THE CLINICAL ELEMENT MODEL
DETAILED CLINICAL MODELS

by

Joseph F. Coyle

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Biomedical Informatics

The University of Utah

May 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Joseph F. Coyle**

has been approved by the following supervisory committee members:

Stanley M. Huff	, Chair	**October 3, 2011** Date Approved
Scott P. Narus	, Member	**October 3, 2011** Date Approved
Paul D. Clayton	, Member	**October 3, 2011** Date Approved
Roberto Rocha	, Member	**October 3, 2011** Date Approved
Dennis L. Parker	, Member	**October 3, 2011** Date Approved

and by **Joyce A. Mitchell** , Chair of the Department of **Biomedical Informatics**

and by Donna M. White, Interim Dean of The Graduate School.

ABSTRACT

Detailed clinical models (DCMs) are the basis for retaining computable meaning when data are exchanged between heterogeneous computer systems. DCMs are also the basis for shared computable meaning when clinical data are referenced in decision support logic, and they provide a basis for data consistency in a longitudinal electronic medical record. Intermountain Healthcare has a long history in the design and evolution of these models, beginning with PAL (PTXT Application Language) and then the Clinical Event Model, which was developed in partnership with 3M. After the partnership between Intermountain and 3M dissolved, Intermountain decided to design a next-generation architecture for DCMs. The aim of this research is to develop a detailed clinical model architecture that meets the needs of Intermountain Healthcare and other healthcare organizations.

The approach was as follows:

1. An updated version of the Clinical Event Model was created using XML Schema as a formalism to describe models.
2. In response to problems with XML Schema, The Clinical Element Model was designed and created using Clinical Element Modeling Language as a formalism to describe models.
3. To verify that our model met the needs of Intermountain Healthcare and others, a desiderata for Detailed Clinical Models was developed.
4. The Clinical Element Model is then critiqued using the desiderata as a guide, and suggestions for further refinements to the Clinical Element Model are described.

A shooting star crosses the sky
And the one who watches,
and waits patiently,
will be delighted for their tomorrow to come.
Robyn M. Nelson (1995-2008)

**

Make a dent in the universe.
Steve Jobs (1955-2011)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to Scott Narus, Roberto Rocha, Paul Clayton, Dennis Parker, Stanley Huff, Tom Oniki, Yan Heras, Craig Parker, Beatriz Rocha, Harold Solbrig, Lee Min Lau, Peter Haug, Steve Schrank, Alan Rector, David Markwell, Sam Heard, Paul V. Biron, Wade Waters, Thomas Beale, Angelo Rossi Mori, and Allan Pryor.

CHAPTER 1

INTRODUCTION

Detailed clinical models (DCMs)[1] define the structure of clinical data, showing how the individual data elements relate to one another, and how the coded elements are bound to terminology concepts. They are the basis for retaining computable meaning when data are exchanged between heterogeneous computer systems. DCMs are also the basis for shared computable meaning when clinical data are referenced in decision support logic, and they provide a basis for data consistency in a longitudinal electronic medical record. Without knowing the logical structure of clinical data, it is impossible to query or use the data in application programs or decision logic.

Intermountain Healthcare has a long history in the design and evolution of these models, beginning with PAL (PTXT Application Language)[2] and then the Clinical Event Model[3, 4], which used Abstract Syntax Notation One (ASN.1)[5, 6] as a formalism and was developed in partnership with 3M. After the partnership between Intermountain and 3M dissolved, Intermountain began to design a next-generation architecture for DCMs.

One of the mistakes made by Intermountain in the past was to use a standard modeling language, but then alter it in such a way that no standard tools could work with this language. This was Intermountain's experience with ASN.1, which did not quite work in its standard form, so additional constructs were added to the language by Intermountain. This immediately meant that only Intermountain's compiler and tools would work with the ASN.1 source code definitions of our models. This defeated the purpose of using a standard modeling language in the first place. The question then arises as to why use a standard modeling language which carries the baggage of generalism (being able to model anything), when you gain none of the benefits.

For the next generation Clinical Event Model, we decided to use a standard modeling language but not alter the language, to guarantee we could work with standard tools. XML Schema[7, 8] was chosen to be the formalism, and instances would exist as XML.[1]

This initial attempt using XML Schema as a formalism is described in Chapter 2. Using XML Schema as a modeling formalism had an unexpected impact on modelers. There was a realization that the verbosity and complexity of XML Schema made the models difficult to comprehend and write, especially for some of the seemingly simple constructs we wanted to represent. We decided to create an XML Schema shorthand language that we could use to generate the final XML Schemas. This new language was named Schemita, using the Spanish diminutive form of Schema.

As time went on, internal discussions raised the point that it was not wise to limit our data instances to only XML. This was mainly driven by the concern that our XML instances were ten times bigger in size than a corresponding ASN.1 instance[9],[2] and how this would affect bandwidth and storage requirements. Arguments were made that data instances could exist as ASN.1 instances or even as Java object instances, and the Schemita could be used to generate the model definitions in these different languages. Moreover, even if we were only using XML for instance data, there was still the question about different structural forms of XML. We had already discovered that it is useful to have more than one XML instance structure for different uses such as for storage versus for the developer to use in user interfaces or decision support. Which Schema would be the master?

Eventually, as the model evolved (solving problems as they occured), it became known as the Clinical Element Model, and the Schemita syntax became known as Clinical Element Modeling Language or CEML, thus breaking the direct tie to XML Schema. In the following chapters, we will describe the Clinical Element Model, CEML, and some of the work using this model.

---

[1] It was also a consideration to move to an XML database to store data instances, but the immaturity of this technology and the fact that Intermountain engineers were uncomfortable with the unknown performance led to the decision to continue using a relational database.

[2] These measurements were informal comparisons between Intermountain's ASN.1 instances and their XML counterparts.

Creating a new syntax may seem like a complete reversal of our initial goal to use a standard modeling language and avoid changes to that syntax. But it was found that this extreme was cumbersome, and in the end found that using a simple and easily parseable specific language was the ideal balance that allowed modelers to build the models, and engineers to build the tools.

The aim of this research is to develop a detailed clinical model architecture that meets the needs of Intermountain Healthcare and other healthcare organizations.

After the development of the Clinical Element Model, in order to determine whether the development efforts had addressed the requirements for DCMs, a desiderata for DCMs was developed from both our experience and from literature review. This desiderata is then used to critique the Clinical Element and discover areas where the model is insufficient. These discoveries will be used to further evolve the Clinical Element Model to the meet the needs of all users.

