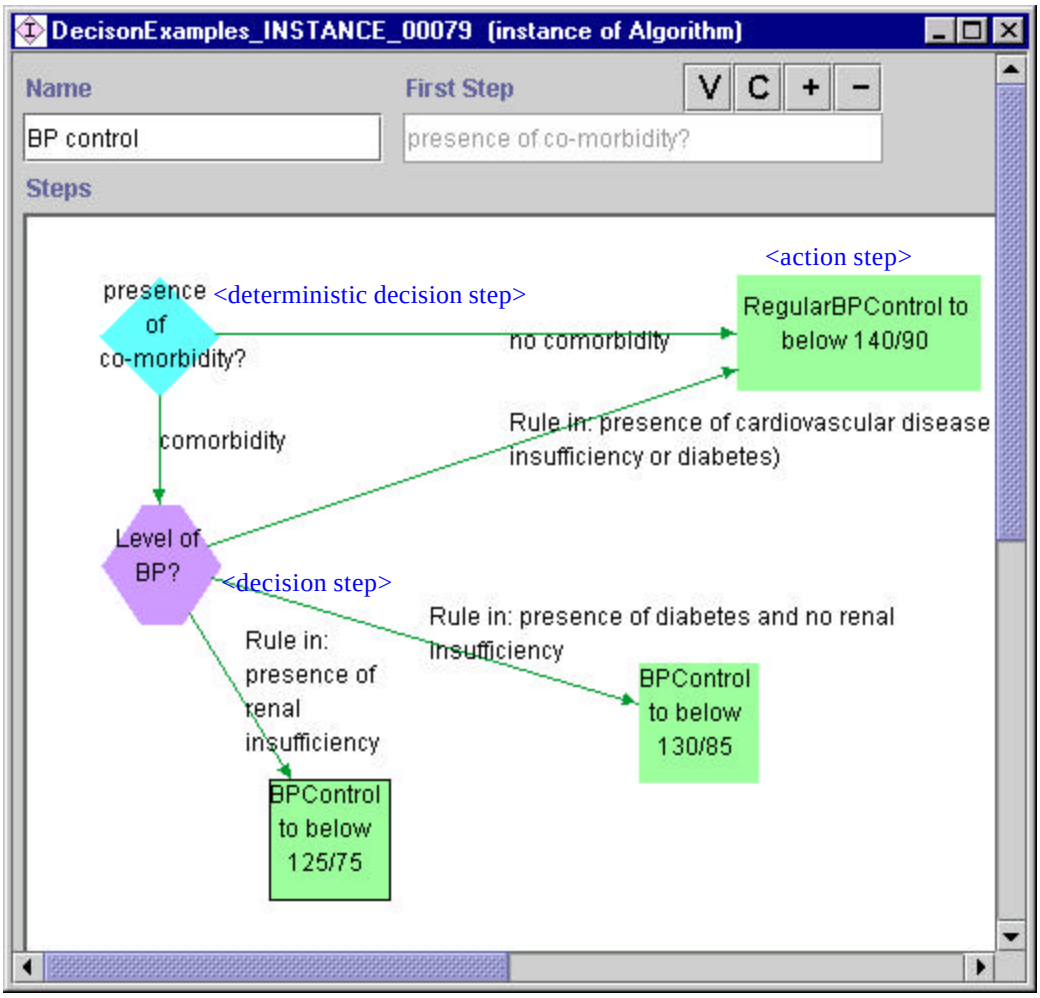


Figure 37. Deterministic decision step used in the BP Control algorithm

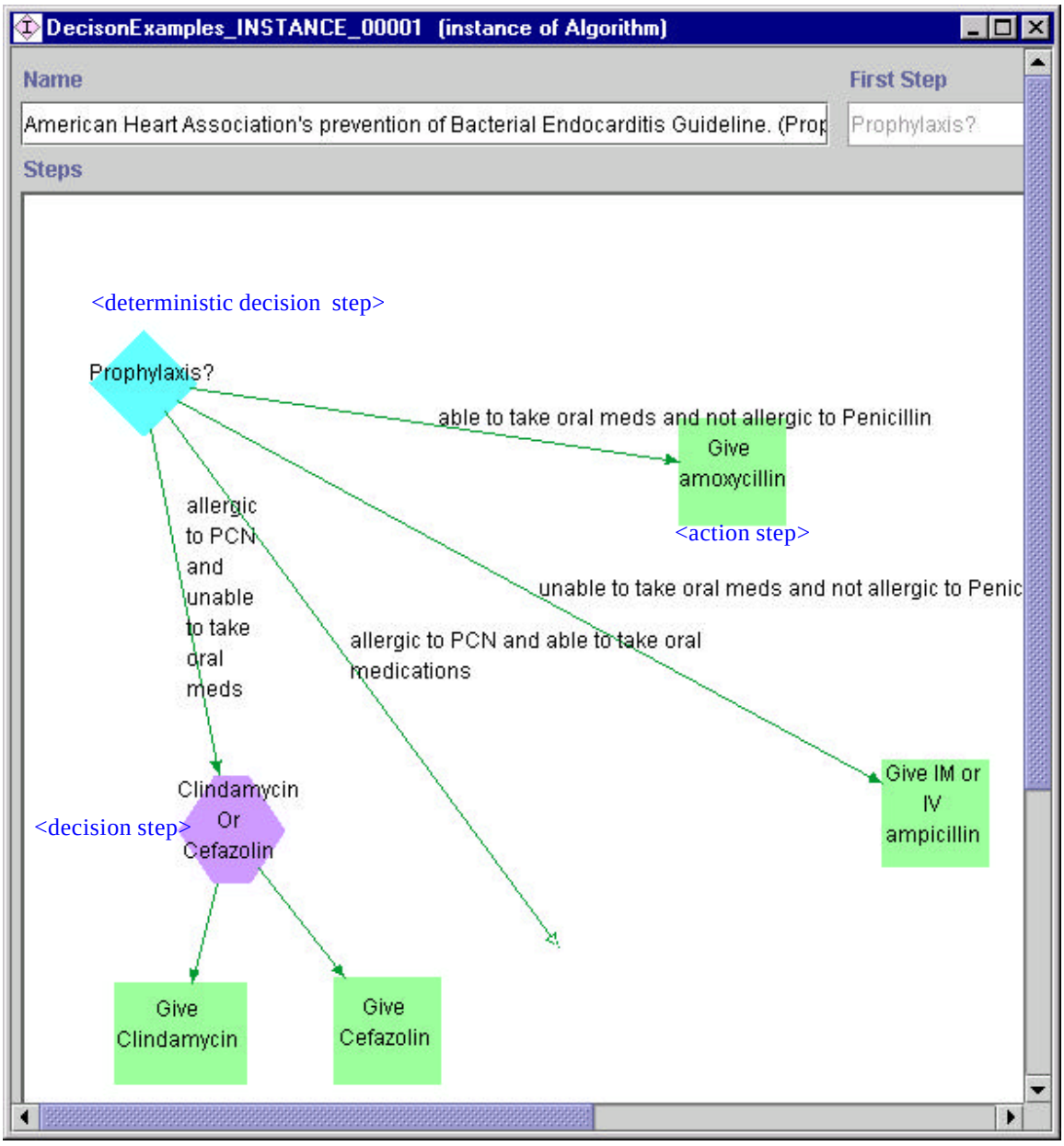


Figure 38. Deterministic decision step used in the prevention of bacterial endocarditis algorithm

4.1.2 Modeling non-deterministic decision Steps

Non-deterministic one-of decision steps represent a decision between guideline steps for which the guideline does not provide deterministic selection criteria. There are many reasons for using this construct such as when a decision cannot be represented unambiguously. The different decision options in a non-deterministic decision step are not necessarily mutually exclusive. The options.condition_value of a non-deterministic decision step must belong to Decision_Condition or its subclasses. Examples of non-deterministic decision steps are shown in Figure 37 and Figure 38.

Ranking the decision options depends on the class of Decision_Condition. Each option contains a degree of preference that may be modeled differently for the different types of Decision_Conditions. The degree of preference will determine how the Decision_Conditions will be ranked. This will assist the user in choosing among the different options. All the options in one non-deterministic decision step must belong the same class so they can be ranked consistently.

4.1.2.1 Utility_Choice_Step

Utility Choice step is a subclass of the Decision Step. It represents a choice step that uses the Utility theory in deciding among several options. It contains a pointer to the decision algorithm used to evaluate the choices. This may either be a decision analysis tree or an influence diagram.

The utility choice step has the same attributes as the decision step, but adds the decision_diagram attribute. The options.condition_value of the utility choice step must be of class UtilityChoice.

4.1.2.2 Choices

There are 3 subclasses of the Choice class: RuleIn Choice, Weighted Choice, and Utility Choice.

Rule In Choice

RuleInChoices specify rule-in, rule-out, strict-rule-in and strict-rule-out criteria for each decision option. These criteria help the user choose one of the decision options.

The strict-rule-in criteria rank a choice as the best among several options. For example, when there are competing diagnoses for a disease, a pathognomonic condition would be a strict-rule-in for the disease.

A strict rule out is analogous to an absolute contraindication. For example, “allergy to penicillin” is a strict rule out for giving penicillin.

A strict-rule-out takes precedence over strict-rule-in when ranking options. If an option contains both a strict-rule-in criterion and a strict-rule-out criterion, and both evaluate to true, then that option should be the last choice.

Strict-rule-ins take precedence over rule-ins and rule-outs. The ranking of rule-ins and rule-outs is left to the user who may use his or her clinical judgment or may develop their own ranking schemes.

All the strict-rule-outs of the same choice are related to each other using the OR relationship (i.e., if there are 2 rule-ins, A and B, then they are equivalent to a single rule-in stating A OR B). Similarly, all the strict-rule-ins of the same choice are related to each other using the OR relationship

Examples of RuleIn Choices are shown in Figure 37, Figure 39, and Figure 40.

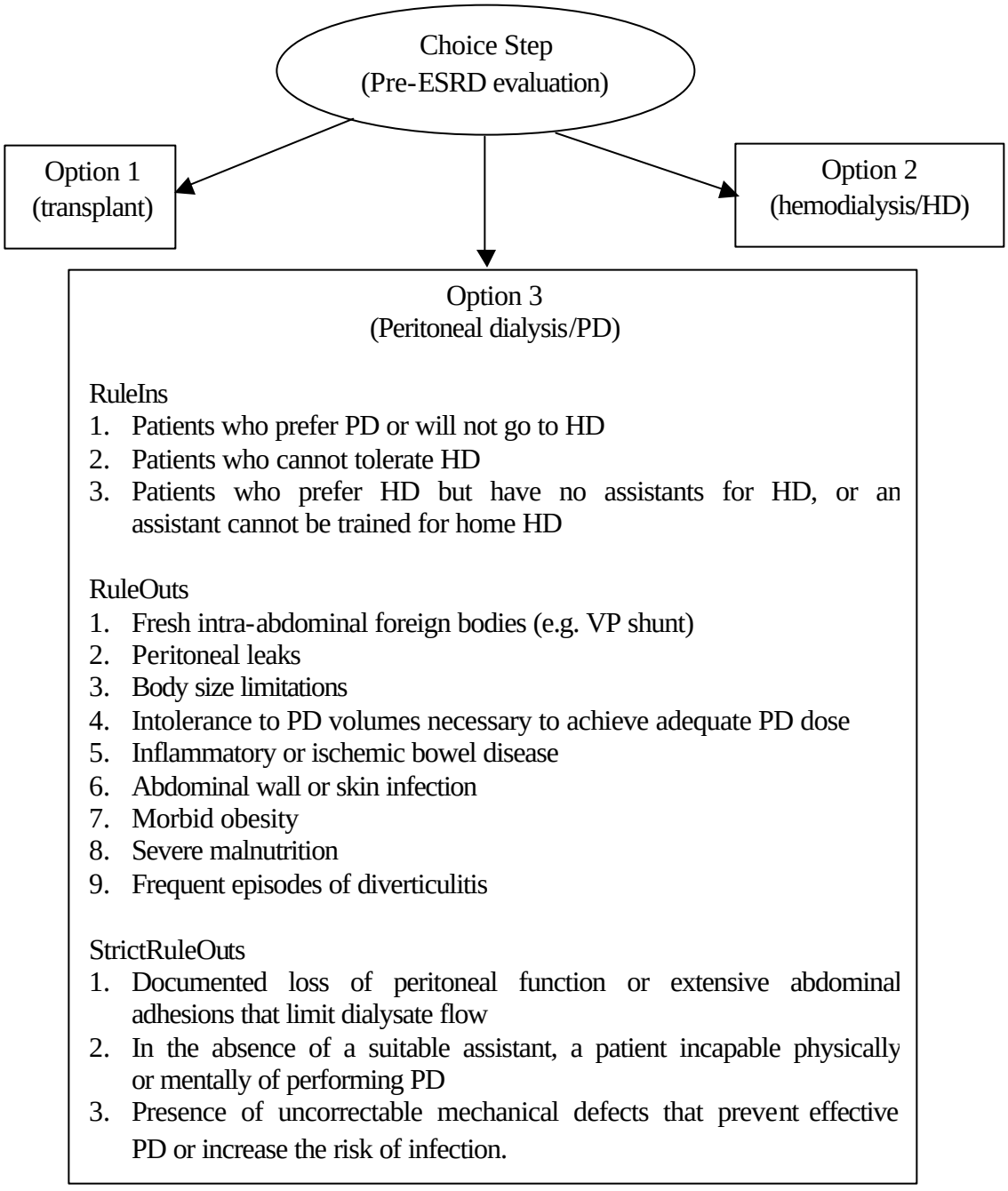


Figure 39. RuleInChoice in pre end-stage renal disease (ESRD) Evaluation. The strict-rule-in for transplant would be availability of a donor kidney. That automatically puts it as first choice.[15]

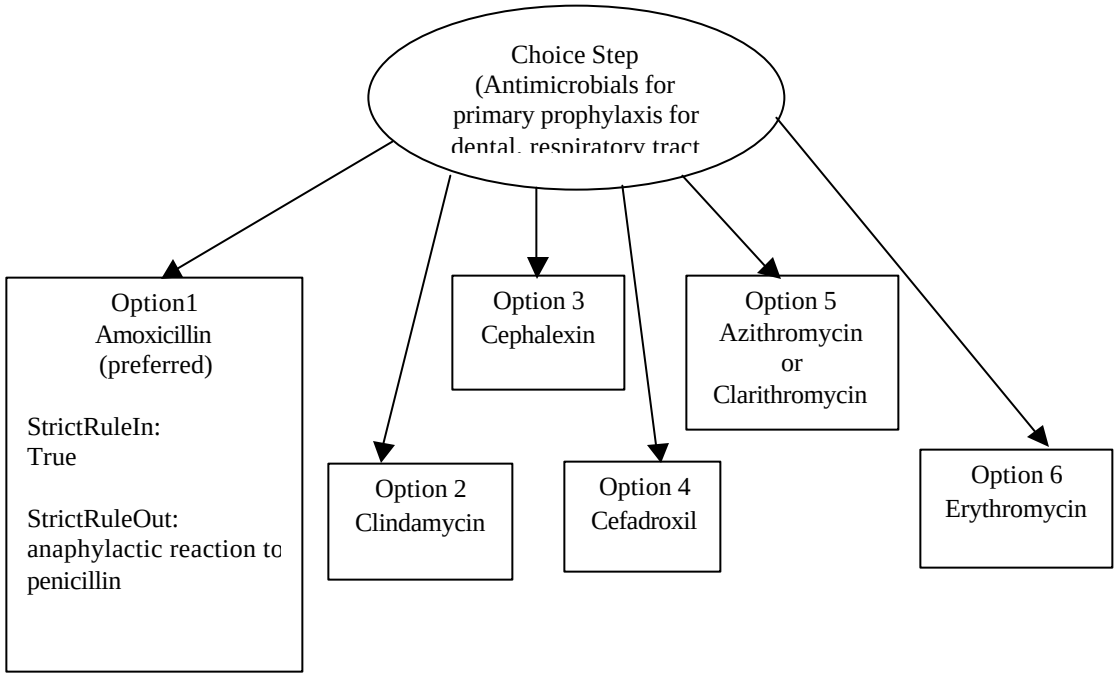


Figure 40. RuleInChoice in the decision on of antimicrobials for primary prophylaxis for dental, respiratory tract or esophageal procedures.[16]

4.1.2.3 Weighted Choice

WeightedChoices contain an array of criteria, each associated with a weight. The weighted criteria for each of the options will determine how an option will be ranked amongst the choices presented to a user at run-time. The sum of the weights for each criterion in a choice has to equal 1. The higher the value of a choice (from 0 to 1), the higher its rank. If a criterion is false or unknown, it is counted as 0 or assessed as a criteria not being met.

4.1.2.4 Utility Choice

A utility choice represents a node in a decision analysis tree or an influence diagram.

4.2 Specifying decision criteria

Criteria are expressed using three_valued_criterion_expressions that are written in a superset of Arden Syntax, called GLIF_Arden (see Section 3.8). The data items that are referenced by the criteria are specified in the medical ontology of GLIF (see Section 2.4).

Suppose that we want to specify the decision criterion: Age > 30 year

The criterion is specified as: (now – Date_Of_Birth) > 30 year

Where:

1. *now* is a special time operator that returns the current time
2. Date_Of_Birth is a primitive data value retrieved from an EPR
3. "30 year" is a literal data item that matches the type of Date_Of_Birth

4.3 *Defining patient data*

In the above example, PNDS is a term that is defined in the medical ontology. We will show how this patient data item is defined in the 3-layered ontology.

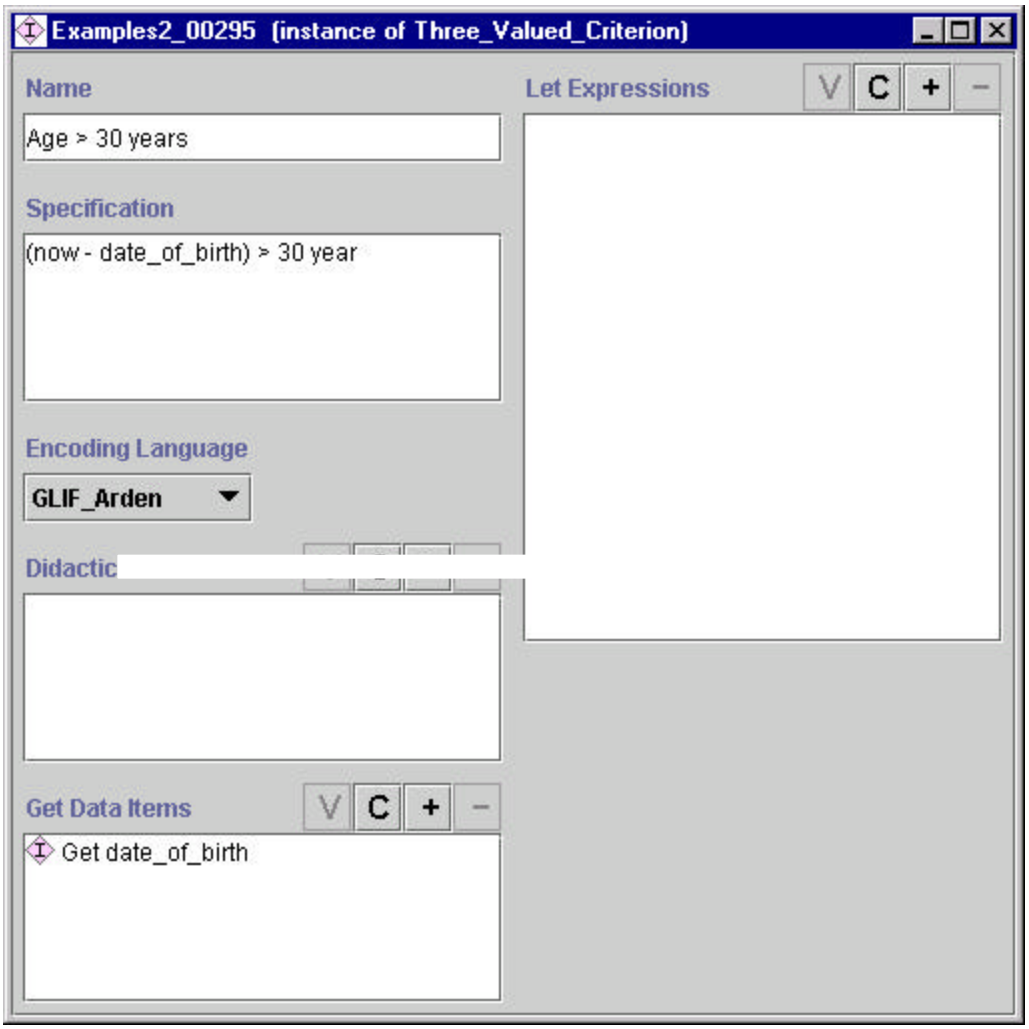


Figure 41. The criterion "Age > 30 year"

Figure 41 shows how the criterion "Age > 30 year" is modeled in GLIF. The primitive data value date_of_birth is retrieved from the EMR using the Get_Data Action specification. Date_of_birth will be defined by the Get_Data action specification to retrieve the data from the Date_Of_Birth patient Data_Item.

Data items are used when specifying decision criteria, as shown in Section 4.2. Patient data items can have quite complex structures depending on the RIM. This can introduce significant

complexity into expression evaluation. For instance, "Latest cough" is difficult to compute because cough, as an observation, has more than one associated time stamps. Many attributes in a RIM such as USAM serve for documentation and retrieval purposes only. So we intend to encourage users to use the Get_Data_Action action specification (see Section 5.3.4) to retrieve appropriates data value(s), assign them to primitive data items and only employ primitive data in computation.

Another example is given in Figure 42 and Figure 43, where the criterion

PostNasal_Drip is_in Symptom

is specified using the literal data item PostNasal_Drip and the variable data item Symptom.

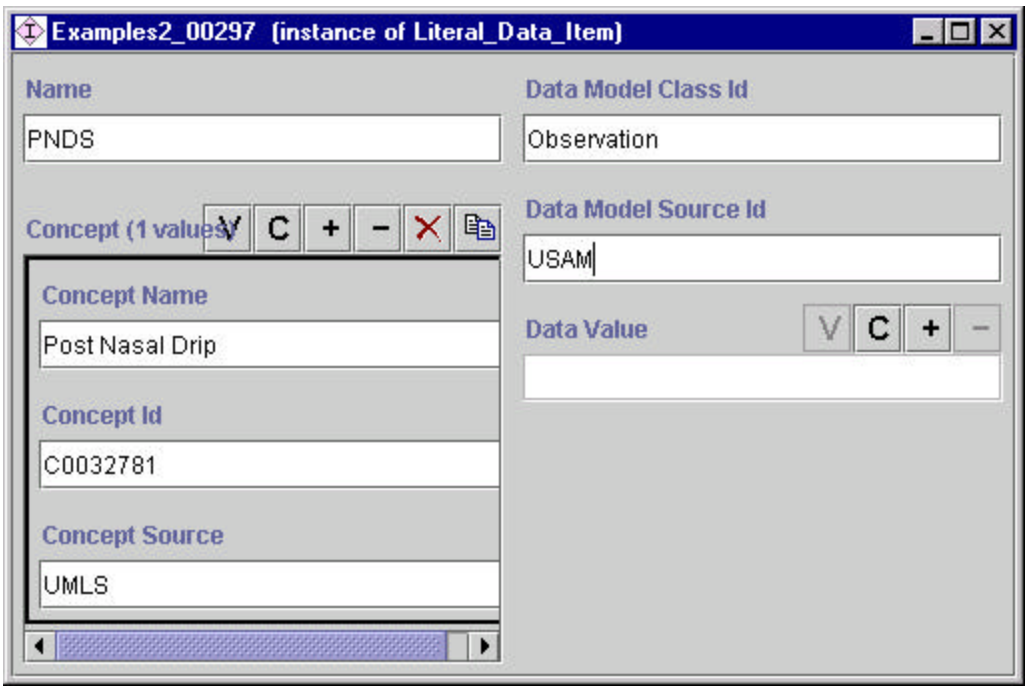


Figure 42. The PostNasal_Drip literal data item

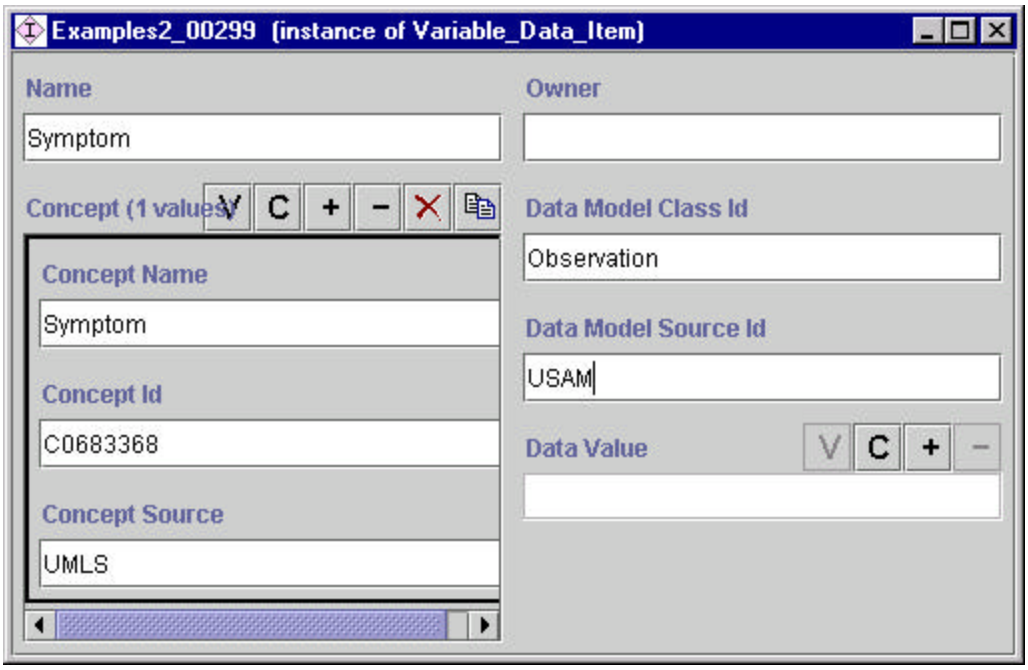


Figure 43. The Symptom variable data item

5. Describing actions

5.1 *Specifying the action and parameters*

See section 3.4(Action Steps).

5.2 *Iterative actions (and decisions)*

The Iteration_Specification class specifies information regarding the loop structure of the iteration. Only action steps and decision steps may be iterated. The action- and decision steps that reference the Iteration_Specification, are iterated until the abort condition or stopping condition criteria hold. The iterations are carried out at a certain frequency, which is expressed by an Iteration_Expression. The Iteration_Expression class is shown in Figure 44.

Different types of iteration expressions are possible. These are: frequency expression, every expression, times expression, and discrete temporal expression table.

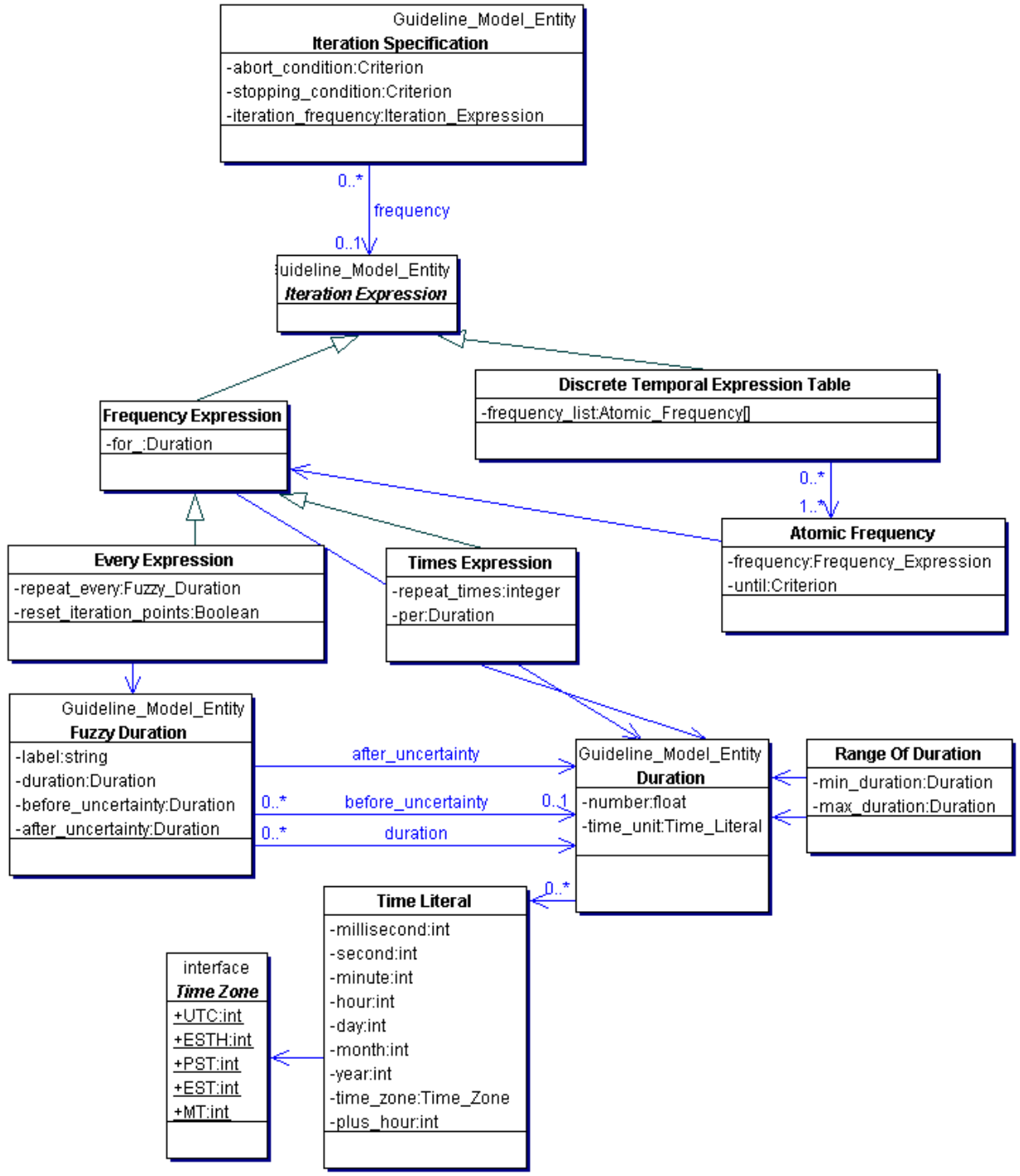


Figure 44. The Iteration Expression class hierarchy

There are two types of Frequency_Expressions: Times_Expression and Every_Expression. Both of them define the frequency at which an iteration should occur and its duration. The duration is specified by the *for* attribute of the Frequency_Expression class.

A *times expression* specifies that something should occur a specified number of times (the *repeat_times* attribute) within a specified interval (the *per* attribute) (e.g., "3 times a day;"). At execution time, this class should be mapped, or refined to a tight temporal expression.

An *every expression* specifies that something should occur every *fuzzy duration*. A *fuzzy duration* is a duration that has an associated before and after uncertainty period. Any time point within (duration-before_uncertainty, duration+after_uncertainty) is considered to be within the limits of the fuzzy duration. For example, for a duration of 4 hours with a before uncertainty of ½ hour and an after uncertainty of 1 hour, represents a fuzzy duration interval of 3½ - 5. The fuzzy duration also has offsets. These are used in conjunction with the every expression (see below). Any time point within (duration-before_offset, duration+after_offset) is considered to be within the limits of the fuzzy duration, but requires resetting of the iteration points used by every expressions. Examples of fuzzy duration expressions are: "4 hours with window –30 minutes, + 1 hour;" and " 5 hours with offset –1 hour, + 1 hour;"

An *every expression* specifies whether or not the iteration points should be reset in cases where the iteration did not occur exactly on the fuzzy duration's duration, but within the interval (duration-before_uncertainty, duration+after_uncertainty). This is specified by the reset_iteration_points Boolean. Taking birth control pills is an example of constant iteration points, which should not be reset. A birth control pill needs to be taken every 24h, but there are before and after windows of 12h. So even if the pill was not taken at the regular time, it can still be taken up to 12h later. If the patient remembered missing a dose more than 12 hours after the normal duration of the every expression, then the dose should be skipped. Iteration points are not reset. So if the pill was first taken at 9pm it should be continued at approximately that time, every 24 h. The every expression for this example is shown in part (a) of Figure 45.

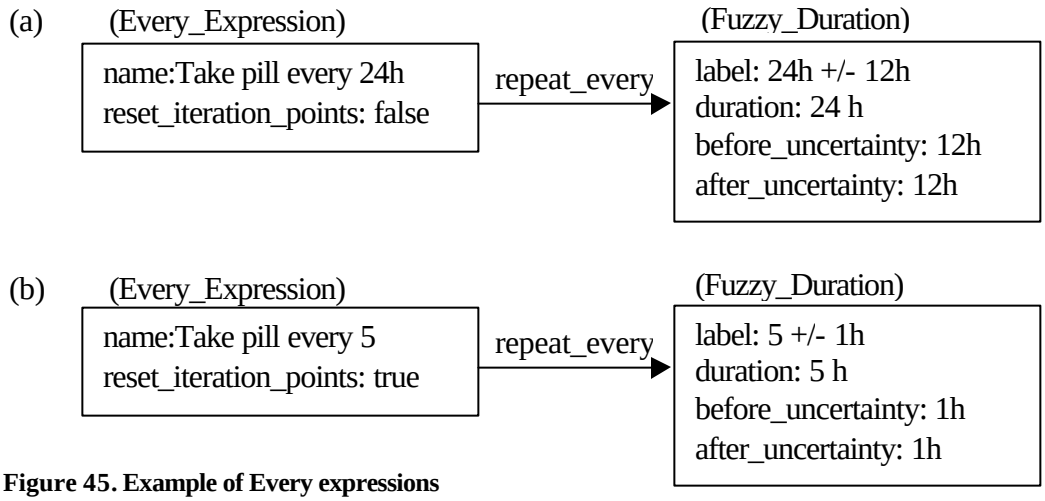


Figure 45. Example of Every expressions

Aspirin represents an example of resetting iteration points. Aspirin should be taken every 4-6 hours. Using an every_expression, the fuzzy duration is specified to have duration of 5 h with before and after uncertainty of 1 h. So if the first aspirin tablet was taken at 7am, the following doses are tentatively scheduled for noon, 5pm, 10 pm, etc. Suppose the second pill was taken at 1 pm. Then the patient should be able to take the next dose at 7pm. This will be enabled if after the

second pill was taken, the iteration points were reset to 7 am, 1 pm, 6 pm, 11 pm, etc. The every expression for this example is shown in part (b) of Figure 45.

A *discrete temporal expression table* is another type of Iteration_Expression. It specifies a list of pairs of frequencies and durations, called *atomic frequencies*. For example, see patient every 5 weeks for 5 months, then, every 2 weeks for 1 month, and then every week for 1 month. The order of rows is important. The rows are to be executed from the top of the table to its bottom.

Executing an action at irregular intervals can be either modeled as iteration or not. For example, a visit schedule that says: "visit every 5 weeks for 5 months and then every 2 weeks for X months" may be modeled as an iterative action, but an immunization schedule that requires giving an immunization at times t; t+ 1 month; t+2 months would better be modeled as an action (Immunization) that is not iterative, but instead is event triggered. The triggering events would be temporal: 1 month after the first dose; 2 months after the first dose. Modeling the immunization schedule as iteration is possible but is not elegant. It can be done by specifying that a shot should be given every 1 month for 1 month, then every 2 months for 2 months, etc.

Another consideration involves maximum doses. Some medications need to be taken at 1-2 pills every 4-5 hours, but no more than 8 pills within 24 hours. The iteration is still every 4-5 hours, but the dose per iteration is dependent on previous doses and may be equal to zero.

Examples of an iteration specification are shown below.

1. Iterate 3 times a day for 10 days

   Iteration_specification.frequency == 3 times a day for 10 days (Times Expression)

2. Iterate 3 times a day for 30 times

   Iteration_Specification.frequency == 3 times a day (Times Expression)

   Iteration_Specification.stopping_condition == 30 times

2. Iterate (see the doctor) every 5 weeks until week 31 after conception, then every 2 weeks for 4 weeks, then every week until week 40.

   Iteration_Specification.abort_condition == end of pregnancy

   Iteration_Specification.stopping_condition == 40 weeks after conception

   Iteration_Specification.frequency == every 5 weeks until 31 weeks after conception;
   every 2 weeks until 35 weeks after conception;
   every 1 week until 40 weeks after conception;
   (Discrete temporal Expression Table)

5.3 *Action Specifications*

The action specification model includes two types of actions: (1) guideline-flow-relevant actions, such as calling a sub-guideline, or computing values for data; and (2) clinically relevant actions, such as making recommendations. Clinically relevant actions reference the medical ontology for representations of clinical concepts such as prescriptions, laboratory test orders, or referrals.

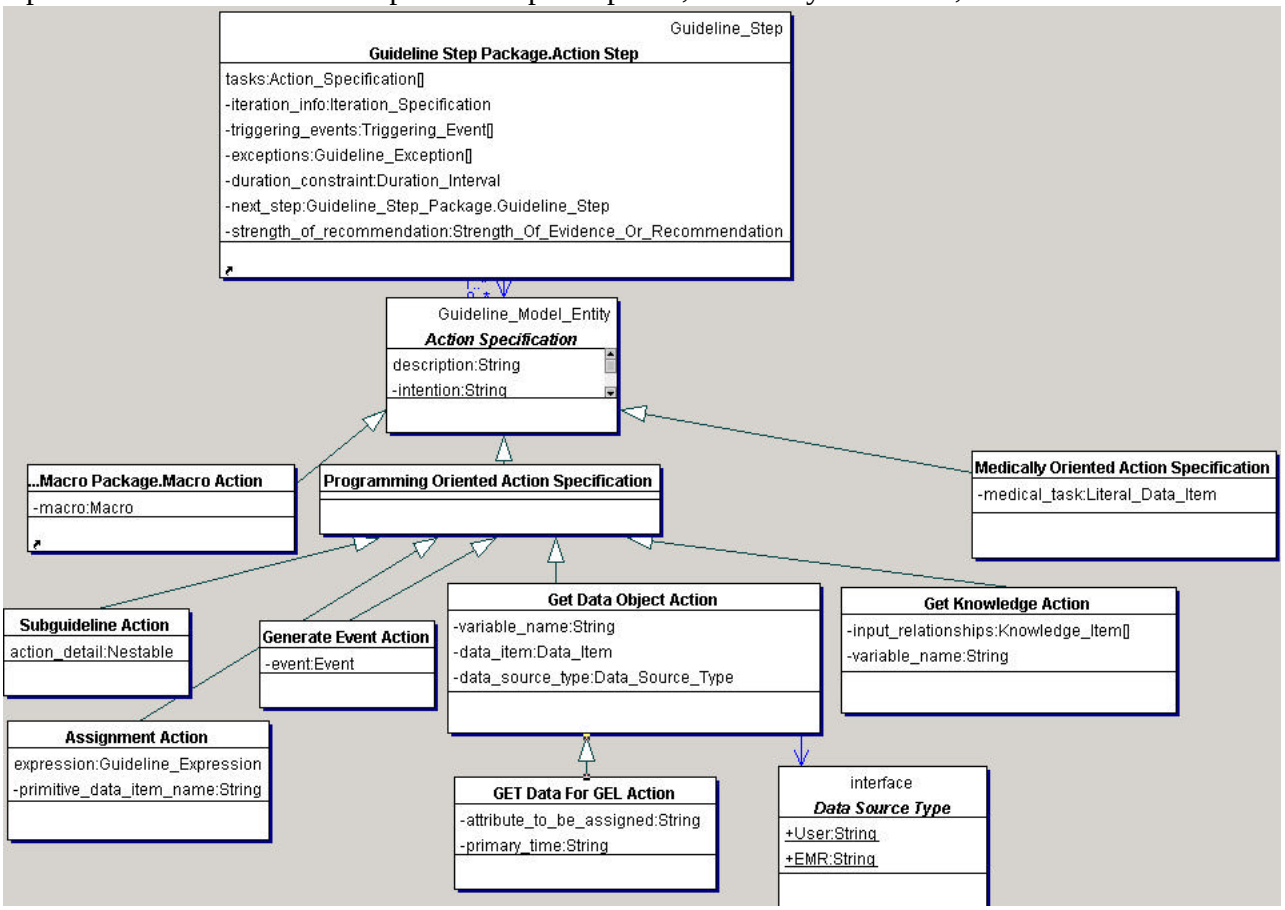


Figure 46. The action specification package

All action specification objects specify the details of a clinical action. The Action Specification class is abstract. Its subclasses are programming oriented action specifications and medically oriented action specifications.

5.3.1 Subguideline Action

This is a programming-oriented action specification, that contains the details of a high-level action in the form of a (sub)guideline or a macro.

5.3.2 Assignment Action

This is a programming-oriented action specification. The Assignment_Action class is used to create/instantiate a primitive data item. This data item is assigned the value resulting from the evaluation of the expression.

5.3.3 Generate Event Action

This is a programming oriented action specification. The Generate_Event_Action class is used to create an event, such as "a data item was written to the EPR".

5.3.4 Get Data Object Action

This is a programming oriented action specification. It is used to explicitly obtain the value of a data item from the EMR or from a user and store it in a variable.