In summary, our approach was as follows:

1. An updated version of the Clinical Event Model was created using XML Schema as a formalism to describe models.
2. In response to problems with XML Schema, The Clinical Element Model was designed and created using Clinical Element Modeling Language as a formalism to describe models.
3. To verify that our model met the needs of Intermountain Healthcare and others, a desiderata for Detailed Clinical Models was developed.
4. The Clinical Element Model is then critiqued using the desiderata as a guide, and suggestions for further refinements to the Clinical Element Model are described.

1.1 Importance to Biomedical Informatics

Detailed Clinical Models and the Clinical Element Model benefit the Biomedical Informatics community at many levels.

At the enterprise level, Intermountain Healthcare benefits because the models create a common representation that the many systems within the enterprise can share. From departmental systems to centralized systems, all developers are working with a common

representation of each medical concept. Even within the central system, different systems such as decision support, the longitudinal repository, and user applications can be developed with a common view of patient data. Without this consistency, each system would need to expend redundant manpower to develop their own internal data models, and ultimately, interfaces would need to be built to share data between these systems. Moreover, each system would evolve independently and interfaces would need to be constantly updated to continue the flow of data between these disparate systems. On the other hand, with a common set of detailed clinical models, these separate systems can evolve together as one.

When the enterprise needs to share data with other institutions, DCMs again simplify the situation. When the interface team receives external data, they only need to map data to one internal representation. And when they need to send data externally, they will only need to export one representation of the data.

At the national and even international level, DCMs will allow the Biomedical Informatics community to develop a curated model repository where models can be shared by all. To achieve this goal, the various organizations involved will need to decide on a common DCM syntax to describe models. This activity will force the community to settle on a set of models that meet the needs of all participants. Ultimately, with a single representation in the international DCM repository, individual institutions will be able to build transformations between these models and their local models, and the national DCM could become the common format which could allow sharing of data, applications, and clinical knowledge between systems.

CHAPTER 2

STANDARDS FOR DETAILED CLINICAL MODELS AS THE BASIS FOR MEDICAL DATA EXCHANGE AND DECISION SUPPORT[1]

2.1 Abstract

Introduction: Detailed clinical models are necessary to exchange medical data between heterogeneous computer systems and to maintain consistency in a longitudinal electronic medical record system. At Intermountain Health Care, we have a history of designing detailed clinical models. The purpose of this paper is to share our experience and the lessons we have learned over the last 5 years. *Design:* Intermountain's newest model is implemented using eXtensible Markup Language (XML) Schema as the formalism, and conforms to the Health Level Seven (HL7) version 3 data types. The centerpiece of the new strategy is the Clinical Event Model, which is a flexible name-value pair data structure that is tightly linked to a coded terminology. *Discussion:* We describe Intermountain's third-generation strategy for representing and implementing detailed clinical models, and discuss the reasons for this design.

2.2 Introduction

Detailed clinical models are the basis for retaining computable meaning when data are exchanged between heterogeneous computer systems. Detailed clinical models are also the basis for shared computable meaning when clinical data are referenced in decision support logic, and they provide a basis for data consistency in a longitudinal electronic medical

---

[1]Reprinted with permission from the International Journal of Medical Informatics, 2003. Coyle JF, Mori AR, Huff SM. Standards for detailed clinical models as the basis for medical data exchange and decision support. 69 (2003) 157-174.

record. Exactly, what we mean by "detailed clinical models" and how they relate to the use of clinical data by computers will be described below.

There are a number of motives for exchanging clinical data between heterogeneous computer systems. Data can be exchanged between different computers within a facility or enterprise in order to make information available to clinicians at the point of care, with the goal of improving clinical decision making. Serology and culture results can be sent from clinical laboratories to public health departments as a means of detecting an epidemic or a bioterrorism attack [10]. Healthcare providers that are participating in clinical trials of new medications or other therapies need to exchange clinical data as part of the research protocols [11]. In all of these situations, the goal is not just to have the data available for humans to read and understand, but to have the data structured and coded in a way that will allow computers to understand and use the information.

The most common strategy for representing data that are sent between different computer systems is to send the data as name-value pairs (also known as entity-attribute-value triplets) [12]. For example, if the results of a hematocrit test were to be sent between two systems, the data could be represented as

- Test name = Hematocrit, Value = 44.1%

The Health Level Seven (HL7) and Digital Imaging and Communications in Medicine (DICOM) standards [13, 14] use this strategy, and the Logical Identifier Names and Codes (LOINC) coding scheme was created in order to supply the "name" part of the name-value pair [15]. The use of standardized codes (and their associated computable definitions) as the names of test results allows computers to use the information in decision logic, outcomes research, and other clinical calculations.

Single-valued measurements like a hematocrit result are represented easily in a single name-value pair as shown above. However, as soon as the clinical measurement is slightly more complicated, variations in how the data can be represented present themselves. For example, there are at least three ways of representing the results of patellar deep tendon reflexes:

- A single name/code and value
    - Left patellar deep tendon reflex intensity is 2+

- Combination of two names/codes and values
    - Patellar deep tendon reflex intensity is 2+
    - Laterality is left
- Combination of three names/codes and values
    - Deep tendon reflex intensity is 2+
    - Body part is patella
    - Laterality is left

When the complex nature of the data allows these different options for representation, it is important to understand that these representations are equivalent, otherwise a computer processing the data will not recognize the alternative forms and will fail to use the data appropriately. The ability of a computer to recognize the equivalence of these statements is based on an underlying detailed clinical model. For the example given, the detailed clinical model could be stated as:

- Type of measurement - (intensity of deep tendon reflex)
- Location of measurement - (patella, or patellar tendon)
- Laterality of measurement - (left side)

For a computer to recognize the equivalence of the three different statements, there must be a more formal way of stating the information model and of referencing standardized terminologies that are used for the names of data elements in the model. Use of standard models and associated standard reference terminologies will enable a computer to detect equivalent representations.

Many examples of alternative data representation exist. A more difficult example than patellar reflex data is the problem of lung auscultation. The results of lung auscultation can be represented either in a finding-focused style or a location-focused style. For example, one can state the finding "wheezing," and then state every lung location where it was heard, such as "wheezing in the right and left upper lobes." Alternatively, one could be location-focused, and state the location "right upper lobe," and state all the findings associated with that location such as "the right upper lobe has wheezing, rales, and egophony."