5.3.5 Get Data For GEL Action

GLIF3's Get_Data_For_GEL_Action retrieves patient data from the EMR as HL7 RIM objects and transforms them to query result data types. It allows a mapping to be specified from GLIF3's default data model, the HL7 RIM, to GEL's data model. A guideline author can use Get_Data_For_GEL_Action to specify that an attribute of a complex RIM class is the source of data values for the query result, and that values of another attribute serve as the primary time in the query result. Thus, the query result is a list of value and primary time pairs similar to Arden's query result data type. However, the value attribute in GEL's query result holds a simple or a complex GEL type. Get_Data_For_GEL_Action specifies which data item from the EMR will serve as the source of data, and which attribute will be selected from the data item. In this way, specific attributes of the data item can serve as the source of the data, rather than the entire data item. For example, Get_Data_Action can retrieve all instances of Medication data items that refer to ACE Inhibitor treatments (Figure 47). It can assign the value of their *data value* attribute, which is a RIM Medication object (Figure 48), to the query result's "value" attribute, and assign the end time of each Medication treatment (*critical_time.high*) to the "primary time" attribute of the query result elements.

(**Instance of Get_Data_Action**)

data_item: ACEI_Item
attribute_to_be_assigned: data_value
variable_name: ACEI
primary_time: data_value.critical_time.high

(Instance of Query_Result)

value: (Medication instance)	value: (Medication instance) …
primary_time: 2002-01-08	primary_time: 1999-03-02

Figure 47. The Get_Data_Action and its query result that holds ACEI Medication objects data values and their primary times. In this example, the latest query result element has the primary_time 2002-01-08.

(**Instance of Variable_Data_Item**)
{name: ACEI_Item
concept: {(**instance of Concept**)
concept_name: ACEI;
concept_id: C0003015;
concept_source: UMLS}
data_model_class_id: Medication
data_model_source_id: HL7-RIM
data_value: {(**instance of Medication**)
service_cd: ACEI concept;
mood_cd: event;
critical_time: {low: null;
high: null;}…} }

Figure 48. A variable data item that defines ACE Inhibitor treatment. Attribute names are on the left, followed by their values. Complex values are in curly braces. The ACEI data specify the appropriate UMLS code and HL7 RIM class (Medication). The figure shows two attributes of Medication. Other attributes include dosage_quantity, rate_quantity, and route_code.

5.3.6 Medically Oriented Action

The Medically_Oriented_Action class is used to define an action that refers to a medical term. This class is used to represent a typical guideline recommendation.

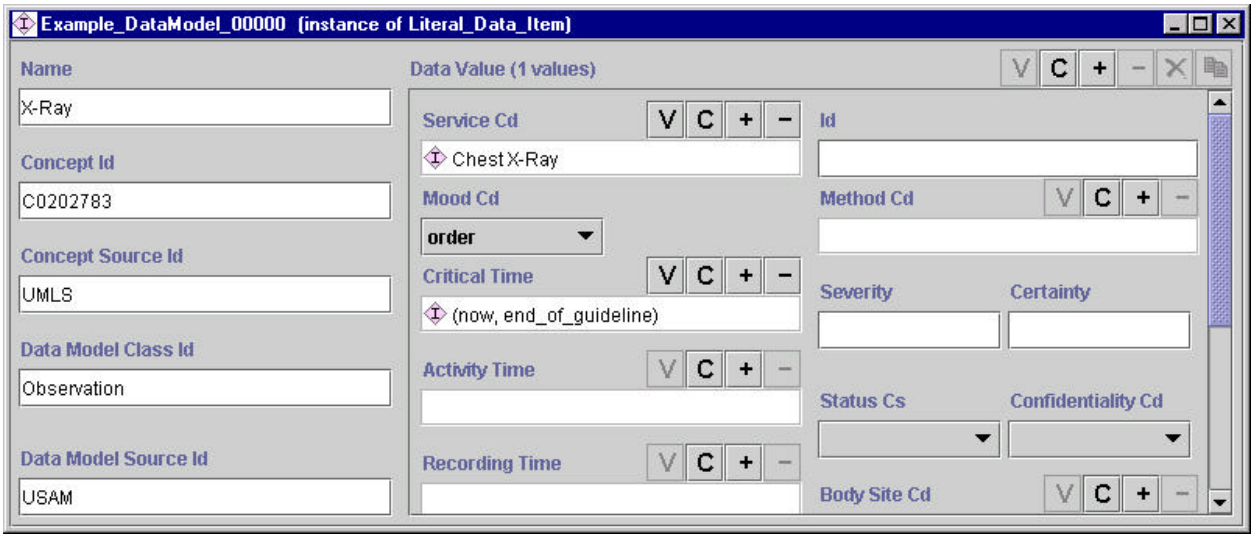


Figure 49. A medically oriented action specification that orders a chest X-ray

6. Patient States

A Patient_State_Step is a guideline step (a node in the flowchart) that is used for two purposes. One purpose is to serve as a label that describes a patient state that is achieved by previous steps. This way, a guideline may be viewed as a state transition graph, where states are scenarios, or patient states, and transitions between these states are the networks of guideline steps (excluding patient state steps) that occur between two patient state steps. The other purpose of a patient state step is an entry point to the guideline (e.g., patient came back to the clinic at clinical state A).

A patient state step has a criterion that describes the state of the patient who is at that patient state. If there is a criterion that refers to a generalization (e.g., "state is not well") it also applies to specializations of that class (e.g., "state is fever"). The hierarchy of concepts is defined in the medical ontology, as shown in Figure 3.

A patient state step is followed by a guideline step.

An example of a Patient_State_Step is given in Figure 51 and Figure 52.

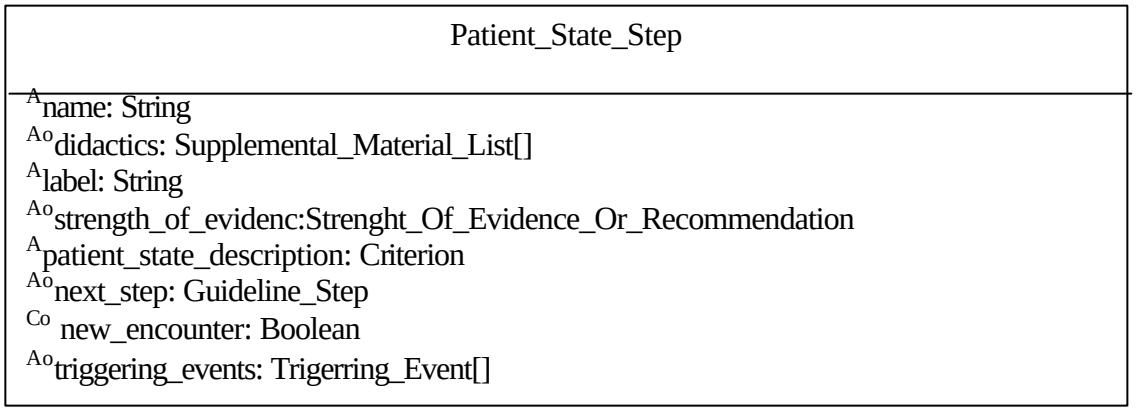


Figure 50. The Patient State Step class. The superscript A, and C indicate the level of specification that the attribute belongs to, while o indicates an optional attribute value.

When a patient arrives at a clinic, his current state is compared to the last patient state that was recorded for him. If he is not at that state, then the patient state steps that represent new encounters are searched. These can be determined either by an *implementation-level* attribute called "new_encounter" of type Boolean, which characterizes a patient state step or by looking at patient state steps whose next-step is triggered by an event of type "new patient encounter".

It is important to acknowledge the fact that a patient might not follow the guideline precisely, and that he/she may also be treated outside the regular clinic.

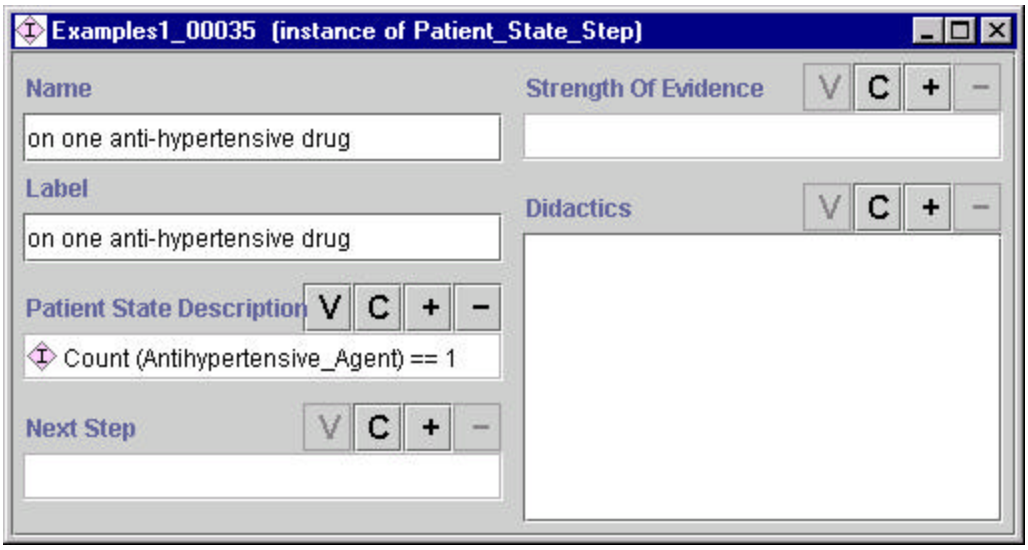


Figure 51. An example of a patient state step

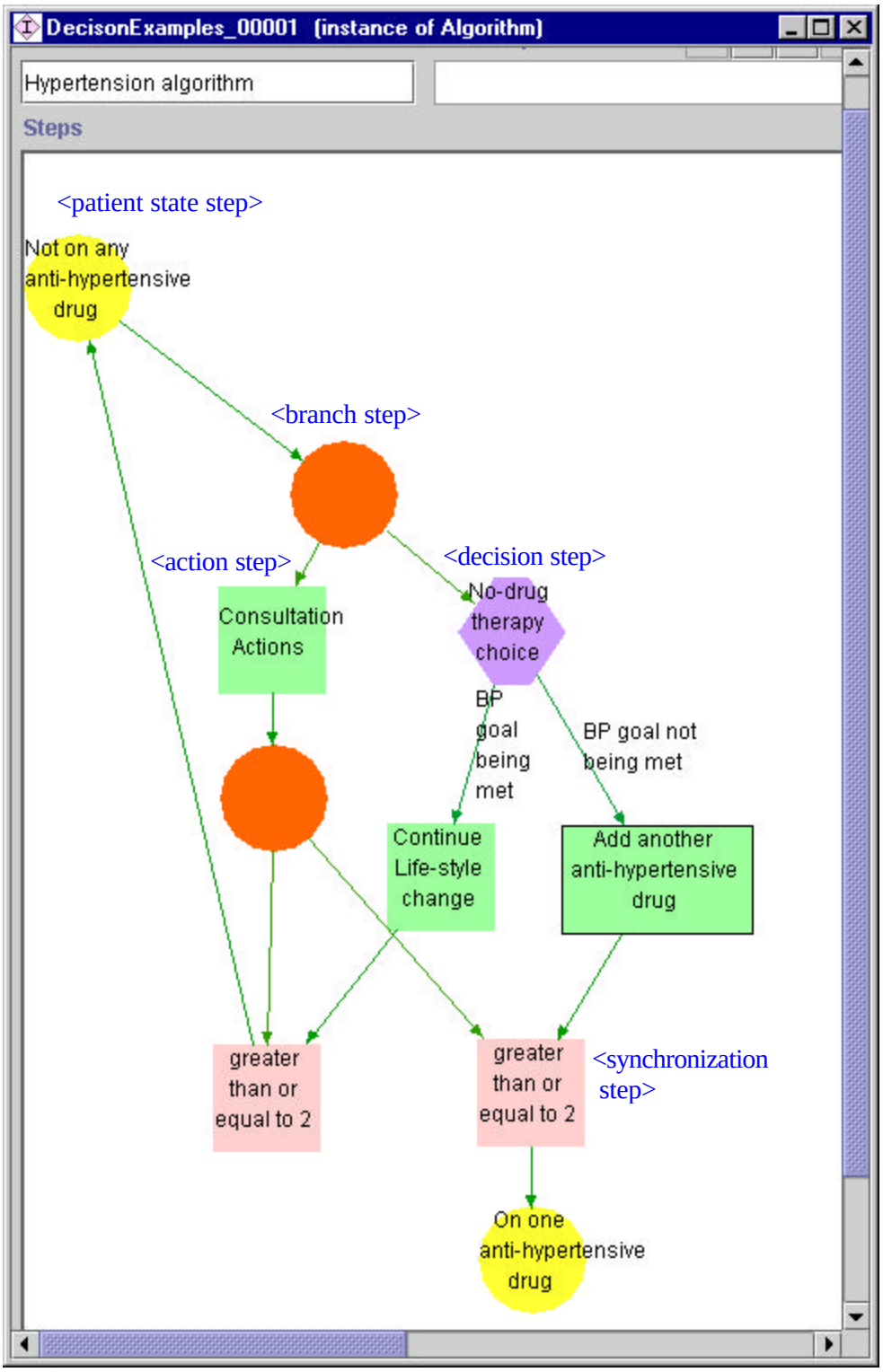


Figure 52. A hypertension guideline showing transitions between two patient state steps

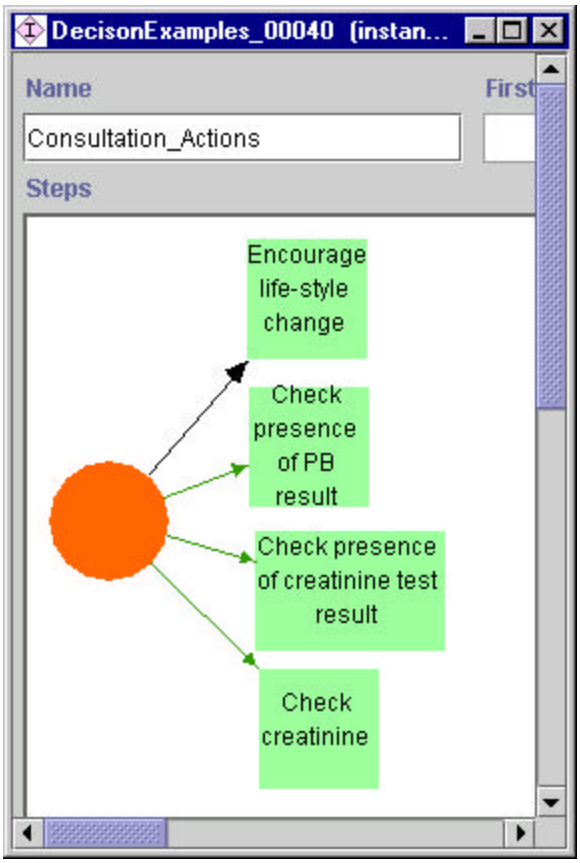


Figure 53. The consultation actions shown in this figure are executed in parallel. This is a zoom-in view of the consultation action shown in Figure 52.

7. Parallel paths in a guideline

7.1 *Branching to multiple paths*

See Section 3.6 (Branch Steps).

7.2 *Synchronizing from multiple paths*

See Section 3.7 (Synchronization Steps).

8. Dealing with complex guidelines

The Nestable class is a superclass of Guideline and Macro. It is an abstract class. Both Guideline and Macro are guideline model entities that can be nested. Nesting allows grouping of parts of a guideline into modular units (subguidelines or macros). This enables partitioning the guideline parts into manageable sized units that can be more easily comprehended. These modular units can also be reused by other guidelines.

The details of action and decision steps of a guideline can be shown in a different guideline that serves as a subguideline of the first guideline. The subguidelines can recursively contain other subguidelines to specify even more details of actions or decisions.
Macros can be used to represent patterns of domain level concepts in a single encapsulated object. This object can be then be mapped to a guideline object containing underlying GLIF steps (i.e., not containing macros). In this way, macros enable declarative specification of a procedural pattern that is realized by a set of primitive GLIF steps.

Nesting is very useful for managing the complexity of guidelines. Nesting enables looking at a guideline from a top-level view and then zooming into/out of some of its parts. Nesting is also useful in representing a guideline in the context of other guidelines. Since nesting allows grouping of parts of a guideline into a single unit, this is a mechanism that can allow model extensibility and reuse of part of a guideline (defining macros), or adaptation of a guideline to a specific institution by replacing specifications for parts of a guideline (i.e., replacing a goal with a procedure).

8.1 *Nesting decisions*

Decision are nested by specifying a subguideline in the decision_detail attribute of a decision step. This subguideline is executed before the decision criterion for that step is evaluated. The subguideline would modify or create new variables and assign them values. The use of these variables in the decision criteria makes the decision nested.

The connector represent the

Decision_Option.rule-in:

Patient_Cough_ACEI_Smoking_state == Cough_not_due_to_smoking_and_not_ACEI

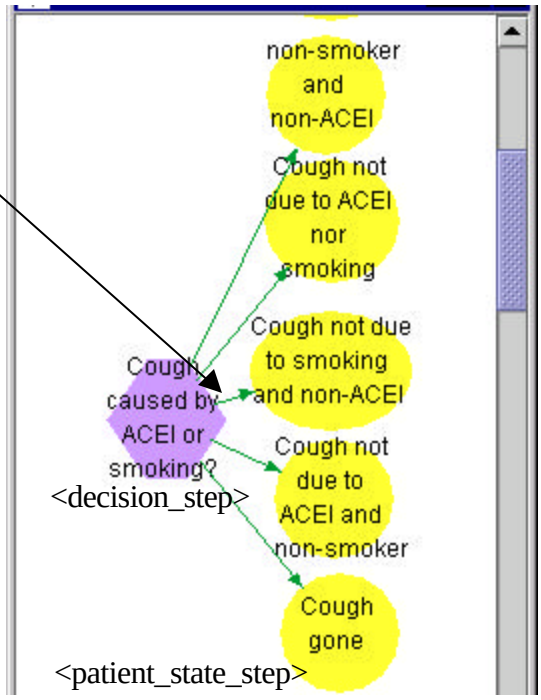


Figure 54. A top-level view of a nested decision step (ACEI=ACE Inhibitor).

A zoom-into view of the Decision Step shows:

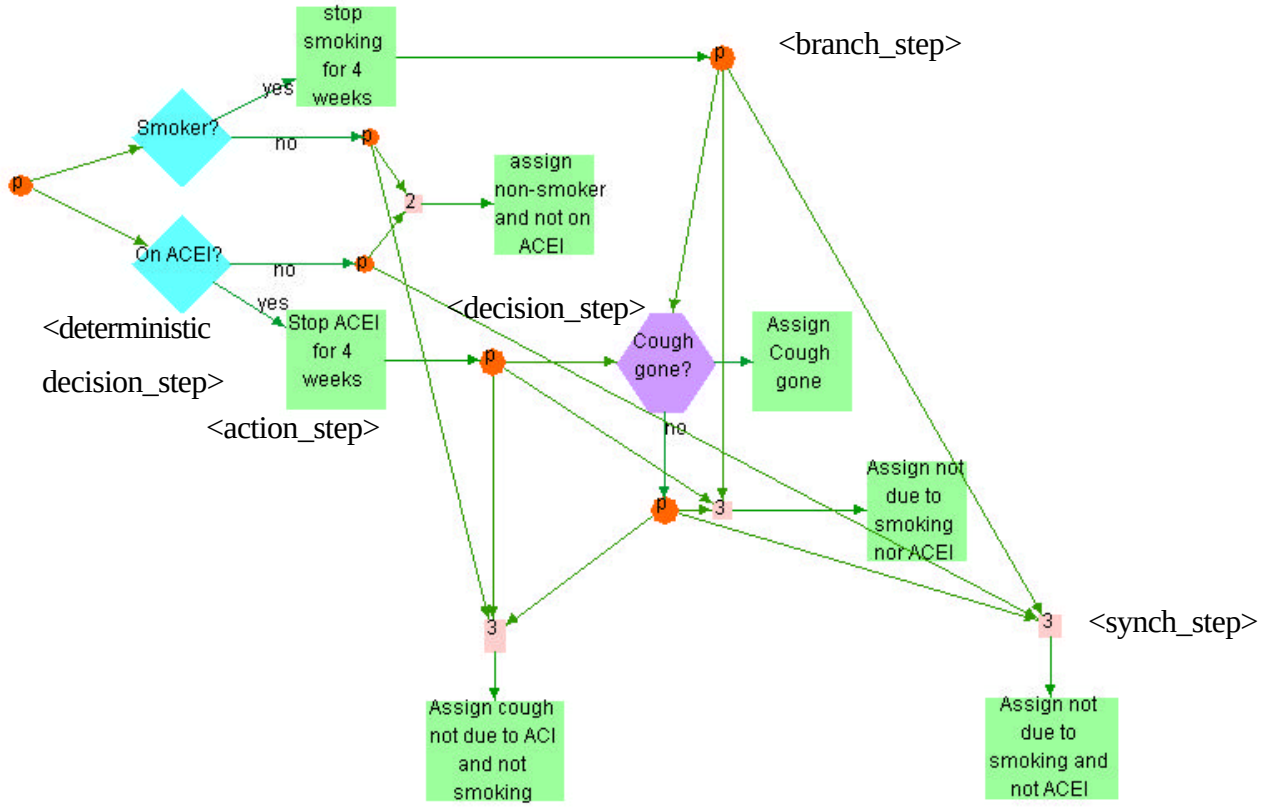


Figure 55. This is the detailed view of the decision step, shown in the previous figure. This subguideline determines state of the patient in terms of his cough, smoking, and ACEI use. The leaf steps of this subguideline assign the cough_smoking_ACEI value to a new data item named Patient_Cough_ACEI_Smoking_State using the Assignment_Action. The data item Patient_Cough_ACEI_Smoking_State that is created by the subguideline is used by the main decision step in its rule-in criterion. The value is used by the decision option's destination to select the next step of the outer guideline.

8.2 *Nesting actions*

Action Steps are nested by including a Subguideline_Action type of task in the step. The Subguideline_Action task has a subguideline attribute that contains the nested subguideline.

Figure 56 shows an example of nesting an action step, for complexity management purposes, while Figure 57 shows an example of nesting which is used for adjusting a local procedure.

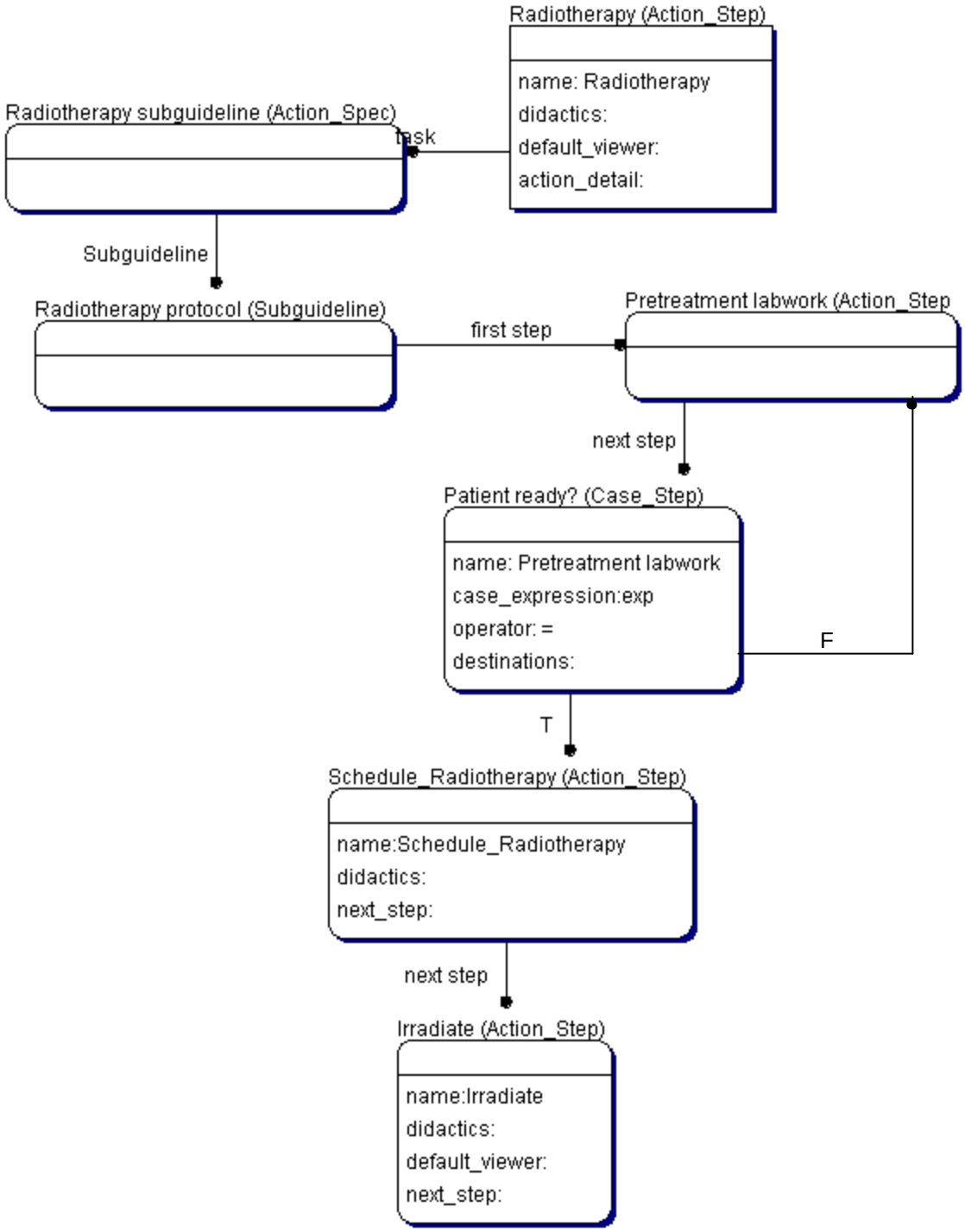


Figure 56. Nesting of an action step, for complexity management purposes.

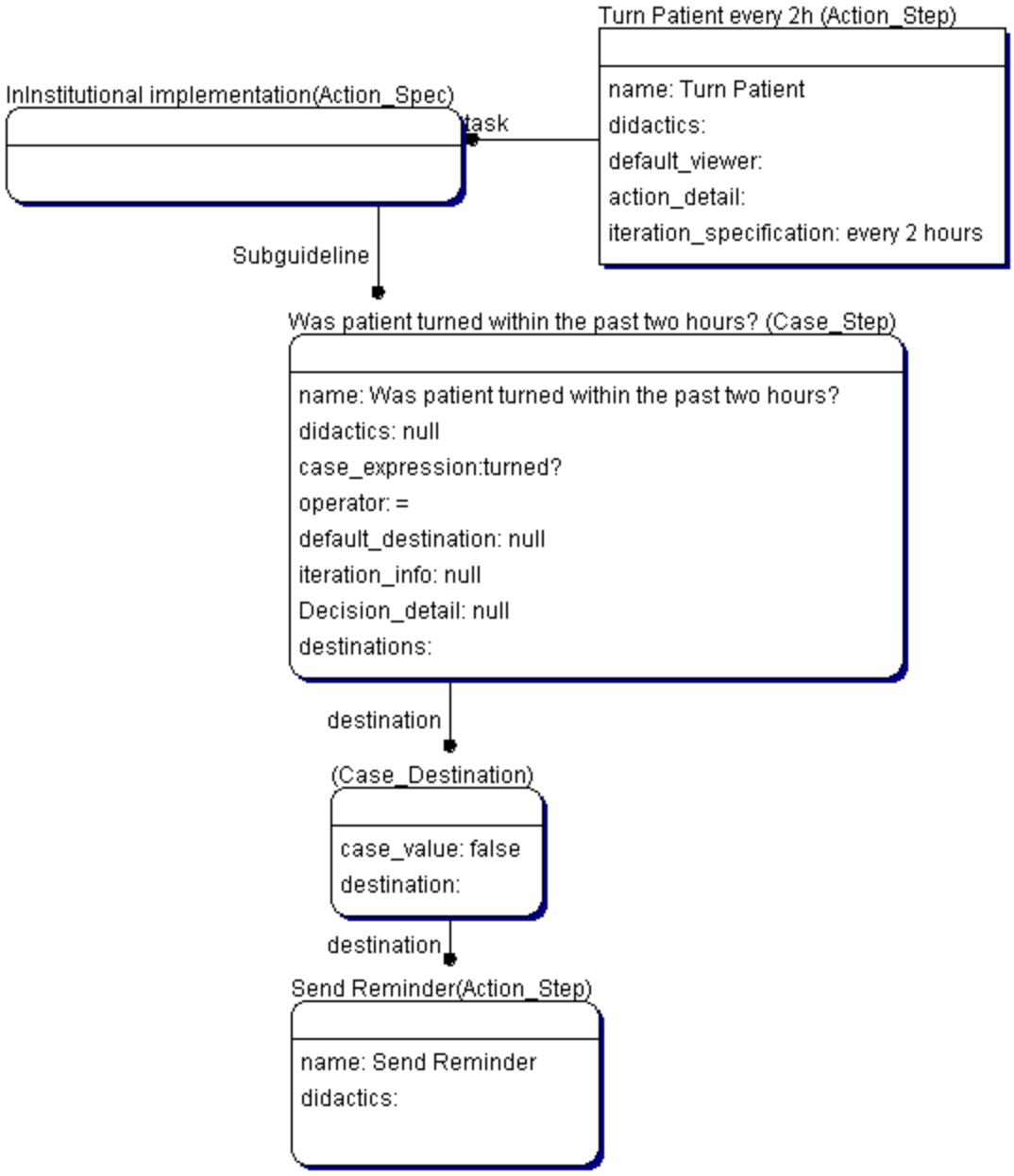


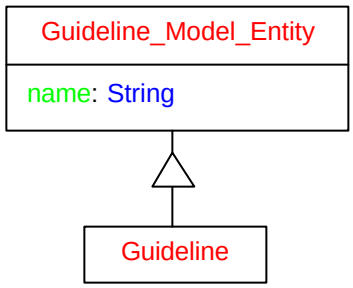
Figure 57. Nesting of an action step, for adjusting a local procedure.

9. RDF-based Syntax for GLIF

The Resource Description Framework (RDF) is an infrastructure that enables the encoding, exchange and reuse of structured metadata. It is developed under the auspices of W3C. RDF has an explicit model for expressing object semantics (objects, attributes). RDF uses XML (eXtensible Markup Language) as a common syntax for the exchange and processing of. Metadata.

The data structure (metadata) definitions of GLIF's object model are given by an RDF Schema. RDF can be used as a format to encode instances that conform to the RDF Schema. Figure 58(a) shows a class diagram that describes part of the GLIF model for the guideline class. Part (b) of the figure shows the corresponding RDF schema. Part (c) of the figure shows an example of an rdf guideline instance that conforms to the RDF schema shown in (b).

(a)



(b)

```
<s:Class rdf:about="&a;Guideline">
<s:subClassOf rdf:resource="&a;Guideline_Model_Entity"/>
</s:Class>
<s:Property rdf:about="&a;name">
        <s:domain rdf:resource="&a;Guideline_Model_Entity"/>
        <s:range rdf:resource="&s;Literal"/
</s:Property>
```

(c)

```
<a:Guideline rdf:about="&a;Cough_INSTANCE_00068">
        <a:name>Management of Chronic Cough in Immunocompetent Adults</a:name>
</a:Guideline>
```

Figure 58. A Partial RDF schema for the GLIF model

10. Acknowledgements

We would like to thanks very much the following people who were very helpful in giving us comments about this document and about the GLIF language that led us to changing this document and some of the constructs of the GLFI language.

Dipl.-Ing. Florian Rissner,
Technical University of Ilmenau,
Germany,
florian.rissner@gmx.net

Carol Broverman PhD
Director, Clinical Informatics
Fast Track Systems
San Mateo, CA
cbroverman@fast-track.com

Micael Kahn MD, PhD
Vice President Medical Informatics
Fast Track Systems
San Mateo, CA 94403
mkahn@fast-track.com

A. Appendix A

1. Macros

A macro is a special class with attributes that define information needed to instantiate a set of underlying GLIF steps. Macros can be used to represent patterns of domain-level concepts. Macro steps benefit authoring, visual understanding, and execution of guidelines.

The Macro class is an abstract class. A new type of macro is defined by creating a subclass of the macro class. The attributes of this subclass are then mapped to attributes of the underlying patterns of GLIF steps to generate the representation of the macro in GLIF steps. The schema attribute of the Macro class describes the mapping of the Macro to GLIF steps. A schema language is still being developed.

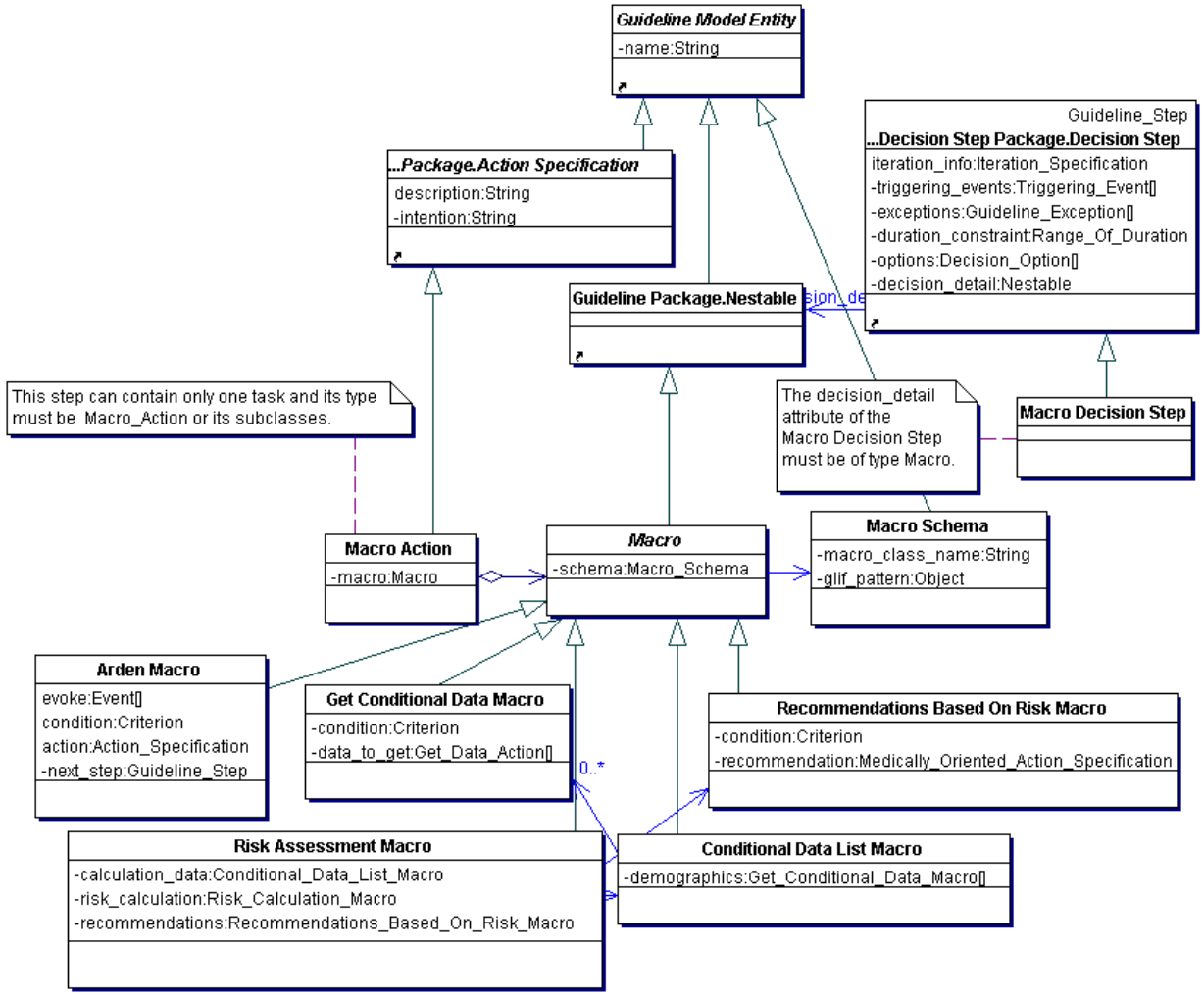


Figure 59. The Macro package

An Arden Macro, like an Arden MLM, has four sub-slots: evoke, condition, and action, that correspond to the evoke, logic and action slots of the knowledge slot of an Arden MLM, and the next_step attribute that links it to the next guideline step. An example of an Arden macro and its expansion is shown in

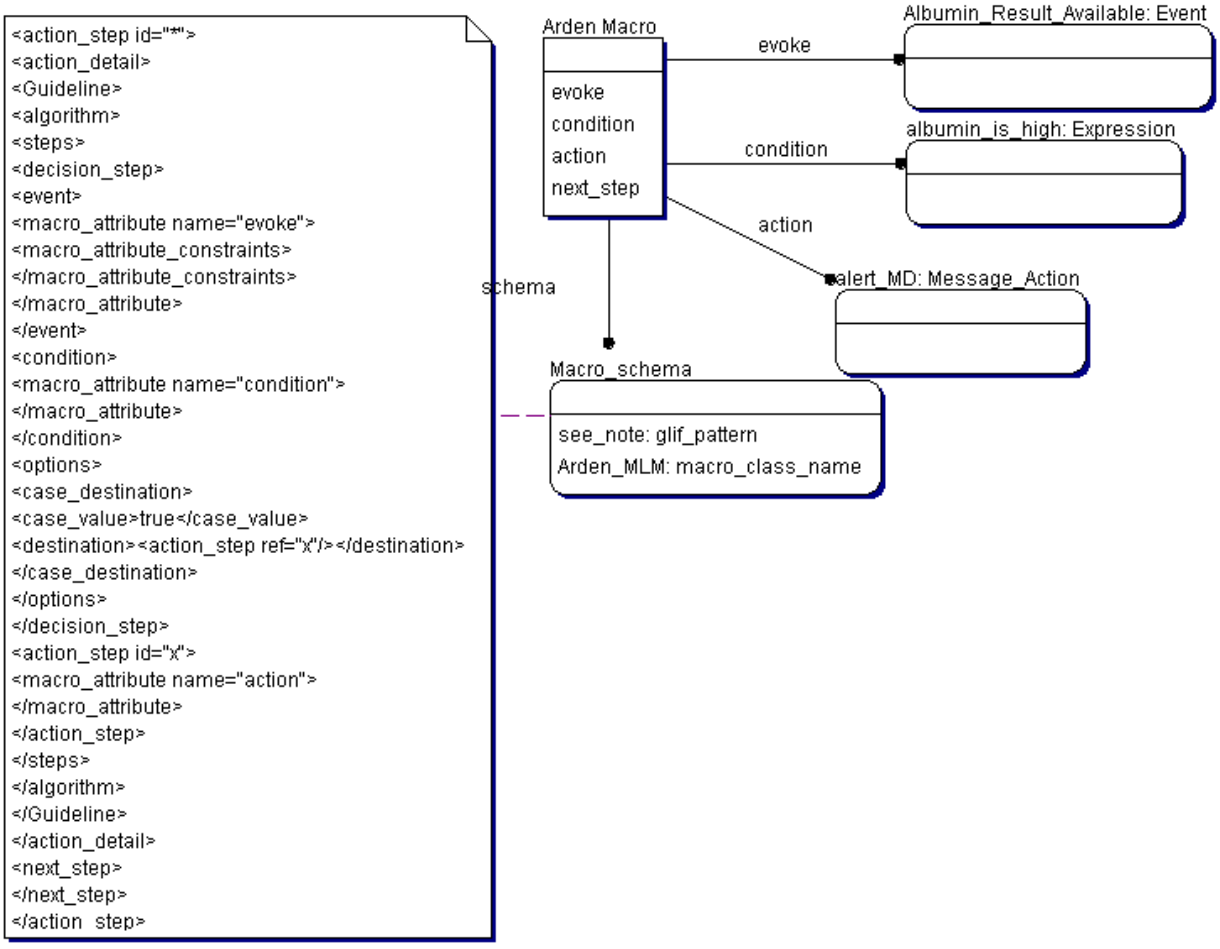


Figure 60 and Figure 61.