The need for a formal way of representing detailed clinical models is closely related to what has been termed the "curly braces problem." The curly braces problem arises from the

practical issues of trying to implement medical logic modules (MLMs) such as rules, alerts, and reminders using Arden syntax [16]. Arden syntax is an HL7 standard for representing medical decision logic. In Arden syntax, data slots are used to create read statements that retrieve data that participate in the logic of MLM from the patient's EMR (or some other data store). Curly braces are used within the read statement "to isolate institution-specific portions (of data access) to one slot. Within the data slot, the institution-specific portions are placed in mapping clauses so that the institution-specific part does not interfere with the MLM syntax." The following snippet from an MLM shows the use of curly braces:

```
data:
        /*creatinine in mg/dl*/
        creatinine = read last {select value from lab where code = 237}
        . . .(more data declarations)
evoke
        /*execute this logic each time a new calcium is stored*/
        storage_of_calcium;
logic:
        /*if creatinine is present and greater than 6, then stop now*/

        IF creatinine is present THEN
                IF creatinine is greater than 6.0 THEN
                        conclude false
                ENDIF
        ENDIF
        . . .(more logic statements)
```

In this example, curly braces are used to enclose an SQL statement that would retrieve the patient's creatinine from a (hypothetical) relational database containing laboratory results. Pryor and Hripcsak [17] demonstrated that a major obstacle to sharing decision logic was creating the mapping from the logical data element in the read statement to the corresponding data in the local EMR. Detailed clinical models that incorporate standard coded terminologies are needed to overcome this problem. If these kinds of models existed and MLMs referenced the common models, then a given institution would still need to map from the shared models to its local EMR representations, but once the mapping was completed, MLMs from any source could be shared without further institution-specific mapping.

The need for detailed clinical models has been recognized by researchers and standard organizations. DICOM, Centre for European Normalisation (CEN), Good Electronic

Health Record (GEHR), HL7, GALEN, and Stephen Johnson have either developed or planned to develop a mechanism for describing and sharing detailed clinical models [13, 14, 18, 19, 20, 21, 22]. Our purpose in writing this paper is to describe a third-generation strategy for representing and implementing detailed clinical models that we are using at Intermountain Health Care, and discuss the lessons we have learned over the last 5 years. We are not proposing this model as the ultimate global solution, but the hope is that this work will contribute to the ongoing discussions about a global solution. This paper will describe our basic requirements and the details of the current model. In a subsequent paper, we hope to compare our model and approach in detail to the work of others working in this area.

2.2.1 Previous Work at Intermountain

Intermountain Healthcare, in partnership with 3M Health Information Systems, has developed 3379 detailed clinical models using Abstract Syntax Notation One (ASN.1) as the formalism [4]. Over the course of this development, many mistakes were discovered with the original design. The first-generation model was an inflexible model where subsequent changes required changes to the handling software or the underlying database, or both. In our case, the underlying system was implemented using a relational database, and each change or addition to the clinical model caused the addition of new columns or tables to the database, as well as requiring changes to the C++ classes that implemented the model. The problem with this type of strategy is that unless the initial modelers are omniscient, then the production system is doomed to failure because changes and additions to the model are very expensive to propagate into deployed systems. Realizing this, a second-generation model called the Event-Observation Model was designed [3]. All Observations were modeled by restriction from a common parent Observation. Each Observation contained a single value, plus modifiers of the value, which were represented as a sequence of Observations. Collections of Observations were stored in a type called Event. Despite the good intention of this design, certain attributes within Event and Observation were still modeled by the previous inflexible method. This was especially true of the attributes representing the who, what, where, why, and when pertaining to the Event. This inflexibility was later rectified by

adding modifiers to Event, and then requiring that all subsequent additions to the model be performed by adding modifiers to Observation and Event. Another basic problem with this model was that collections of Events could not be stored in another Event. This prohibited the design of deeply nested data types, such as physical exams, from existing Events.

Not only were mistakes made in the overall structure of the model, but modelers also made mistakes when they subclassed existing Events and Observations. During the subclassing procedure, modelers were able to restrict certain attributes to have a specific value. Unfortunately, different modelers had different ideas regarding the purpose of certain attributes. For example, originally, the Observation attribute called obsId (observation identifier) was modeled to store a code, which mapped a specific Observation restriction to its real-world counterpart. Unfortunately, since this information can be inferred from the ASN.1 type name itself, some modelers decided to use obsId for other uses. Thus, obsId began serving double duty, requiring software enhancement to understand when it had one meaning, and when it had another.

Another mistake was made in the way ASN.1 was used in the initial design. There were semantic constraints needed in the model that were not part of ASN.1, and so new ASN.1 constructs, such as Instance-Of-Concept, Instance-Of-Subtype, and WithTypes were created [4]. Thus, the resulting models could not be compiled with commercial ASN.1 tools, and we had to maintain our own ASN.1 tools.

A third-generation model, with eXtensibl.epse Markup Language (XML) Schema [7, 8] as the formalism, has been developed to correct the stated problems, and to meet all of our current requirements. The major requirements that we considered as we developed the new model are as follows:

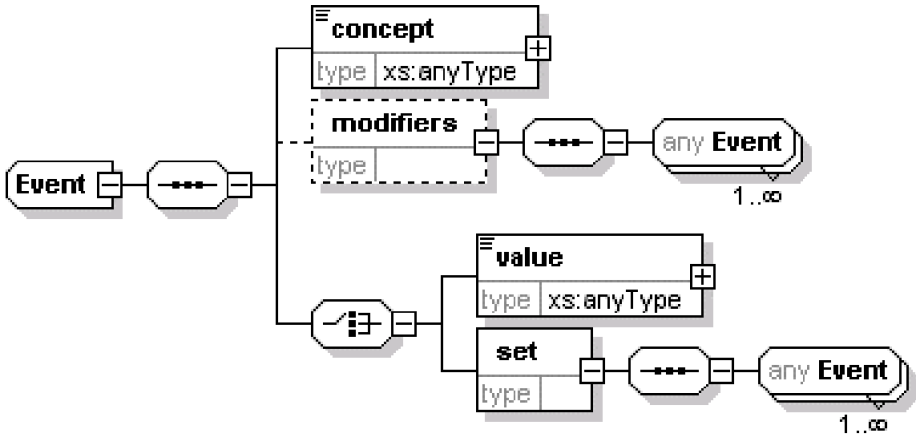
- The model must be comprehensive - it must accommodate representation of anything that can be stated about a patient.
- The model must be flexible and extensible - it must be possible to add elements and attributes to the model without requiring changes to the underlying software and database.
- It must use an existing formalism (XML Schema, ASN.1, Conceptual Graphs, etc.) without modification.

- There must be a tight linkage to standard terminologies such as LOINC, systematized nomenclature of medicine - clinical terms (SNOMED CT), HL7 Vocabulary Tables, etc.
- There must exist a mechanism to state negation, in order to say that something was NOT observed or was NOT present.
- A process for change management must be followed in order to know which version of a model was in effect at the time data were stored.
- There is a need to easily change the cardinality of values, e.g., to note a single complication versus selecting all the complications that apply.
- There must exist the ability to allow any degree of arbitrary collections and batteries. It must be possible to retain as part of the permanent record how the data were originally seen by the user, or as they were sent by an application.