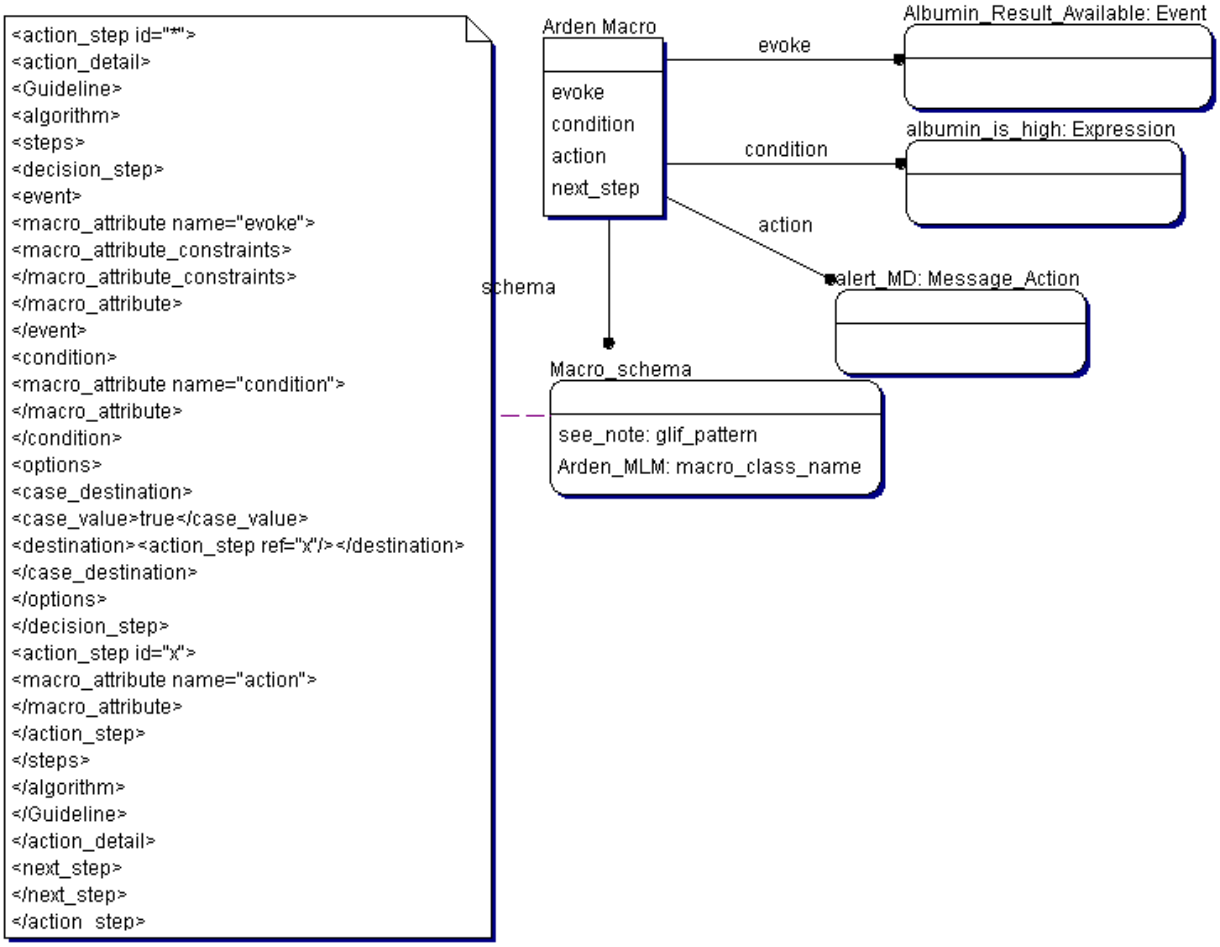


Figure 60. An example of an MLM Macro used to alert a physician if the patient has a high albumin value

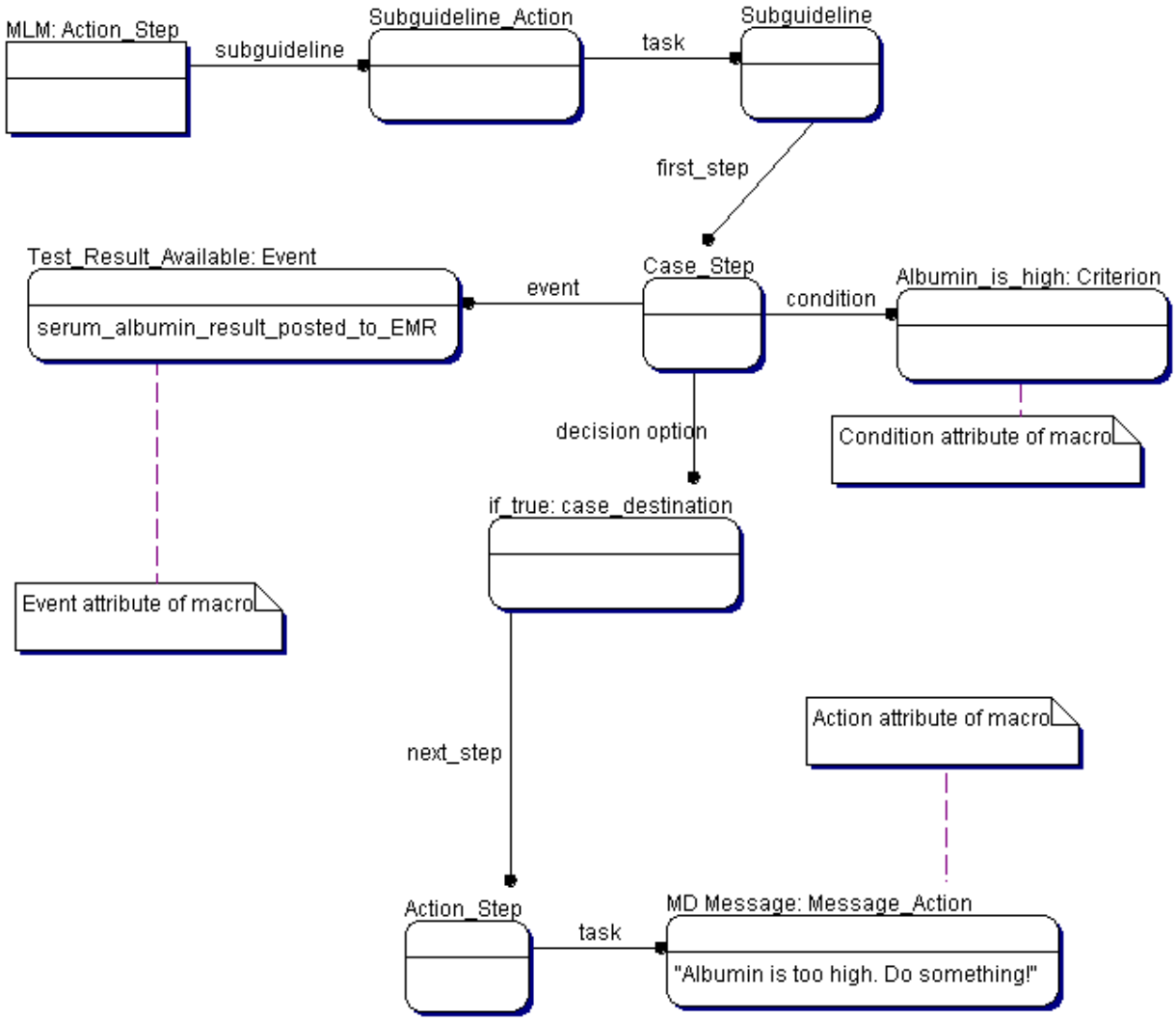


Figure 61. An expansion of the MLM macro from

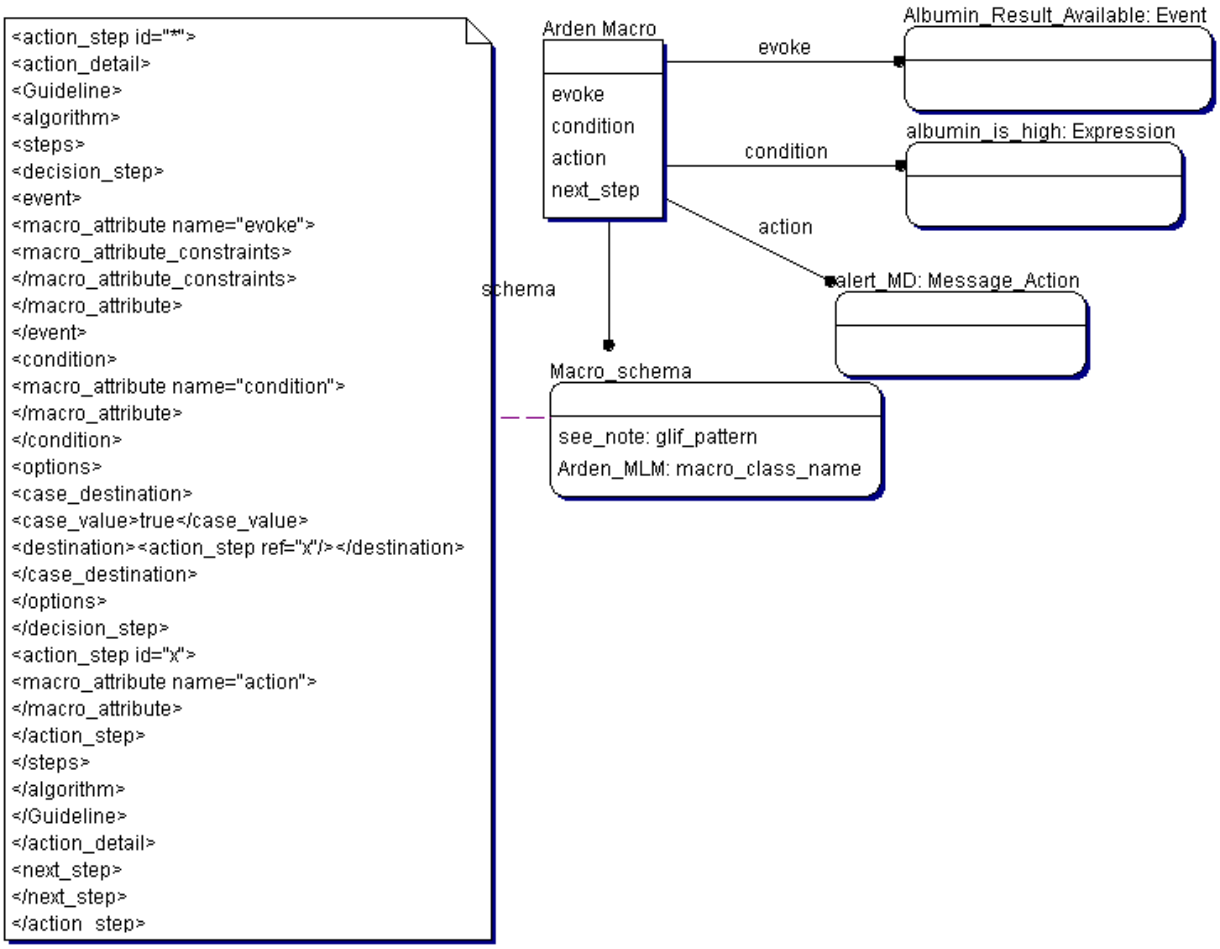


Figure 60 into primitive GLIF steps.

Risk Assessment Macro

A Risk Assessment Macro has three parts, or “steps”:

1. **Collecting patient data** - Data that is needed for calculating risk is collected through the *Conditional_Data_List_Macro*. The data may be obtained conditionally based on values of previously collected data. In the example shown in Figure 62, demographics data is obtained for all patients. Menstrual history is obtained only if the condition adult female is true. The Conditional_Data_List_Macrois modeled using an ordered list of *Get_Conditional_Data* macros. This macro contains a condition and a list of patient data items that must be obtained.

2. **Computing risk** - The risk calculations are performed through the *Risk Calculation Macro*. This macro has to be defined. It would contain definitions of variables that are to be created and the calculation of those variables through Assignment_Actions.

3. Recommendations based on risk - Recommendations based on computed risk and individual risk factors are provided through the *Recommendation_Based_On_Risk_Macro*. Recommendations are provided only if an associated condition is true. In the example shown in Figure 63, the Exercise_recommendation is provided only to high-risk persons. Thus, the Recommendations_Macro is structurally similar to the Get_Conditional_data_macro.

Figure 62 and Figure 63 show an example of a risk assessment macro and its expansion.

Figure 62. Risk Assessment Macro based on age, gender, menstrual history, and pregnancy history. This risk assessment macro recommends exercise for risk > 70%, maintaining life style for risk < 30%, and estrogen treatment for post menopausal women.

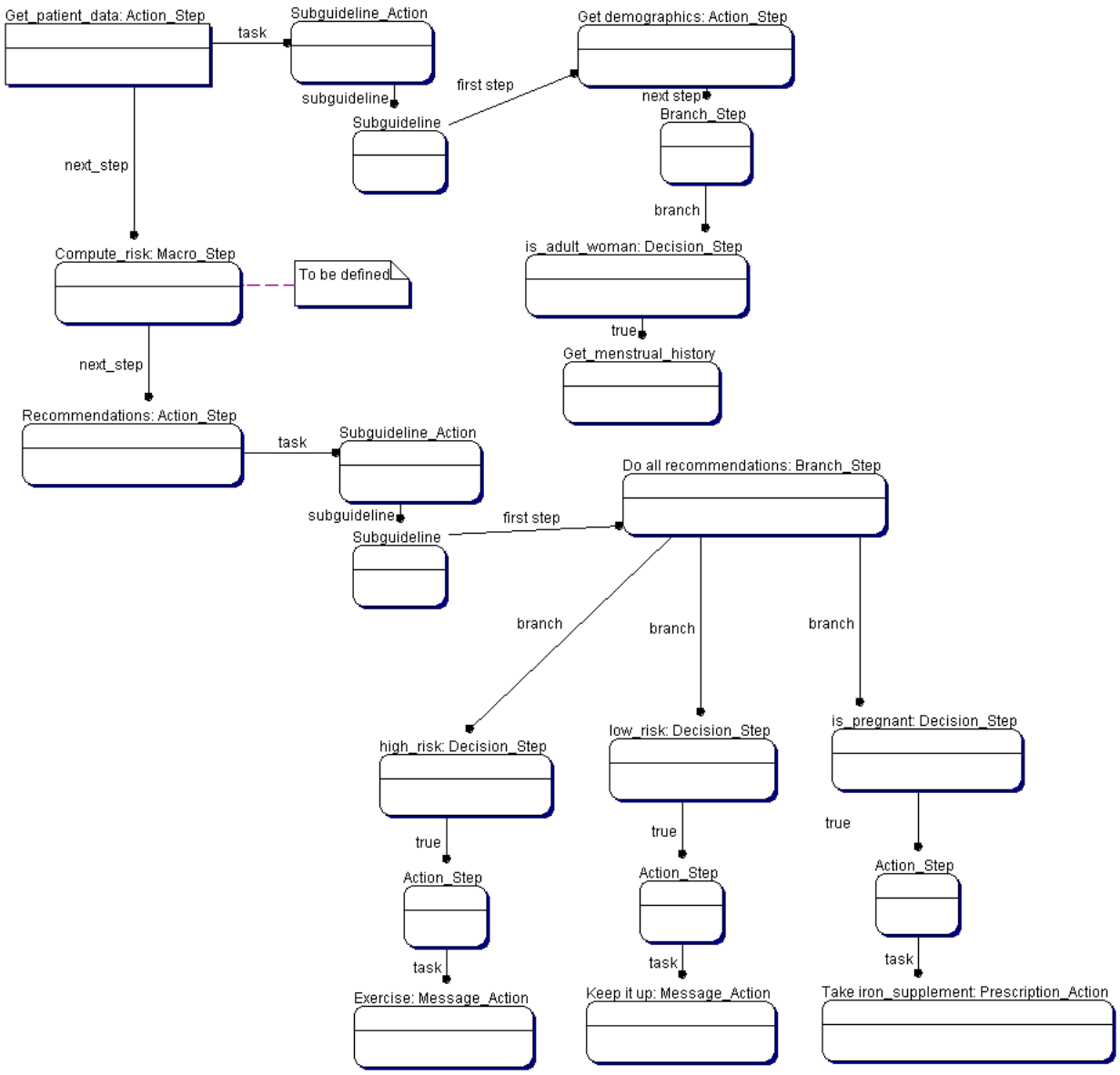
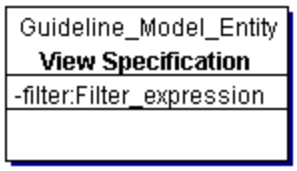


Figure 63. An expansion of the Risk Assessment Macro shown in Figure 62 into its primitive GLIF steps.

2. Views of a guideline

For each guideline default viewers may be specified. Since different users may be interested in different parts of a large, complex guideline, differential display capability is supported. This capability is provided through the use of filters that collapse segments of the guideline into a default view of the guideline customized to a given user, situation, etc. Default viewers are specified using a View_Specification.



A capability to provide multiple views of the same guideline was added in GLIF3. Since different users may be interested in different parts of a large, complex guideline, differential display capability is supported. This capability is provided through the use of filters that collapse segments of the guideline into a default view of the guideline customized to a given user, situation, etc.

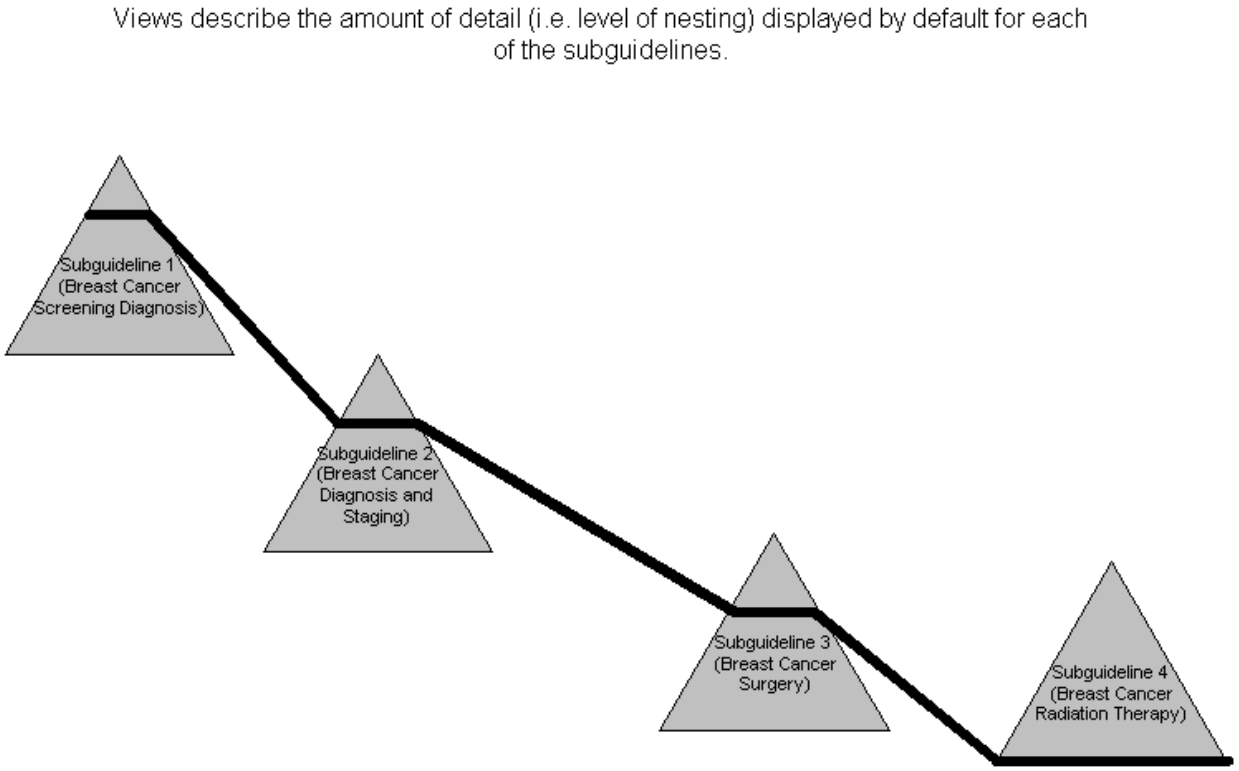


Figure 64. A breast cancer guideline viewed by a radiation oncologist

If a guideline consists of subguidelines, each of these subguidelines may be visualized as a triangle, with one step at the highest level and multiple steps at the lowest level. That is, the width of the triangle is proportional to the number of steps at that level of nesting. The top of the triangle has less detail and therefore has a smaller number of steps. The bottom of the triangle has more detail and therefore has a larger number of steps.

A given filter (e.g., MD_Radiation_Oncologist) will define the default level of nesting/Zoom-in for each of the subguidelines. It will be up to the guideline author to define the subguidelines in an appropriate way (e.g., to avoid too many steps per screen for a given viewer) and to define the

level of nesting required for each given subguideline. In the above example, suppose that a breast cancer guideline has four subguidelines. A Radiation Oncologist looking at the guideline may see, by default, relatively little detail about screening, diagnosis and surgery. He will see a great deal of detail regarding radiation therapy, however. A surgeon looking at the same guideline may see little detail on screening and diagnosis, a lot of detail on surgery and little detail on radiation therapy.

The status quo of specialty bodies publishing guidelines may change as multi-specialty organizations publish multi-specialty documents. Guidelines may become quite complex. Much of medicine is multi-disciplinary in nature. The distinction between specialties is artificial. For example, the distinction between cardiology and nursing is for the convenience of practitioners. The patient suffering a myocardial infarction (heart attack) is likely to require care from both a cardiologist and a nurse. The information needs of the cardiologist, however, are very different from those of the nurse. The purpose of default views in GLIF should be to reveal to the cardiologist only the relevant portions of the myocardial infarction guideline, which may be different from that shown to the nurse.

Views are default filters through which we interact with the guideline. By definition, views do not change guideline logic (e.g. if an RN should do something different from an MD, this should be represented in the guideline logic, not in the view). Although we anticipate that the most common use of views will be user and/or location, there may be other relevant filters (e.g. situation such as routine vs. disaster). The view class is a guideline entity. Alternatively, the view could have been modeled as an enumerated type attribute. The main purpose of this class is to allow differential display in the simplest possible way.

The view specification was chosen to be at the level of guideline entities and not at the attribute level. We may later choose to make attributes (and not entire guideline entities) visible or invisible to some users.

The BNF notation for filter expressions:

term: filter_type = domain_ontology_filter_instance

filter_expression: term | expression binary_operator expression | unary_operator expression | (expression)
binary_operator: OR | AND

unary_operator: NOT

filter_type: USER | LOCATION

domain_ontology_filter_instance: MD | RN | ...

An example of a view specification is shown in Figure 65 and Figure 66.

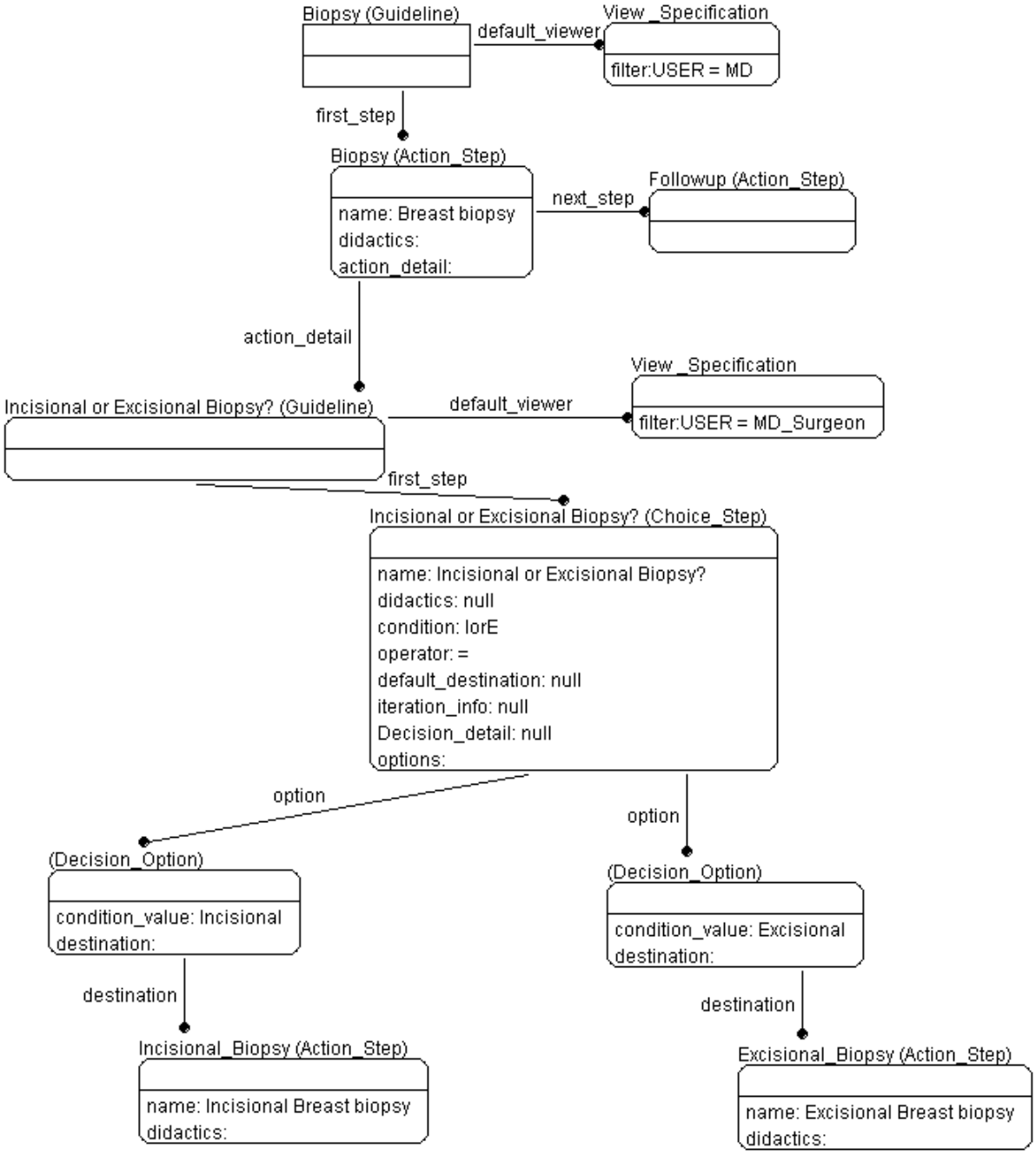
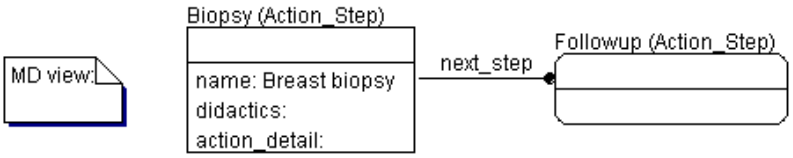


Figure 65. Specifying views: a guideline might call for a breast biopsy. Lets say that all MDs want to see that a breast biopsy is called for, however, surgeons want to know what kind of biopsy is needed, incisional or excisional.


MD view:
Biopsy (Action_Step)
name: Breast biopsy
didactics:
action_detail:
next_step
Followup (Action_Step)

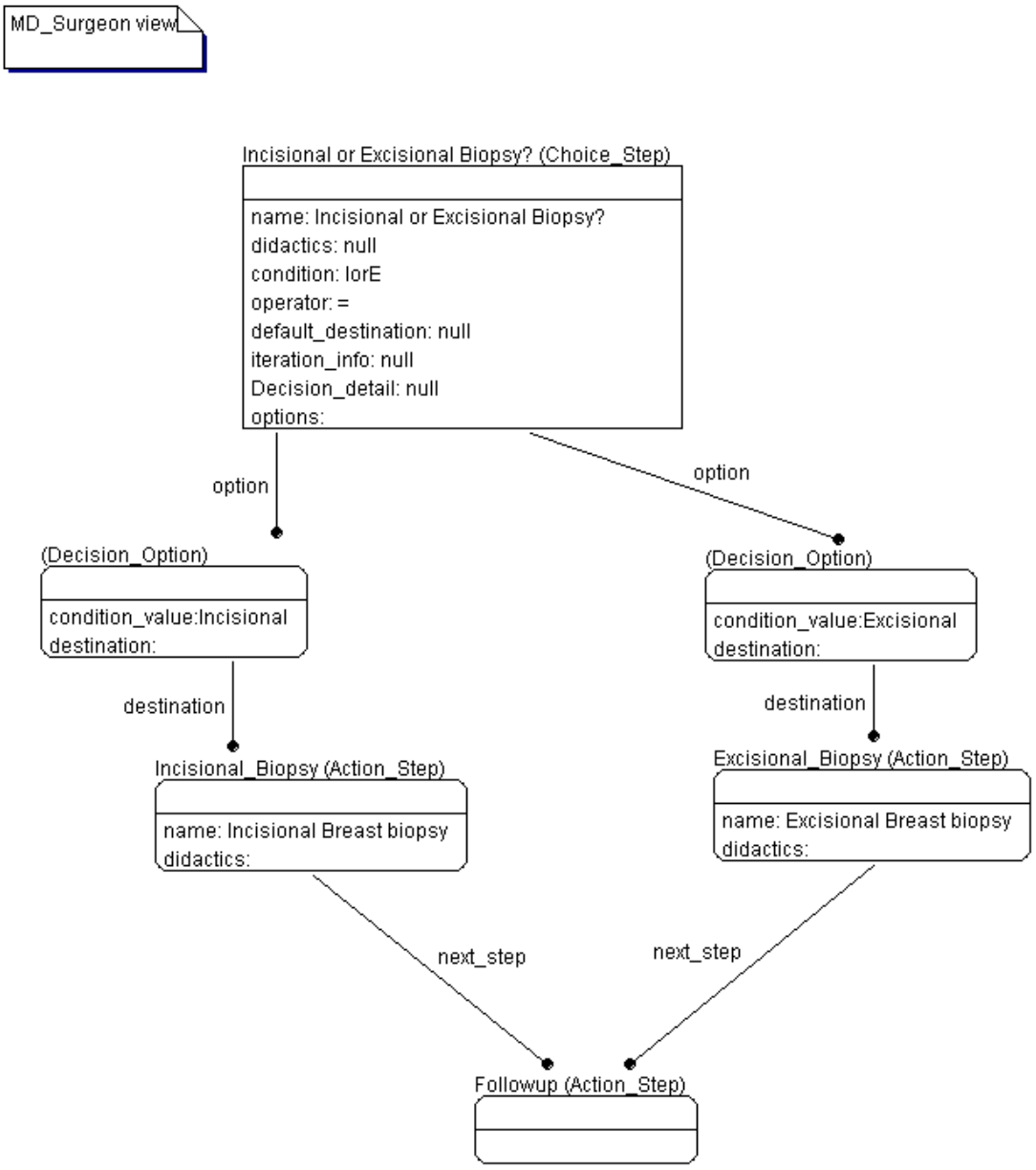

MD_Surgeon view
Incisional or Excisional Biopsy? (Choice_Step)
name: Incisional or Excisional Biopsy?
didactics: null
condition: IorE
operator: =
default_destination: null
iteration_info: null
Decision_detail: null
options:
option
option
(Decision_Option)
condition_value:Incisional
destination:
(Decision_Option)
condition_value:Excisional
destination:
destination
destination
Incisional_Biopsy (Action_Step)
name: Incisional Breast biopsy
didactics:
Excisional_Biopsy (Action_Step)
name: Excisional Breast biopsy
didactics:
next_step
next_step
Followup (Action_Step)

Figure 66. How different users view the guideline. This example shows how nesting deals with views. If the viewer is an MD he sees the top-level view of the action step Biopsy. He can zoom into the action-detail subguideline, to see that incisional or excisional biopsies can be performed. An MD_Surgeon will directly see the zoomed-in view of biopsy directly, showing the decision that is made between incisional and excisional biopsy.

3. Specifying events and exceptions

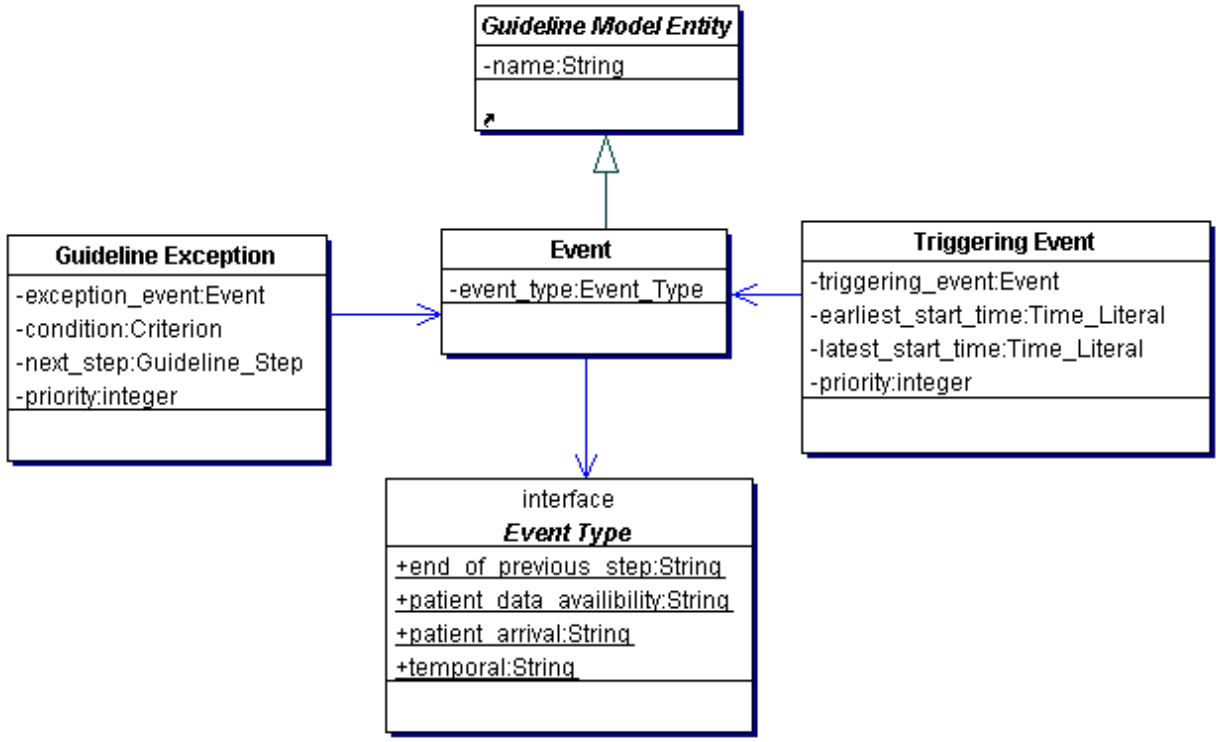


Figure 67. Events and Exceptions class diagram

Action- and decision steps have an attribute, called triggering_events, which specify the events that trigger the start of the step, and the associated earliest and latest times after which the step should be started. A step may contain several triggering events. Any one of the triggering events that occur can trigger the step. If more than one of these triggering events occur at the same time, then the highest priority event is chosen to trigger the step, as specified by the priority attribute of the Triggering Event class. Different event types are defined: end of a previous guideline step, patient arrival, patient data availability, and temporal events, such as a certain point of time has arrived.

The links that connect the guideline steps of an algorithm (i.e., the "next step" attribute of action steps, patient state steps and synchronization steps, the "default next step" of decision steps, the "branches" of branch steps, and the "destination" of decision options) represent triggering events of type "end of previous step" that trigger the guideline step that is adjacent to the arrowhead of

the link. There is no need to explicitly specify them as triggering events unless earliest or latest start times should be specified. [What is the priority of "Next_Step" that is not explicitly specified?]

If an action step has a start_time constraint, then it is applied to all of the action's tasks.

Examples of events are shown in Figure 68.

Action and decision steps, as well as guidelines, have an attribute, called exceptions, which specifies the exceptions that should be checked during the execution of the step. The exceptions are of the class GLException. This class specifies the exception-event that should be checked for, a (guarding) condition and a next step. If the exception event occurs and the condition holds, then we terminate the step associated with the exception, and move on to the next step that is defined by the exception.

Any of the exceptions that occur can stop the execution of the current step and pass control to another step. [problem: Again, control can go outside branch and synchronization] In cases where several exceptions are defined (each with its own next_step), their priorities are compared, and the highest priority exception is chosen to trigger the step. This way, the control is passed to the guideline step that is specified by the exception that has the highest priority.

An example of an exception is shown in Figure 69.

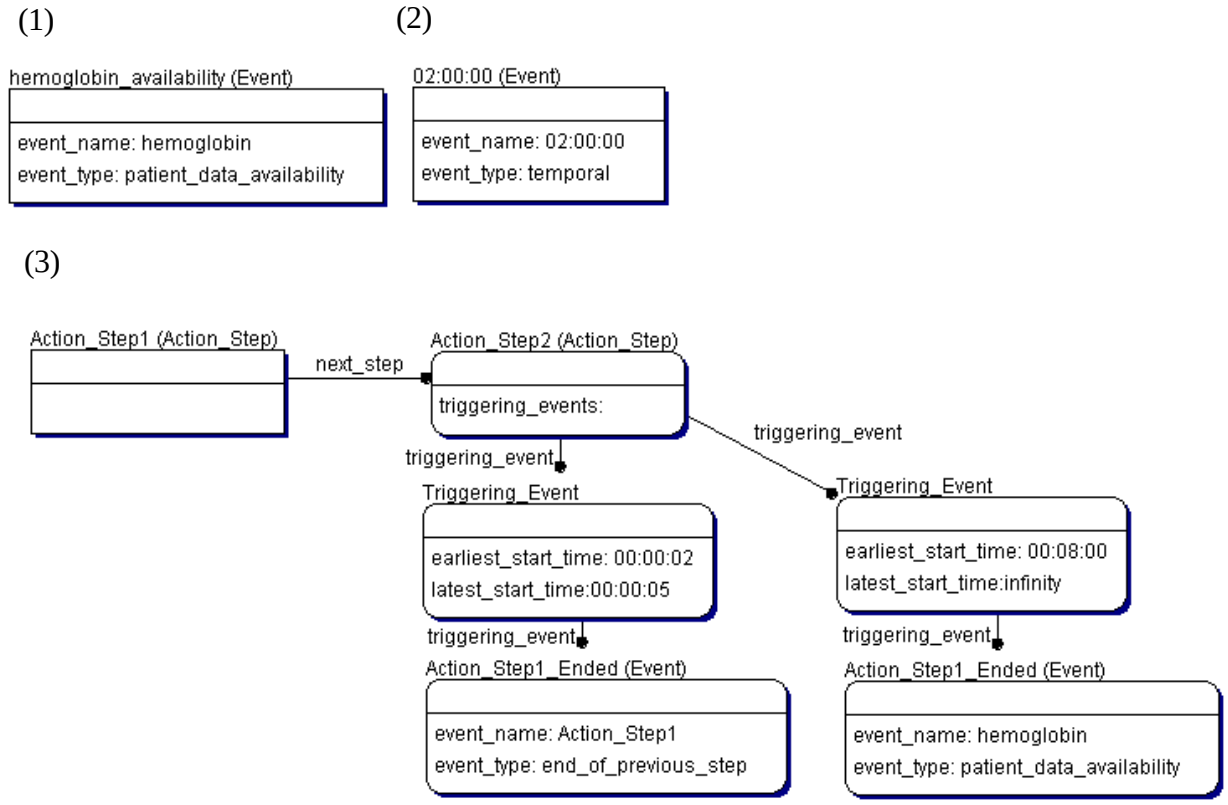


Figure 68. Examples of triggering_events. 1) hemoglobin data is available; 2) 2 am arrived; 3) Action Step2 is invoked by one of two events: (a) at least 2 seconds and not more than 5 seconds after Action Step1 ended; (b) at least 8 minutes after hemoglobin data is available.

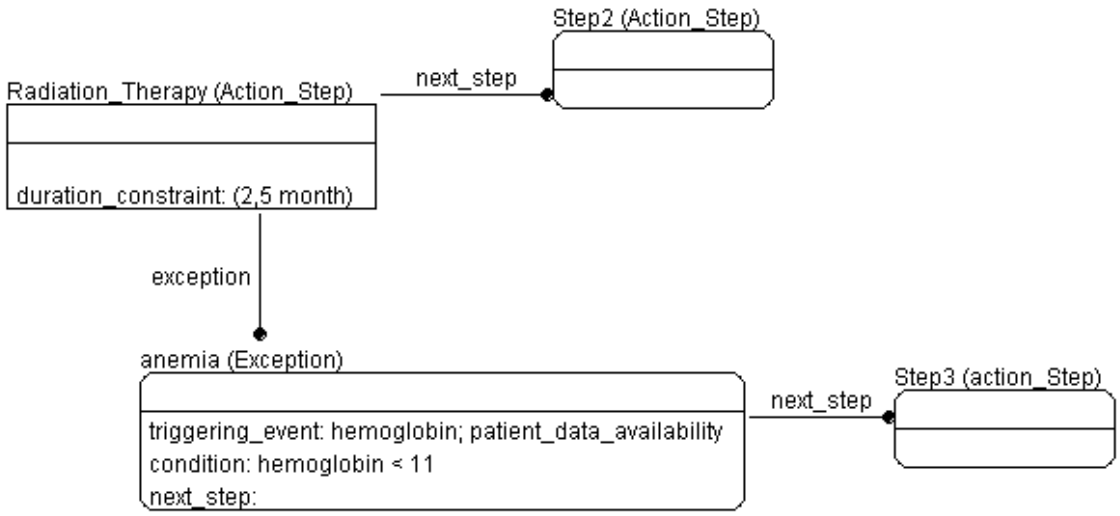


Figure 69. An example of an exception. When radiation therapy is conducted, you check for the exception of anemia (hemoglobin result with a value of < 11). If it occurs then you go to Step3. If it doesn't you finish radiation therapy and go to step2 .