2.3 A Formal Model

The root of the new implementation is an abstract type called Event (Figure 2.1). All types are a restriction of Event. Currently, Event has only one subtype, ClinicalEvent, from which all children are derived. Event was created as a general nonmedical class in the event that we want to use this design for nonclinical data. Event is a recursive design that allows the construction of complex deeply nested data types. It consists of the elements concept, modifiers, and a choice between value or set, and contains the attributes instanceIdentifier and contextControl (Table 2.1). The constructs concept, value, instanceIdentifier, and contextControl are terminal constructs, while modifiers and set are recursive, containing a sequence of Event.

The abstract type ClinicalEvent (Figures 2.2 and 2.3) is the restriction of Event that brings the design into the medical domain, and is the basis for the rest of the model. In this restriction, all values are restricted to be HL7 version 3 data types [23], and the most universal of the modifiers are applied (Table 2.2). Modifiers are themselves restrictions of ClinicalEvent, and thus can contain modifiers, but frequently, these are restricted out during modeling (Figure 2.4).



Attribute	Type	Use	Default
instanceIdentifier	xs:ID	required	
contextControl	xs:NMTOKEN	optional	normal

Figure 2.1. The abstract type Event contains the elements concept and modifiers, with a choice between value or set, and contains the attributes instanceIdentifier and contextControl.

Table 2.1. The elements and attributes of Event

Property	Type	Description
Concept	Element	A data element which names or denotes the data contained in either value or set
Modifiers	Element	Additional coded items (modeled as a sequence of Events) which modify or further describe the data contained in either value or set
Value	Element	Value of the data element named by concept
Set	Element	A container (modeled as a sequence of Events) to allow creation of collections of nested data types, as named by concept
instanceIdentifier	Attribute	Global unique identifier (GUID) for instances of data stored in the Event structure; it is used like a primary key in a database
contextControl	Attribute	Controls context of modifiers and AttributionInfo inheritance to children in a set

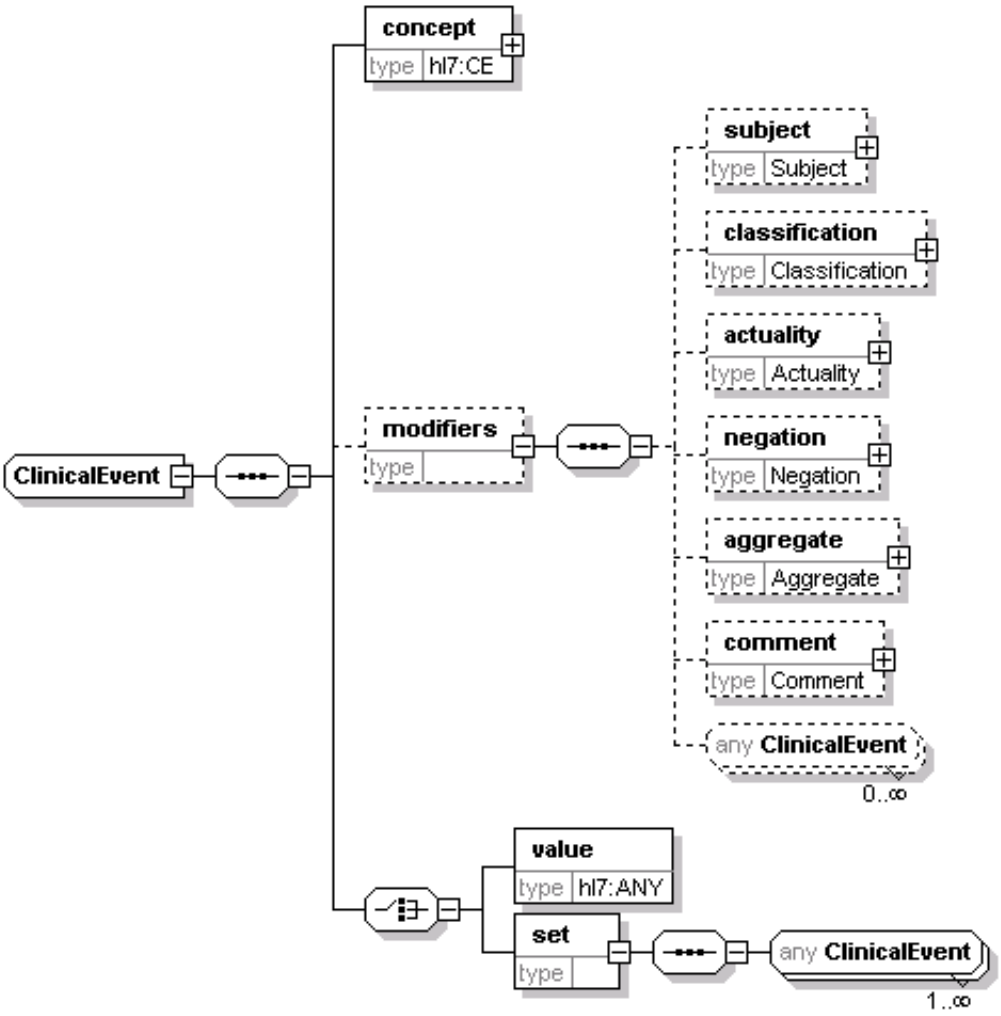


Figure 2.2. The abstract type ClinicalEvent.

```
<xs:complexType name="ClinicalEvent" abstract="true">
 <xs:complexContent>
  <xs:restriction base="Event">
   <xs:sequence>
    <xs:element name="concept" type="hl7:CE"/>
    <xs:element name="modifiers" minOccurs="0">
     <xs:complextType>
      <xs:sequence>
        <xs:element name="subject" type="Subject" minOccurs="0"/>
        <xs:element name="classification" type="Classification" minOccurs="0"/>
        <xs:element name="actuality" type="Actuality" minOccurs="0"/>
        <xs:element name="negation" type="Negation" minOccurs="0"/>
        <xs:element name="aggregate" type="Aggreate" minOccurs="0"/>
        <xs:element name="comment" type="Comment" minOccurs="0"/>
        <xs:any namespace="ClinicalEvent" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complextType>
     </xs:element>
     <xs:choice>
      <xs:element name="value" type="hl7:ANY"/>
      <xs:element name="set">
      <xs:complextType>
        <xs:sequence>
         <xs:any namespace="ClinicalEvent" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complextType>
     </xs:element>
    </xs:choice>
   </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complextType>
```

Figure 2.3. XML Schema for the abstract type ClinicalEvent.

Table 2.2. The universal modifiers of ClinicalEvent

Name	Default	Description
Subject	Self	Describes the subject of the data; examples are Self, Family Member, Donor, and Fetus
Classification		Sequence of categories into which these data fit; examples are Diagnosis and Procedure
Actuality	Actual	Describes the existence of the data; examples are Actual and Hypothetical
Negation	Not negated	Negates the value; the only legal value is Negated
Aggregate		A coded value which is a synonym for all the data contained in the ClinicalEvent
Comment		A recursive comment container to record coded and textual comments

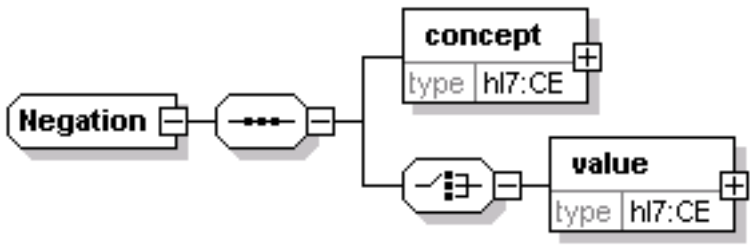


Figure 2.4. The type Negation is an example of a universal modifier which has been restricted to contain no modifiers.

ClinicalEvent is the parent of all further derived types. Subtypes can either be declared abstract, as is ClinicalEvent, or they can be instantiating types, from which XML instance documents can be created. An abstract type is beneficial when the same list of modifiers is useful for many other subtypes. Subtypes can then derive from this abstract type, inheriting the abstract type's modifiers, while adding modifiers specific to the subtype. When an instantiating type is modeled, the sequence of ANY constructs are restricted out of modifiers and set, prohibiting further subtypes from adding new modifiers and new Events to set.

Since ClinicalEvents can be deeply nested, rules for modifier context are required. The rules for modifier inheritance, from a parent (the enclosing event) to its children inside of set, depend on the value of the contextControl attribute in the modifiers. The default value is normal which is to be assumed in the following rules unless otherwise stated. The rules are as follows:

- A child event will inherit all instantiations of modifiers from its parent chain if those modifiers exist as a possibility within the child, except if the contextControl attribute has been set to block in the parent modifier. For example, in the type BloodPressure (Figure 2.5), say that the modifier BodyLoc is instantiated with a value at the level of BloodPressure and contextControl is set to normal. Then, both SystolicBloodPressure and DiastolicBloodPressure would inherit that BodyLoc value. But, if contextControl is set to block, then SystolicBloodPressure and DiastolicBloodPressure would not inherit the BodyLoc value.
- If the same modifier is instantiated multiple times along the parent chain, the most proximal self or parent instantiation to the child will apply to the child. However, if the contextControl attribute has been set to additive in the child modifier, all modifiers in the parent chain will apply. For example, in the type BloodPressure (Figure 2.5), say that the modifier BodyLoc is instantiated with a value at the level of BloodPressure and at the level of SystolicBloodPressure and contextControl is set to normal. Then, SystolicBloodPressure would use its own BodyLoc value, and ignore the value in BloodPressure. But DiastolicBloodPressure would inherit the BodyLoc value from its most proximal parent, which is BloodPressure. Furthermore,

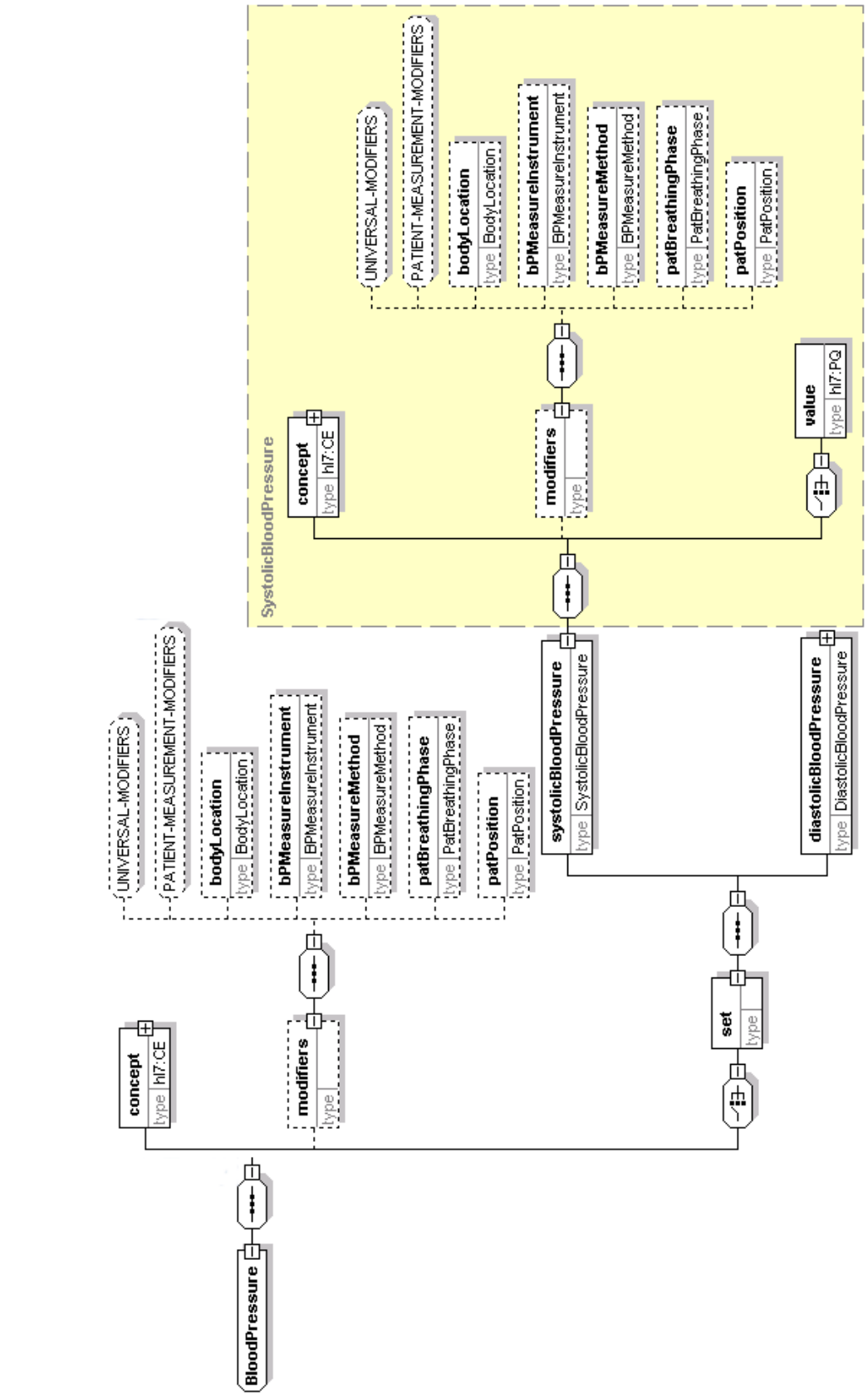


Figure 2.5. The type BloodPressure.

if contextControl is set to additive (which does not make sense in this case), then SystolicBloodPressure would use its own BodyLoc value and it would also inherit the BodyLoc value from BloodPressure. DiastolicBloodPressure would inherit the BodyLoc value as before, from BloodPressure.