B. Appendix B:

BNF for GEL: GLIF Expression Language

NON-TERMINALS

```
 CompilationUnit ::= ( StatementOrExpression ( <EOL> )+ )* <EOF>
  StatementOrExpression ::= Assignment
                |FunctionStatement
                |LetStatement
                |IfStatement
                |ConcludeStatement
                |Expression
Statement ::=Assignment
                |FunctionStatement
                |LetStatement
                |IfStatement
                |ConcludeStatement
Assignment ::= Id <ASSIGN> Expression <SEMICOLON>
LetStatement ::=<LET> Id <BE> StringConst <SEMICOLON>
FunctionStatement ::= <ID> "(" ( ArgumentList )? ")" <SEMICOLON>
IfStatement ::= IF> Expression <THEN> Statement ( <ELSE> Statement )? <ENDIF> <SEMICOLON>
ConcludeStatement ::= <CONCLUDE> Expression <SEMICOLON>
Expression::= ConditionalExpression
ConditionalExpression ::=  ListAppendExpression
ListAppendExpression ::= List <COMMA> Expression
                | WhereExpression ( <MERGE> WhereExpression )?
 WhereExpression::= OrExpression ( <WHERE> OrExpression )?
 OrExpression ::=
          ConditionalAndExpression ( <OR> ConditionalAndExpression | <XOR> ConditionalAndExpression )*
 ConditionalAndExpression ::= ComparisonExpression ( <AND> ComparisonExpression )*
                |<AT_LEAST> Number <OF> "(" ArgumentList ")"
 ComparisonExpression ::= ConcatExpression ( <EQUAL> ConcatExpression | <NOTEQUAL> ConcatExpression |
                    <LT> ConcatExpression | <LEQUAL> ConcatExpression |
                      <GT> ConcatExpression | <GEQUAL> ConcatExpression | <IS_WITHIN> <SAME_DAY_AS>
ConcatExpression | <IS_WITHIN> <PAST>
                ConcatExpression | <IS_WITHIN> ConcatExpression ( <TO> ConcatExpression | <PRECEDING>
ConcatExpression | <FOLLOWING>
                ConcatExpression | <SURROUNDING> ConcatExpression ) | <IS_BEFORE> ConcatExpression |
                 <IS_AFTER> ConcatExpression | <IS_IN> ConcatExpression | <OCCURS_AT> ConcatExpression |
                <OVERLAPS> ConcatExpression )*
ConcatExpression ::= AddExpression ( <CONCAT> AddExpression )*
AddExpression::=
MultiplyExpression ( <MINUS> MultiplyExpression | <PLUS> MultiplyExpression )*
MultiplyExpression ::= PowerExpression ( <TIMES> PowerExpression | <DIVIDE> PowerExpression )*
```

```
PowerExpression ::= B4AfterExpression ( <POWER> PowerExpression )*
B4AfterExpression ::= UnaryExpression ( <BEFORE> UnaryExpression | <AFTER> UnaryExpression )*
UnaryExpression
        ::=UnaryMinus
        |UnaryPlus
        |MinusDuration
        |PlusDuration
        |<NOT> UnaryExpression
        |<FIRST> UnaryExpression
        |<LAST> UnaryExpression
        |<LATEST> UnaryExpression
        |<EARLIEST> UnaryExpression
        |<ANY_OF> "(" ArgumentList ")"
        |<ALL_OF> "(" ArgumentList ")"
        |<IS_NULL> UnaryExpression
        |<IS_BOOLEAN> UnaryExpression
        |<IS_UNKNOWN> UnaryExpression
        |<IS_NUMBER> UnaryExpression
        |<IS_TIME> UnaryExpression
        |<IS_DURATION> UnaryExpression
        |<IS_STRING> UnaryExpression
        |<IS_LIST> UnaryExpression
        |<TIME_OF> UnaryExpression
        |<EXTRACT_YEAR> UnaryExpression
        |<EXTRACT_MONTH> UnaryExpression
        |<EXTRACT_DAY> UnaryExpression
        |<EXTRACT_HOUR> UnaryExpression
        |<EXTRACT_MINUTE> UnaryExpression
        |<EXTRACT_SECOND> UnaryExpression
        |<EXTRACT_DATE> UnaryExpression
        |Duration <AGO>
        |Duration <FROM_NOW>
        |PrimaryExpression
    UnaryMinus ::= "(" <MINUS> Number ")"
    UnaryPlus ::= "(" <PLUS> Number ")"
    MinusDuration ::= "(" <MINUS> Duration ")"
    PlusDuration ::= "(" <PLUS> Duration ")"
    PrimaryExpression ::= Literal
        |Function
        |Id
        |"(" Expression ")"
        |  It
     It ::= <IT>
     Id  ::= <ID> ( "." <ID> )*
     Duration ::= <NUMBER> ( <YEAR> | <MONTH> | <WEEK> | <DAY> | <HOUR> | <MINUTE> |
                  <SECOND> )
     Function
        ::=
          <ID> "(" ( ArgumentList )? ")"
     ArgumentList
        ::=
          Expression ( "," Expression )*
     Interval ::=
              <INTERVAL> ( "(" | "[" ) ( ( <NUMBER> | <MINUS_INFINITY> ) "," ( <NUMBER> |
             <INFINITY> ) ( ")" | "]" ) | UnaryMinus "," UnaryMinus ( ")" | "]" ) | UnaryMinus "," (
```

```
                <NUMBER> | <INFINITY> ) ( ")" | "]" ) |
               ( Date "," Date ) ( ")" | "]" ) | ( Duration | MinusDuration ) "," ( Duration | MinusDuration ) ( ")" |
   "]" ) )
ListElement ::=Literal  |Id
List ::= ( "{" "}" | "{" ListElement ( <COMMA> ListElement )* "}" )
StringConst ::= <STRING>
Number ::= <NUMBER> Id
        |<NUMBER>
Date::=<DATE>
        |<NOW>
ComplexType ::=
          <STRUCT> <ID> "{" ( <EOL> )+ ( <ID> ":=" Literal <SEMICOLON> ( <EOL> )+ )+ "}"
Literal ::= StringConst
        |Duration
        |Number
        | Date
        |Interval
        |List
        |<TRUE>
        |<FALSE>
        |<UNKNOWN>
        | ComplexType
```

Tokens/Terminals

```
TOKEN : /* RESERVED WORDS */
{
  < BE: "be" >
| < BOOLEAN: "boolean" >
| < DATE1: "date" >
| < DURATION: "duration" >
| < FALSE: "false" >
| < INFINITY: "infinity" >
| < MINUS_INFINITY: "-infinity" >
| < IT: "it" >
| < LET: "let" >
| < NOW: "now" >
| < NULL: "null" >
| < RES_NUMBER: "number" >
| < RES_STRING: "string" >
| < TIME: "time" >
| < TRUE: "true" >
| < UNKNOWN: "unknown" >
| < STRUCT: "struct" >
}

TOKEN : /* OPERATORS */
{
  < COMMA: "," >
| < WHERE: "where" >
| < OR: ("|" | "or") >
| < XOR: ("*|" | "xor") >
```

```
| < AND: ("&" | "and") >
| < NOT: ("!" | "not") >
| < EQUAL: ("==" | "=") > // here to next comment -- same precedence
| < NOTEQUAL: ("!=" | "<>" ) >
| < LT: "<" >
| < LEQUAL: "<=" >
| < GT: ">" >
| < GEQUAL: ">=" >
| < IS_WITHIN: "is within" >
| < TO: "to" >
| < PRECEDING: "preceding" >
| < FOLLOWING: "following" >
| < SURROUNDING: "surrounding" >
| < PAST: "past" >
| < SAME_DAY_AS: "same day as" >
| < BEFORE: "before" >
| < AFTER: "after" >
| < IS_BEFORE: "is " < BEFORE > >
| < IS_AFTER: "is " < AFTER > >
| < IS_IN: ("is ")? "in" >
| < OCCURS_AT: "occurs at" > // end same precedence
| < IS_NULL: "is " < NULL > >
| < IS_BOOLEAN: "is " < BOOLEAN > >
| < IS_UNKNOWN: "is " < UNKNOWN > >
| < IS_NUMBER: "is number" >
| < IS_TIME: "is " < TIME > >
| < IS_DURATION: "is " < DURATION > >
| < IS_STRING: "is string" >
| < IS_LIST: "is list" >
| < CONCAT: "||" | "concat" >
| < PLUS: "+" >
| < MINUS: "-" >
| < TIMES: "*" >
| < DIVIDE: "/" >
| < POWER: ("**" | "^") >
| < AGO: "ago" >
| < FROM_NOW: "from now" >
| < YEAR: "years" | "year" >
| < MONTH: "months" | "month" >
| < WEEK: "weeks" | "week" >
| < DAY: "days" | "day" >
| < HOUR: "hours" | "hour" >
| < MINUTE: "minutes" | "minute" >
| < SECOND: "seconds" | "second" >
| < EXTRACT_YEAR: "extract " < YEAR > >
| < EXTRACT_MONTH: "extract " < MONTH > >
| < EXTRACT_DAY: "extract " < DAY > >
| < EXTRACT_HOUR: "extract " < HOUR > >
| < EXTRACT_MINUTE: "extract " < MINUTE > >
| < EXTRACT_SECOND: "extract " < SECOND > >
| < EXTRACT_DATE: "extract " < DATE1 > >
| < ANY_OF: "any of" >
| < ALL_OF: "all of" >
| < LAST: "last" >
| < FIRST: "first" >
```

```
| < INTERVAL: "interval" >
| < AT_LEAST: "at least" >
| < OF: "of" >
| < EVERY: "every" >
| < OVERLAPS: "overlaps" >
| < LATEST: "latest" >
| < EARLIEST: "earliest" >
| < MERGE: "merge" >
| < TIME_OF: "time of" >
}

TOKEN : /* STATEMENTS */
{
  < IF: "if" >
| < THEN: "then" >
| < ELSE: "else" >
| < ENDIF: "endif" >
| < CONCLUDE: "conclude" >
| < ASSIGN: ":=" >
}

TOKEN : /* IDENTIFIERS -- VARIABLES OR FUNCTION NAMES */
{
  < ID: ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","0"-"9"] | "_" (["a"-"z","A"-"Z","0"-"9"])+ )* ("." ["a"-"z","A"-"Z"]
(["a"-"z","A"-"Z","0"-"9"] | "_" (["a"-"z","A"-"Z","0"-"9"])+ )* )* >
}

TOKEN : /* LITERALS */
{
  < STRING: "\""(~["\"", "\n", "\r"])*"\"" >
| < NUMBER:
            (["0"-"9"])* "." (["0"-"9"])+ (<EXPONENT>)? (["l","L","f","F"])?
           |(["0"-"9"])+ <EXPONENT> (["l","L","f","F"])?
           |(["0"-"9"])+ >
| < EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
| < DATE: ["0"-"9"]["0"-"9"]["0"-"9"]["0"-"9"]"-"["0"-"9"]["0"-"9"]"-"["0"-"9"]["0"-"9"]
            ("T"["0"-"9"]["0"-"9"](":"["0"-"9"]["0"-"9"](":"["0"-"9"]["0"-"9"]("."(["0"-"9"])+ )?("Z" | <DIFF>)?)?)?)?
>
| < DIFF: "+"["0"-"9"]["0"-"9"]":"["0"-"9"]["0"-"9"]
           |"-"["0"-"9"]["0"-"9"]":"["0"-"9"]["0"-"9"] >
}
```

Arden operators not supported by GEL

operator	**description**	**Reason for not supporting**
Sort time	Sorts a list by primary time	Could now be supported
Sort data	Sorts a list by value	Function sortAttribute ?

Exist(s)	Checks if a list is not empty	Could be supported as an operator // isEmpty is a function
Count	Returns the number of elements in a list	Could be supported
minimum	Returns the smallest value from items in a list of identical types	Function minimumAttribute ?
maximum	Returns the largest value from items in a list of identical types	Function maximumAttribute ?
Any	Returns true if any of the items in a list is true	AnyAttribute?
All	Returns true if all of the items in a list are true	AllAttribute?
No	returns true if all the items in a list are false	NoAttribute ?
Element [index]	Returns the i-th element from a list	Could be supported. primary time is maintained
index latest		Could now be supported
index earliest		Could now be supported
index minimum		
index maximum		
reverse	Reverse order of elements in a list, maintaining primary_time	Can be supported
minimum ... from		

maximum .... from		
first ... from		Could now be supported
last ... from		Could be supported
latest ... from		Could now be supported
earliest ... from		Could now be supported
Index minimum ... from		
index maximum ... from		
first .. from		Could be supported
last .. from		Could be supported
interval	Returns the difference between the primary times of succeeding items in a list	Could now be supported
Formatted with		Will not be supported in GEL
Matches pattern		Will not be supported in GEL
SUM		Will not be supported in GEL
AVERAGE		Will not be supported in GEL
MEDIAN		Will not be supported in GEL
Stddev		Will not be supported in GEL
variance		Will not be supported in GEL
Extract characters		Will not be supported in GEL
Seqto	generates a list of integers in ascending order	Will not be supported in GEL

nearest... from		Will not be supported in GEL
index nearest ... from		Will not be supported in GEL
slope		Will not be supported in GEL
increase	list of differences between successive items in a list	Will not be supported in GEL
decrease		Will not be supported in GEL
% increase		Will not be supported in GEL
% decrease		Will not be supported in GEL
Numeric functions		Will not be supported in GEL
synonyms		Will not be supported in GEL

Operators that exist in GEL but not in Arden Syntax:

Unary: from now, is unknown

Binary: overlaps, xor, |* , is a, is-a, occur/occurs/occurred at, at least...of

from now
In Arden, you can refer to the time of an event/occurrence in the past by saying "two days ago". But there is no similar syntax for referring to the time of a future event. "from now" was added so that we can say "[do x] two days from now".

is unknown
Testing if something is null is not the same thing as testing if it is unknown. If I have a data item that has not been initialized or assigned a value, it evaluates to null (e.g. if someone attempts to use the value of a data item without first getting it from the patient record or a physician). This is something that can be tested by Arden. If we want to note that I don't know whether the result of a logical expression is true or false then we can assign the value "unknown" to a variable representing the results of the expression. This variable has a value ("unknown") and is therefore not null.

overlaps
“overlaps” is used for comparing intervals (time or other intervals). So for example, [3:5] overlaps [2:4] would evaluate to true but [3:5) overlaps [5:9] would evaluate to false.

xor, |*
A xor B means ((A or B) and not (A and B))
|* is a synonym for exclusive or (xor).

at least...of
The "at least...of" operator allows us to express very basic existential/universal quantification (i.e., "at least 1 of ... " is equivalent to "there exists ..." and "not (at least 1 of (not...))" is equivalent to "for all ..."). It also allows expressing "k of n" criteria.

occur/occurs/occurred at
"occur/occurs/occurred **at**" is synonymous with Arden's "occur/occurs/occurred **equal**" operator and would be evaluated exactly the same way. It just seemed like it would be clearer to use it in some situations.

GEL Functions used by GLIF

1. isEmpty that accepts a List as a parameter and returns TRUE if the list is empty (i.e., contains no elements, or contains elements that are all empty) and FALSE otherwise.
2. selectAttribute accepts a complex type as an argument and selects an attribute out that complex type.
3. selectAttributeFromList accepts a list of complex objects as an argument. It then returns a list whose elements are the selected attribute of each element in the argument list. Unlike Select_Action, it returns the value only, without maintaining timestamps.
4. containsValuesTimeStamped accepts two list arguments, where the second contains timestamps (what Get_Data returns) and the first one does not (what Get_Knowledge returns). The function returns a list of Booleans of length equal to the length of the second argument of the function. The Booleans take a True value if in that position of the second argument of the function there exists a value that is contained in the first argument of the function.
5. containsValues accepts two list arguments, both without timestamps. The function returns a list of Booleans of length equal to the length of the second argument of the function. The Booleans take a True value if in that position of the second argument of the function there exists a value that is contained in the first argument of the function.

The GLIF Expression Language (GEL)

Types supported by GEL are listed below and expressions involving constants of these types are provided as examples of how to write valid expressions in GEL. A variable in GEL can be assigned a value of any one of the types described below:

Number (real numbers)
String
Extended Boolean (true, false, unknown)
Absolute Date and Time
Duration
List
Numeric Interval
Duration Interval
Absolute Date and Time Interval

Number

Operations supported on numbers include comparisons, addition, subtraction, multiplication, division, exponentiation, unary plus, and
unary minus. A number in GEL is a floating point/real number by default. Use of unsupported operators with numerical values is an error (causes a type mismatch exception to be raised).

Unary operators:

+
Description: unary plus operator
Sample expression: (+3)
Returns: 3
Note: the parentheses are required

-
Description: unary minus operator
Sample expression: (-50)
Returns: -50
Note: the parentheses are required

is number
Description: checks type of argument and returns true if it is a number
Sample expression: is number 225
Returns: true
Sample expression: is number “hey”
Returns: false

Binary operators:

+
Description: addition operator
Sample expression: 2 + 3
Returns: 5

-

Description: subtraction operator
Sample expression: 2 - 3
Returns: -1

*
Description: multiplication operator
Sample expression: 50 * (-3)
Returns: -150

/
Description: division operator
Sample expression: 180 / 6
Returns: 30
Sample expression: 22 / 7
Returns: 3.142857142857143

^ *or* **
Description: exponentiation operator
Sample expression: 2 ^ 5
Returns: 32
Sample expression: 3 ** 6
Returns: 729
Sample expression: 2 ^ (-4)
Returns: 0.0625

<
Description: less than operator
Sample expression: 5 < 4
Returns: false

>
Description: greater than operator
Sample expression: (-9) > (-18)
Returns: true

<=
Description: less than or equal to operator
Sample expression: 51 <= 51
Returns: true

>=
Description: greater than or equal to operator
Sample expression: 200 >= 165
Returns: false

= *or* ==
Description: equality operator
Sample expression: 20 == 12
Returns: false
Sample expression: 1 = 1
Returns: true

!= *or* <>
Description: inequality operator
Sample expression: 20 <> 12

Returns: true
Sample expression: 1 != 1
Returns: false

Ternary operators:

is within … to …
Description: checks that first argument is in the inclusive range defined by the second and third arguments
Sample expression: 5 is within 4 to 5
Returns: true
Sample expression: 10 is within 2 to 9
Returns: false

String

Operations supported on strings include concatenation and lexicographic comparisons. Use of unsupported operators with string values is an error (causes a type mismatch exception to be raised).

Unary operators:

is string
Description: checks type of argument and returns true if it is a string
Sample expression: is string 225
Returns: false
Sample expression: is string "hey"
Returns: true

Binary operators:

||
Description: concatenation operator
Sample expression: "hello " || "world"
Returns: "hello world"

<
Description: less than operator (checks whether the 1st argument lexicographically precedes the 2nd argument)
Sample expression: "a" < "aa"
Returns: true
Sample expression: "d" < "b"
Returns: false

>
Description: greater than operator (checks whether the 1st argument lexicographically follows the 2nd argument)
Sample expression: "yy" > "ab"
Returns: true

<=
Description: less than or equal to operator (checks whether the 1st arg. lexicographically precedes or equals the 2nd)
Sample expression: "cd" <= "cd"
Returns: true

>=
Description: greater than or equal to operator (checks whether the 1st arg. lexicographically follows or equals the 2nd)
Sample expression: "zed" >= "zee"
Returns: false

= *or* ==
Description: equality operator
Sample expression: "why" == "not"
Returns: false

!= *or* <>
Description: inequality operator
Sample expression: "why" <> "not"
Returns: true

Ternary operators:

is within … to …
Description: checks that first argument is in the inclusive range defined by the second and third arguments
Sample expression: "aa" is within "a" to "b"
Returns: true
Sample expression: "c" is within "cc" to "ea"
Returns: false

4. Extended Boolean

Extended booleans in the expression language describe a 3-valued logic (true, false, and unknown). Operations on extended booleans include logical ands, ors, xors, etc. Use of unsupported operators with extended boolean values is an error (causes a type mismatch exception to be raised).

Unary operators:

is boolean
Description: checks type of argument and returns true if it is an extended boolean
Sample expression: is boolean unknown
Returns: true
Sample expression: is boolean 0
Returns: false

is unknown
Sample expression: is unknown true
Returns: false
Sample expression: is unknown false
Returns: false
Sample expression: is unknown unknown
Returns: true

not *or* !
Description: logical not
Sample expression: not true
Returns: false
Sample expression: ! false
Returns: true
Sample expression: not unknown
Returns: unknown

any of
Description: returns true if any of the logical expressions in its argument evaluates to true. Expects a comma separated "list" of logical expressions as its argument.
Sample expression: any of (3>4, 67 < 99, true == true, true xor false)
Note, equivalent to: any of (false, true, true, true)
Returns: true

all of
Description: returns true if all of the logical expressions in its argument evaluate to true. Expects a comma separated "list" of logical expressions as its argument.
Sample expression: all of (3>4, 67 < 99, true == true, true xor false)
Note, equivalent to: all of (false, true, true, true)
Returns: false

Binary operators:

= *or* ==
Description: equality operator
Sample expression: true == unknown
Returns: false

!= *or* <>
Description: inequality operator
Sample expression: false != unknown
Returns: true

and *or* &

Description: logical and

Sample expression: true and true
Returns: true
Sample expression: true and false
Returns: false
Sample expression: true and unknown
Returns: unknown
Sample expression: false & false

Returns: false
Sample expression: false & unknown
Returns: false
Sample expression: unknown & unknown
Returns: unknown

or *or* |
Description: logical or

Sample expression: true or true
Returns: true
Sample expression: true or false
Returns: true
Sample expression: true or unknown
Returns: true
Sample expression: false | false
Returns: false
Sample expression: false | unknown
Returns: unknown
Sample expression: unknown | unknown
Returns: unknown

xor *or* *|
Description: exclusive or

Sample expression: true xor true
Returns: false
Sample expression: true xor false
Returns: true
Sample expression: true xor unknown
Returns: unknown
Sample expression: false *| false
Returns: false
Sample expression: false *| unknown
Returns: unknown
Sample expression: unknown *| unknown
Returns: unknown

The following binary operator expects a number followed by a comma-separated list of logical expressions:

at least … of …
Description: returns true if the number of logical expressions in its right argument that evaluate to true equal or exceed its numeric argument.
Sample expression: at least 2 of (3>4, 67 < 99, true == true, true xor false)
Note, equivalent to: at least 2 of (false, true, true, true)
Returns: true
Sample expression: at least 5 of (3>4, 67 < 99, true == true, true xor false)
Note, equivalent to: at least 5 of (false, true, true, true)
Returns: false

Absolute Date and Time

Absolute dates and times and operations on them are defined with respect to a Gregorian calendar. Operations on absolute dates and times include comparisons, subtraction, etc. Use of unsupported operators with absolute date and time values is an error (causes a type mismatch exception to be raised). An absolute date and time value that does not end in a Z for universal coordinated time (UTC) or in a +/- hh:mm offset is assumed to be defined in local time. Note that the expression **now** yields the current time on the particular system running an interpreter for GEL.