All the elements within the ClinicalEvent structure are either a subtype of ClinicalEvent, or a subtype of the HL7 version 3 data type ANY [24]. Leaf nodes in a ClinicalEvent type always terminate with an HL7 data type, and are restricted in subtypes according to the rules of schema restriction (Figure 2.6) [7, 8]. The exception to this involves restriction of the HL7 coded types which are subclasses of hl7:CD. The HL7 coded types do not have an attribute or element that corresponds to domain [24], and so it is impossible to restrict the schema to have a fixed domain value.

Instead, the ClinicalEvent Model stores the domain value within xs:appinfo [7], which will make it available to the parser for external domain validation (Figure 2.7).

SemanticLink (Figure 2.8) is a subclass of ClinicalEvent, and is comprised of a set containing three ClinicalEvents. SemanticLinkSource and SemanticLinkDestination store the instanceIdentifiers of the source and destination ClinicalEvents. SemanticLinkRelationship

```
<xs:element name="value">
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="hl7:PQ">
        <xs:attribute name="value" use="required">
          <xs:simpleType>
            <xs:restriction base="hl7:int">
              <xs:minInclusive value="0"/>
              <xs:maxInclusive value="500"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="unit" use="optional" fixed="mmHg"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Figure 2.6. The restriction of the HL7 type PQ to have values between 0 and 500 and the units mmHg (millimeters of mercury).

```
<xs:element name="value" type="hl7:CE">
  <xs:annotation>
    <xs:appInfo>
      <domain>12345</domain>
    </xs:appInfo>
  </xs:annotation>
</xs:element>
```

Figure 2.7. Example of restricting an HL7 coded type to the domain of 12345.

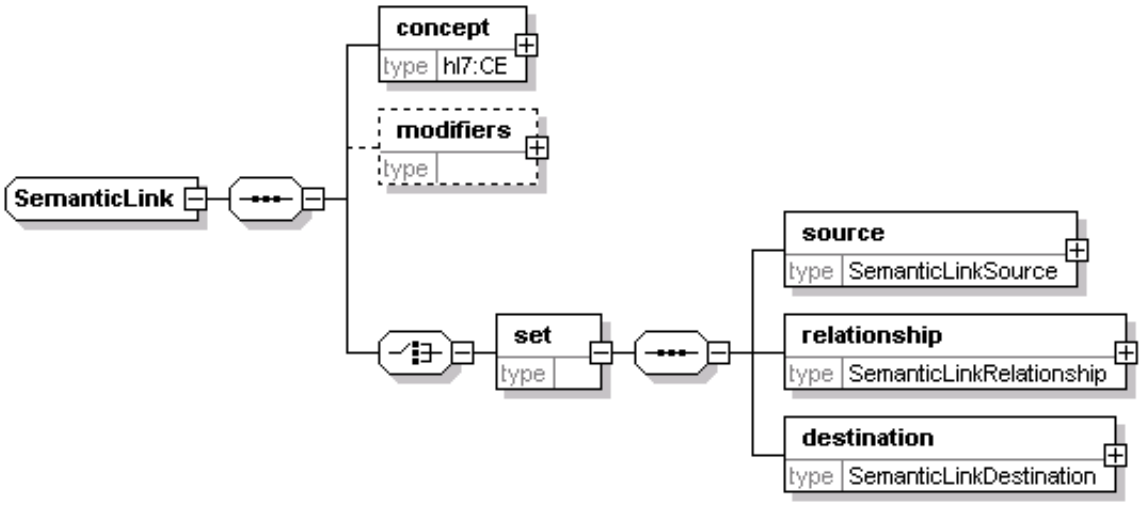


Figure 2.8. The type SemanticLink.

contains a coded relationship value, such as pertains to, caused by, etc., which correspond to the Link Types described in CEN ENV 13606 [18, 19].

AttributionInfo (Figure 2.9) is a subclass of ClinicalEvent, and is comprised of a sequence of AttributionItem. Each AttributionItem contains elements action, actor, reason, time, and location.

PatientEMR (Figure 2.10) is a subclass of ClinicalEvent, and is the container for a sequence of instances of ClinicalEvent, functioning as the patient's electronic medical record.

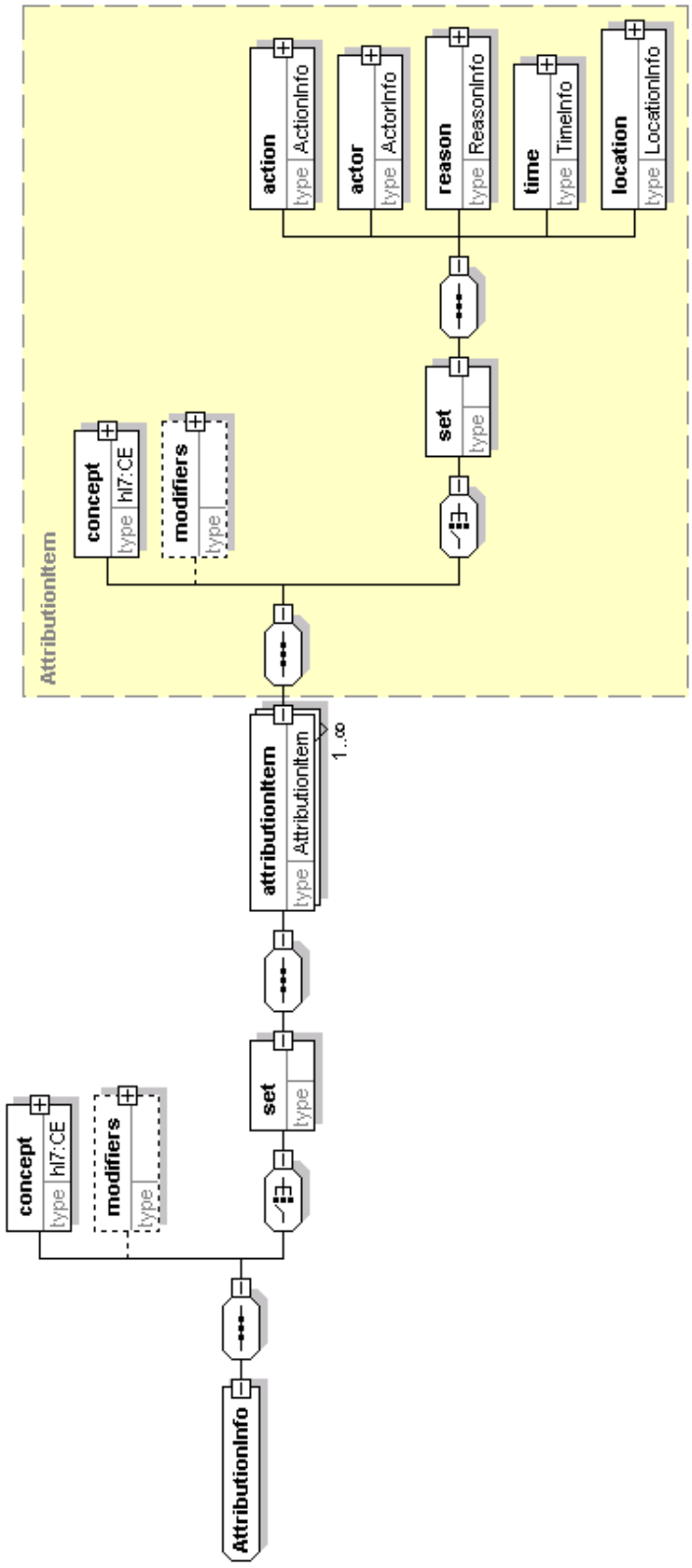


Figure 2.9. The type AttributionInfo.

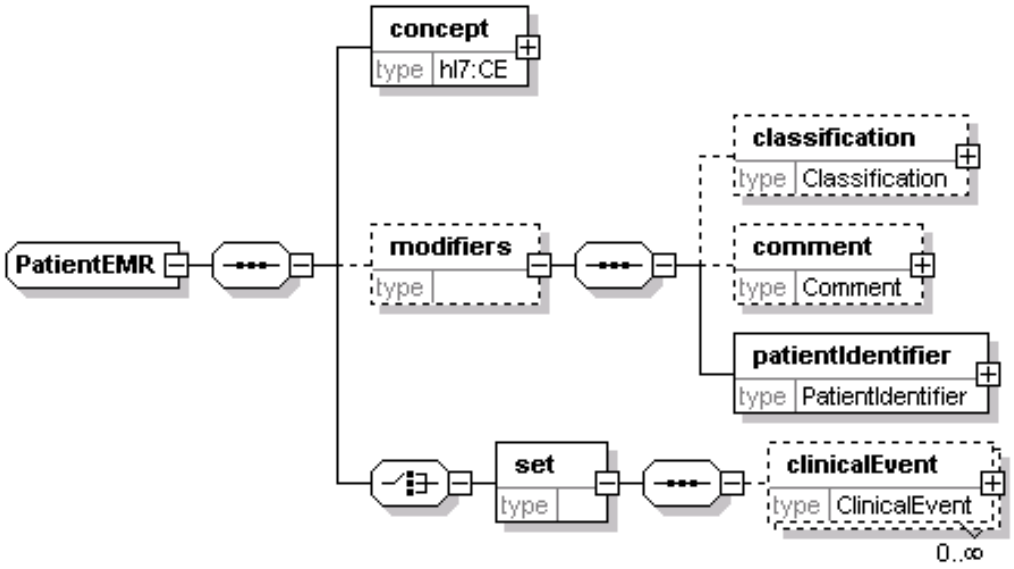


Figure 2.10. A patient's medical record is stored as a sequence of Clinical Events.

2.3.1 Example Use of the Model

The process of creating complex nested types is best illustrated by an example, and for this purpose, a vital signs battery will be modeled and named VitalSigns. Anticipating the creation of many kinds of measurements, we first create PatientMeasurement as an abstract type, and this will be the basis for creating all patient measurements (Figure 2.11). In this restriction, general modifiers for all patient measurements are added. Since this is an abstract type intended for further restriction, the sequence of ANY constructs remains in modifiers and set.

Next, the elements of VitalSigns are created, which include systolic and diastolic blood pressures, heart rate, temperature, and respiratory rate. These types are modeled as sub-types of PatientMeasurement. Provided is an example of a systolic blood pressure (Figure 2.12). Modifiers specific to a blood pressure measurement are added, and the choice between value and set is restricted to value, and specifically to an HL7 PQ (physical quantity). PQ is restricted to have an integer value between 0 and 500, with the unit mmHg (millimeters of mercury) (Figure 2.6).

Once all the types that comprise a vital signs battery are modeled, they can be placed within the set of the new VitalSigns type. But, since systolic and diastolic blood pressures

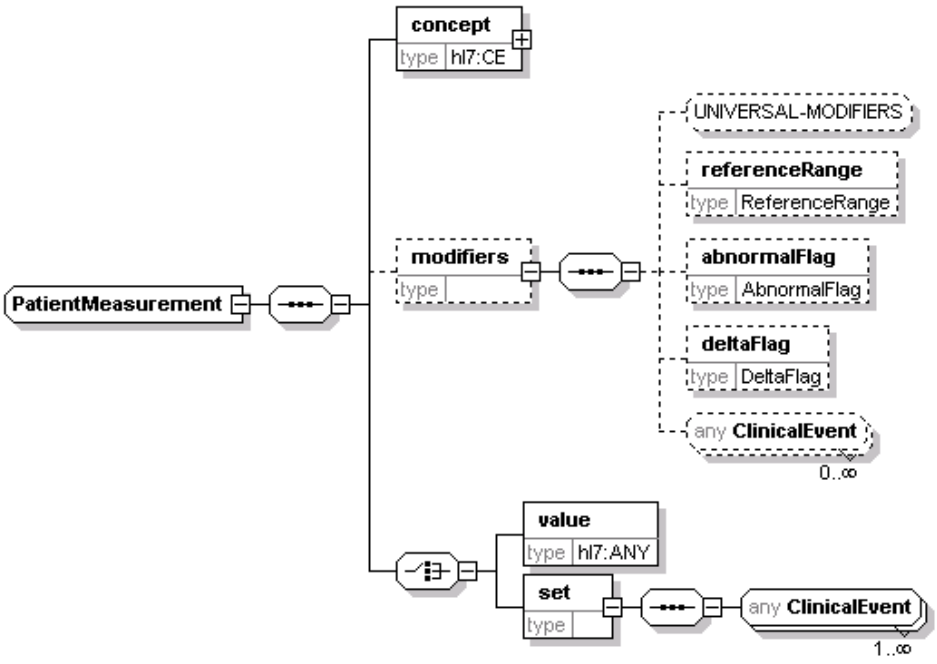