Unary operators:

is time

Description: checks type of argument and returns true if it is an absolute date and time

Sample expression: is time 1999-03-04T03:30:45.742-03:00
Returns: true
Sample expression: is time 2000-09-12
Returns: true
Sample expression: is time **now**
Returns: true

Sample expression: is time 23
Returns: false

extract date

Description: extracts the date portion of the argument and returns it as an absolute date and time in local time

Sample expression: extract date 1998-03-04T03:30:45.742+05:30
Returns: 1998-03-04

Sample expression: extract date **now** (assuming **now** is 2000-10-03T17:59:10.240-04:00)
Returns: 2000-10-03

extract year

Description: extracts the year portion of an absolute date and time

Sample expression: extract year 1998-03-04T03:30:45.742-03:00
Returns: 1998

extract month

Description: extracts the month portion of an absolute date and time

Sample expression: extract month 2001-11-05
Returns: 11

extract day

Description: extracts the day of the month from an absolute date and time

Sample expression: extract day 1950-12-25
Returns: 25

extract hour

Description: extracts the hour of the day from an absolute date and time

Sample expression: extract hour 1960-10-01T03:04:30
Returns: 3

extract minute

Description: extracts the number of minutes past the hour from an absolute date and time

Sample expression: extract minute 1960-10-01T03:04:30
Returns: 4

extract second

Description: extracts the number of seconds past the hour from an absolute date and time

Sample expression: extract second 1960-10-01T03:04:30
Returns: 30

Binary operators:

-
Description: subtract one absolute date and time from another to produce a duration in seconds
Sample expression: 2000-03-01T00:00:00 - 2000-02-01T00:00:00
Returns: 2505600 seconds

occurs at
Description: checks that first argument and the second argument are equal
Sample expression: 2000-03-10T05:04:03 occurs at 2000-03-10T12:55:43
Returns: false
Sample expression: 2000-03-10T00:00:00 occurs at 2000-03-10T23:59:59
Returns: false
Sample expression: 2000-03-10T05:04:03 occurs at 2000-03-10T05:04:03
Returns: true

is within same day as
Description: checks that first argument and the second argument occur on the same day (a new day begins at midnight)
Sample expression: 2000-03-10T05:04:03 is within same day as 2000-03-10T12:55:43
Returns: true
Sample expression: 2000-03-10T00:00:00 is within same day as 2000-03-10T23:59:59
Returns: true
Sample expression: 2001-03-10T05:04:03 is within same day as 2000-03-10T12:55:43
Returns: false

is before
Description: determines whether one date occurs before another
Sample expression: 2000-03-01T00:00:00 is before 2000-02-01T00:00:00
Returns: false

is after
Description: determines whether one date occurs before another
Sample expression: 2000-03-01T00:00:00 is after 2000-02-01T00:00:00
Returns: true

<
Description: less than operator (equivalent to is before)

>
Description: greater than operator (equivalent to is after)

<=
Description: less than or equal to operator

>=
Description: greater than or equal to operator

= *or* ==
Description: equality operator (same as occurs at)
Sample expression: 2010-03-01T00:00:00 == 2009-03-01T00:00:00

Returns: false

!= *or* <>
Description: inequality operator
Sample expression: 2010-03-01T00:00:00 != 2009-03-01T00:00:00
Returns: true

The following binary operators expect a time followed by a duration:

is within past

Description: checks that first argument is within the duration specified by **now** minus the second argument to **now**
Sample expression: 2000-10-02T00:00:00 is within past 2 days (assuming that **now** is 2000-10-04T19:04:18.650-04:00)
Returns: false
Note: this operator calculates past two 2 days as **48 hours before the present time**
If two days prior is meant to start at midnight, other expressions could be substituted such as:
(2000-10-02T00:00:00 >= extract date (2 days ago)) and (2000-10-02T00:00:00 <= extract date now)
Sample expression: 2000-10-02T23:30:00 is within past 2 days (assuming that **now** is 2000-10-04T19:04:18.650-04:00)
Returns: true

-

Description: Subtracts a duration from an absolute date and time

Sample expression: now – 3 days (assuming now is 2000-10-20T15:03:38.419-04:00)
Returns: 2000-10-17T15:03:38.419-04:00
Sample expression: 1998-01-31 - 28 days
Returns: 1998-01-03T00:00:00-05:00

The following binary operators expect a time and a duration as arguments (in no particular order):

+

Description: Adds a duration to an absolute date and time

Sample expression: 1995-03-04 + 720 days
Returns: 1997-02-21T00:00:00-05:00
Sample expression: 5 hours + 1999-03-04T05:00:00
Returns: 1999-03-04T10:00:00-05:00

Ternary operators:

… is within … to …
Description: checks that first argument is in the inclusive range defined by the second and third arguments
Sample expression: 2000-03-10T05:04:03 is within 2000-03-10T05:04:03 to 2000-05-10T05:04:03
Returns: true

The following ternary operators expect as arguments a time followed by a duration followed by a time:

… is within … preceding …

Description: checks that first argument is in the inclusive range defined by the third argument minus the second argument to the third argument
Sample expression: 2000-03-10T05:04:03 is within 4 months preceding 2000-05-10T05:04:03
Returns: true

… is within … following …

Description: checks that first argument is in the inclusive range defined by the third argument to the third argument plus the second argument
Sample expression: 2000-10-03T06:45:23 is within 5 days following 2000-10-01T00:55:46
Returns: true

… is within … surrounding …

Description: checks that first argument is in the inclusive range defined by the third argument minus the second argument to the third argument plus the second argument
Sample expression: 2000-09-29T17:20:01 is within 5 days surrounding 2000-10-01T00:55:46
Returns: true
Sample expression: 2000-10-05T00:00:00 is within 5 days surrounding 2000-10-01T00:55:46
Returns: true
Sample expression: 2000-10-06T19:05:40 is within 5 days surrounding 2000-10-01T00:55:46
Returns: false
Sample expression: (extract date 2000-10-06T19:05:40) is within 5 days surrounding (extract date 2000-10-01T00:55:46)

Returns: true

5. Duration

Operations supported on durations include comparisons, addition, subtraction, multiplication, and division. Use of unsupported operators with duration values is an error (causes a type mismatch exception to be raised. Note that because of the fuzziness associated with certain durations (is 1 year 365 or 366 days? Is 1 month 28, 29, 30, or 31 days?), defaults are used for the number of days in a year (1 year = 365 days in our model), and the number of days in a month (1 month = 31 days in our model). This means that certain operators would return results that differ from the expected. For example the query 1 year == 12 months would return false because 365 days is not equal to 372 (12*31) days.
Ultimately, the best approach to evaluating such fuzzy or vague comparisons might be to apply appropriate methods for handling uncertainty from the Artificial Intelligence literature on uncertainty, or to disallow precise calculations from being made from such imprecise expressions.

5.1 *Unary Operators*

is duration

Description: checks type of argument and returns true if it is a duration

Sample expression: is duration 3 years
Returns: true
Sample expression: is duration 5 months
Returns: true
Sample expression: is duration 20 hours
Returns: true

Sample expression: is duration 23
Returns: false

ago

Description: computes an absolute date and time equivalent to the current time (**now**) minus a duration

Sample expression: 2 days ago (assuming **now** is 2000-10-03T18:19:06.270-04:00)
Returns: 2000-10-01T18:19:06.270-04:00

from now

Description: computes an absolute date and time equivalent to the current time (**now**) plus a duration

Sample expression: 2 days from now (assuming **now** is 2000-10-03T18:19:06.270-04:00)
Returns: 2000-10-05T18:19:06.270-04:00

+
Description: unary plus operator
Sample expression: (+3 days)
Returns: 3 days
Note: the parentheses are required

-
Description: unary minus operator
Sample expression: (-50 hours)
Returns: -50 hours
Note: the parentheses are required

Binary operators:

+

Description: Adds a duration to another duration (returns a duration in seconds unless the duration specifiers are the same)

Sample expression: 340 days + 91 days
Returns: 431 days
Sample expression: 6 hours + 42 days
Returns: 3650400 seconds

-

Description: Subtracts a duration from another duration (returns a duration in seconds unless the duration specifiers are the same)

Sample expression: 340 days - 91 days
Returns: 249 days
Sample expression: 6 hours - 25 seconds
Returns: 21575 seconds

*
Description: Multiplies a duration by a number to obtain another duration. Order of arguments does not matter.

Sample expression: 40 days * 3
Returns: 120 days

Sample expression: 5 * 30 seconds
Returns: 150 seconds

/
Description: Divides a duration by a number to obtain another duration or divides a duration by a duration to obtain a number

Sample expression: 40 days / 2
Returns: 20 days
Sample expression: 2 minutes / 1 second
Returns: 120

<
Description: less than operator
Sample expression: 40 days < 26 days
Returns: false
Sample expression: 360 hours < 1 year
Returns: true

>
Description: greater than operator
Sample expression: 5 years > 12 months
Returns: true

<=
Description: less than or equal to operator
Sample expression: 26 minutes <= 26 minutes
Returns: true
Sample expression: 5 years <= 90 months
Returns: true

>=
Description: greater than or equal to operator
Sample expression: 9 years >= 9 years
Returns: true

= *or* ==
Description: equality operator
Sample expression: 3 days == 5 days
Returns: false

!= *or* <>
Description: inequality operator
Sample expression: 3 days != 5 days
Returns: true

6. List

A list can contain any of the basic operators listed on the first page (including lists). Operations supported on lists include membership checking, etc. Use of unsupported operators with lists is an error (causes a type mis match exception to be raised).

6.1 *Unary Operators*

is list

Description: checks type of argument and returns true if it is a list

Sample expression: is list {{1, 2}, 3, "hey", 1999-03-04}
Returns: true
Sample expression: is list 567
Returns: false

first
Description: returns the first element in a list

Sample expression: first {2000-01-02T00:00:00, 24, 3, "hey", 1999-03-04}
Returns: 2000-01-02T00:00:00

Sample expression: first {{1, 2}, 3, "hey", 1999-03-04}
Returns: {1, 2}

last
Description: returns the last element in a list

Sample expression: last {2000-01-02T00:00:00, 24, 3, "hey", 1999-03-04}
Returns: 1999-03-04

Sample expression: last {{1, 2}, 3, "hey", "string"}
Returns: "string"

6.2 *Binary Operators*

is in

Description: checks whether first argument occurs in the list represented by the second argument

Sample expression: 2 is in {50, 99, 2, 3, "hey", 1999-03-04}
Returns: true
Sample expression: 55 is in {50, 99, 2, 3, "hey", 1999-03-04}
Returns: false

where

Description: the where operator is generally used to select values from a list, and has the form: “expr1 where expr2” (expr1 is usually a list, but can also be a value of any of the other basic types). The right argument to the where operator (expr2) is expected to be a logical expression, a list of extended boolean values, or **true**, **false**, or **unknown**. When the right argument is **true**, the left argument is returned unchanged. When it is **false** or **unknown**, an empty list is returned. When the right argument is a logical expression, it may make use of the keyword “it” to refer to the individual elements contained in the left hand side argument (when this is a list), or to refer to the non-list value that is the left hand side argument. The valid logical expressions that may appear on the right hand side of the where are:

is number it

is string it

is boolean it

is unknown it

is duration it

is time it

is list it

it < subexpr

subexpr < it

it <= subexpr

subexpr <= it

it > subexpr

subexpr > it

it >= subexpr

subexpr >= it

it == subexpr

subexpr == it

it != subexpr

subexpr != it

subexpr is in it

(where subexpr is a value of one of the basic types)

Sample expression: 1 where true

Returns: 1

Sample expression: 1 where false

Returns: {}

Sample expression: 1 where unknown

Returns: {}

Sample expression: 1 where {true, false, unknown, true, true}

Returns: {1, 1, 1}

Sample expression: {4,5,6,7,8,9,10} where it < 7

Returns: {4, 5, 6}

Sample expression: {4,5,6,7,8,9,10} where 7 < it

Returns: {8, 9, 10}

Sample expression: {4,5,6,7,8,9,10} where it <= 7

Returns: {4, 5, 6, 7}

Sample expression: {4,5,6,7,8,9,10} where 7 <= it

Returns: {7, 8, 9, 10}

Sample expression: {1,2,3,4,5,6,7} where it > 4

Returns: {5, 6, 7}

Sample expression: {1,2,3,4,5,6,7} where 4 > it

Returns: {1, 2, 3}

Sample expression: {1,2,3,4,5,6,7} where it >= 4

Returns: {4, 5, 6, 7}

Sample expression: {1,2,3,4,5,6,7} where 4 >= it

Returns: {1, 2, 3, 4}

Sample expression: {1,2,3,4,5,6,7} where it == 4

Returns: {4}

Sample expression: {1,2,3,4,5,6,7} where it != 4

Returns: {1, 2, 3, 5, 6, 7}

Sample expression: {{"CHF", "Mary", 1}, {"CHF", "Don", 2}, {"Angina", "Sam", 3}} where “CHF” is in it

Returns: {{"CHF", "Mary", 1}, {"CHF", "Don", 2}}

Sample expression: {{1,2}, 2, 4 hours, "hey", 1999-10-23, 3 days, "why", "one"} where 1 is in it

Returns: {{1, 2}}

Sample expression: interval[2,3] where 2 is in it

Returns: interval[2,3]

Sample expression: interval[2,3] where 9 is in it

Returns: {}

Sample expression: {{1,2}, 2, 3, 4, "hey", 1999-10-23, 3 days} where is number(it)

Returns: {2, 3, 4}

Sample expression: {"a", "b", 3 days, 4 hours} where is number(it)

Returns: {}

Sample expression: {{1,2}, 2, 3, 4, "hey", 1999-10-23, 3 days, "why", "one"} where is string(it)

Returns: {"hey", "why", "one"}

Sample expression: {{1,2}, 2, 3, 4} where is string(it)

Returns: {}

Sample expression: {{1,2}, 2, 4 hours, "hey", 1999-10-23, 3 days, "why", "one"} where is duration it

Returns: {4 hours, 3 days}

Sample expression: {{1,2}, 2, 4 hours, "hey", 1999-10-23, 3 days, "why", "one"} where is time(it)

Returns: {1999-10-23}

Sample expression: {{1,2}, 2, 4 hours, "hey", 1999-10-23, 3 days, "why", "one"} where is list(it)

Returns: {{1,2}}

Sample expression: {true, false, unknown, 1, 1999-03-04T05:00:00, "a"} where is boolean(it)

Returns: {true, false, unknown}

Sample expression: {true, false, unknown, 1, 1999-03-04T05:00:00, "a"} where is unknown(it)

Returns: {unknown}

Numeric Interval

Operations supported on numeric intervals include inclusion and overlap comparisons. The values appearing within a numeric interval specification are real numbers with the exception of the special keywords –infinity and infinity. An interval is specified by using the keyword "interval" followed by "[" (to represent an inclusive lower bound) or "(" (to represent a non-inclusive lower bound), and two comma-separated numbers followed by "]" (to represent an inclusive upper bound) or ")" (to represent a non-inclusive upper bound). The number specified as the lower bound must be less than or equal to the number specified as the upper bound. Use of unsupported operators with numerical interval values is an error (causes a type mismatch exception to be raised).

6.3 *Binary Operators*

is in

Description: checks whether first argument occurs in the interval represented by the second argument

Sample expression: 1 is in interval((-1), 5)
Returns: true
Sample expression: (-10) is in interval((-50), (-2))
Returns: true
Sample expression: 5 is in interval(5, 29]
Returns: false
Sample expression: 5 is in interval[5, 29]
Returns: true

overlaps
Description: checks whether two numeric intervals overlap

Sample expression: interval[5,29) overlaps interval[26, 900]

Returns: true

Sample expression: interval(1, 50) overlaps interval(1, 50)

Returns: true

Sample expression: interval[3,5) overlaps interval(5, 99]

Returns: false

Duration Interval

Operations supported on duration intervals include inclusion and overlap comparisons. The values appearing within a duration interval specification are durations. An interval is specified by using the keyword "interval" followed by "[" (to represent an inclusive lower bound) or "(" (to represent a non-inclusive lower bound), and two comma-separated durations followed by "]" (to represent an inclusive upper bound) or ")" (to represent a non-inclusive upper bound). The duration specified as the lower bound must be less than or equal to the duration specified as the upper bound. Use of unsupported operators with duration interval values is an error (causes a type mismatch exception to be raised).

6.4 *Binary Operators*

is in

Description: checks whether first argument occurs in the interval represented by the second argument

Sample expression: 1 day is in interval((-1 day), 5 days)
Returns: true
Sample expression: (-10 years) is in interval((-50 years), (-2 years))
Returns: true
Sample expression: 5 hours is in interval(5 hours, 29 days]
Returns: false
Sample expression: 5 hours is in interval[5 hours, 29 days]
Returns: true

overlaps

Description: checks whether two duration intervals overlap

Sample expression: interval[5 minutes, 29 minutes) overlaps interval[26 minutes, 900 minutes]

Returns: true

Sample expression: interval(1 month, 50 months) overlaps interval(1 month, 50 months)

Returns: true

Sample expression: interval[3 seconds, 5 minutes) overlaps interval(5 minutes, 99 hours]

Returns: false

Absolute Date and Time Interval

Operations supported on absolute date and time intervals include inclusion and overlap comparisons. The values appearing within an absolute date and time interval specification are absolute dates and times. An interval is specified by using the keyword "interval" followed by "[" (to represent an inclusive lower bound) or "(" (to represent a non-inclusive lower bound), and two comma-separated absolute date and time values followed by "]" (to represent an inclusive upper bound) or ")" (to represent a non-inclusive upper bound). The absolute date and time specified as the lower bound must occur before or equal the absolute date and time specified as the upper bound. Use of unsupported operators with absolute date and time interval values is an error (causes a type mismatch exception to be raised).

6.5 *Binary Operators*

is in

Description: checks whether first argument occurs in the interval represented by the second argument

Sample expression: 1999-03-04 is in interval(1998-10-12, 2000-02-05T05:00:00)

overlaps

Description: checks whether two absolute date and time intervals overlap

Sample expression: interval(1998-10-12, 2000-02-05T05:00:00) overlaps interval(1998-10-12, 2000-02-05T05:00:00)

Returns: true

REFERENCES

1. Object Management Group. The Common Object Request Broker: Architecture and Specification; 1999. Report No.: OMG Document Number 91.12.1.

2. Bernstam E, Ash N, Peleg M, Tu S, Boxwala AA, Mork P, et al. Guideline classification to assist modeling, authoring, implementation and retrieval. In: Proceedings of the American Medical Informatics Association (AMIA) Annual Symposium,; 2000 November 2000; Los Angeles, CA,: American Medical Informatics Association; 2000. p. 66-70.

3. Advisory Committee on Immunization Practices A. Prevention and Control of Influenza. *Morbidity and Mortality Weekly Report* 2000;49(RR03):1-38.

4. Irwin RS, Boulet LS, Cloutier MM, Gold PM, Ing AJ, O'byrne P, et al. Managing Cough as a Defense Mechanism and as a Symptom, A Consensus Panel Report of the American College of Chest Physicians. *Chest* 1998;114(2):133S-181S.

5. American College of Cardiology/American Heart Association/American College of Physicians-American Society of Internal Medicine. Guidelines for the Management of Patients with chronic Stable Angina. *J Am Col Cardiol* 1999;33:2092-2197.

6. ACP-ASIM. Screening for Thyroid Disease. *Ann Int Med* 1998;129:141-143.

7. AHCPR. Acute Low Back Problems in Adults, Clinical Practice Guideline Number 14: AHCPR Publication No. 95-0642; 1994. Report No.: 95-0642.

8. AHCPR. Heart Failure: Evaluation and Care of Patients With Left-Ventricular Systolic Dysfunction, Clinical Practice Guideline Number 11: AHCPR Publication No. 94-0612; 1994. Report No.: 94-0612.

9. ACP-ASIM. Acute Major Depression and Dysthymia. *Ann Intern Med* 2000;132:738-742.

10. Peleg M, Boxwala AA, Tu S, Greenes RA, Shortliffe EH, Patel VL. Handling Expressiveness and Comprehensibility Requirements in GLIF3. In: to be published in Proc. MedInfo 2001; 2001; 2001.

11. Lindberg C. The Unified Medical Language System (UMLS) of the National Library of Medicine. *J Am Med Rec Assoc* 1990;61(5):40-42.

12. Schadow G, Russler DC, Mead CN, McDonald CJ. Integrating Medical Information and Knowledge in the HL7 RIM. In: Proc. AMIA Annual Symposium 2000; 2000; 2000. p. 764-768.

13. Peleg M, Ogunyemi O, Tu S, Boxwala AA, Zeng Q, Greenes RA, et al. Using Features of Arden Syntax with Object-Oriented Medical Data Models for Guideline Modeling. In: Submitted to AMIA Annual Symposium, 2001; 2001; 2001.

14. 8601 I, inventor Data elements and interchange formats - information interchange - Representation of dates and times. 1998.

15. National Kidney Foundation N. Clinical practice guidelines for peritoneal dialysis adequacy. New York; 1997.

16. American Society of Health-System Pharmacists A. Therapeutic guidelines for nonsurgical antimicrobial prophylaxis. *Am J Health Syst Pharm* 1999;56(12):1201-50.