Figure 2.11. The abstract type PatientMeasurement.

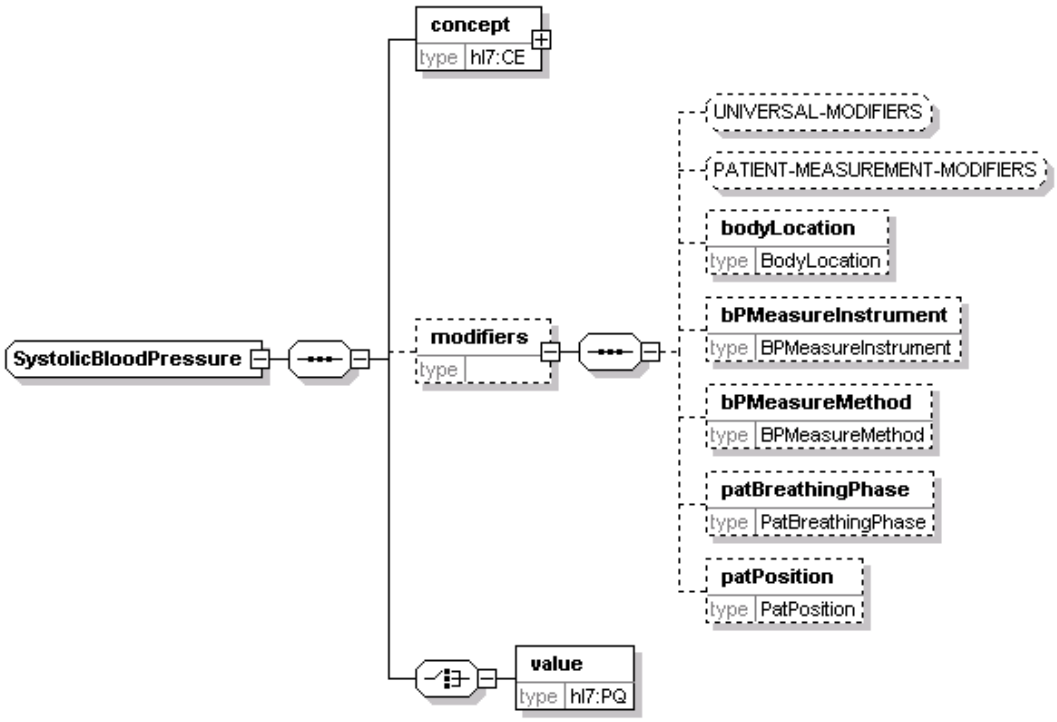


Figure 2.12. A subtype of PatientMeasurement called SystolicBloodPressure.

usually share context, it makes sense to make a BloodPressure type to contain them. This will also allow us to demonstrate the nesting capabilities of the model and to demonstrate modifier context rules. The type BloodPressure is modeled as a set of SystolicBloodPressure and DiastolicBloodPressure (Figure 2.5). Notice that both BloodPressure and SystolicBloodPressure are created with the same modifiers. This allows instantiations to put modifier data at the BloodPressure level or the SystolicBloodPressure level, as appropriate.

The VitalSigns example is completed by adding the new BloodPressure type, along with Temperature, HeartRate, and RespiratoryRate to a set within the type VitalSigns (Figure 2.13). VitalSigns could then be nested inside of still larger constructs such as a complete physical exam.

As shown by this example, subtypes of ClinicalEvent are usually detailed models of clinical data. However, the ClinicalEvent structure is flexible enough that it can also be used to create folders, allowing users to store arbitrary collections of existing clinical events. These folders are not to be confused with directories in a computer file system; they are simply another subtype of ClinicalEvent. A folder is an instance of ClinicalEventFolder (Figure 2.14) which contains a set of instance identifiers from instantiated ClinicalEvents already within the data store. This gives the users the ability to group existing data as they see fit, while not affecting the storage of that data.

2.4 Discussion

In the Clinical Event Model, the patient's electronic medical record is viewed as a series of ClinicalEvents (Figure 2.15). A ClinicalEvent can be an individual laboratory result, a drug order, or even a collection of data as complex as a complete physical exam. ClinicalEvents are recursive, and thus can be deeply nested to create complex constructs such as a complete physical exam.

Each ClinicalEvent is linked to one AttributionInfo structure, which stores the who, what, where, why, and when information concerning the data stored in the ClinicalEvent. A few examples of the information contained in AttributionInfo are the healthcare worker who made an observation, the time of the observation, the time the observation was stored

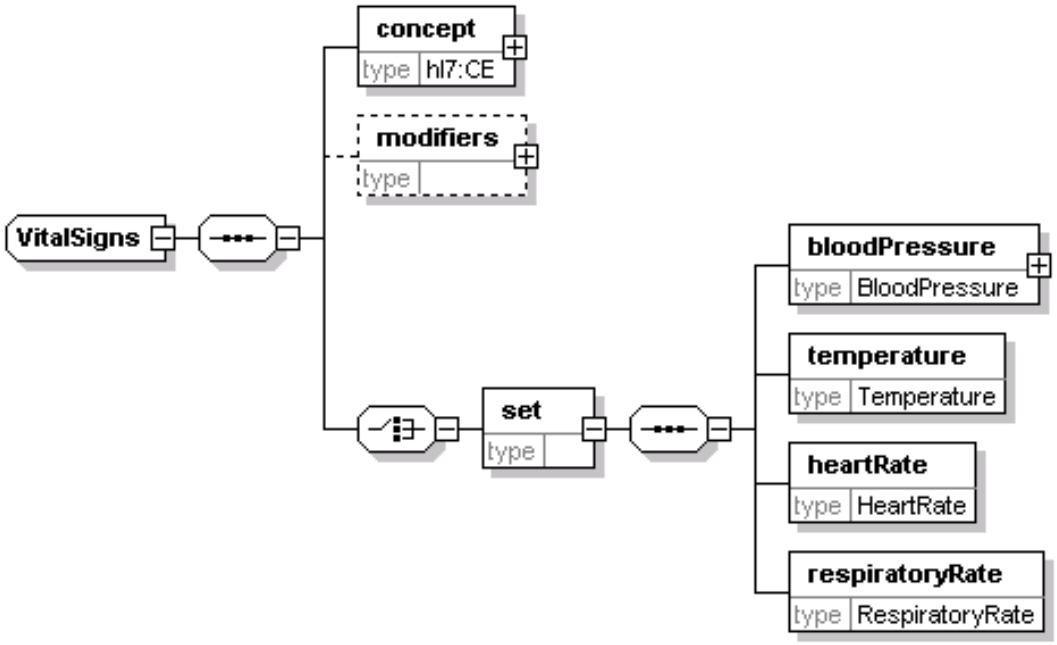


Figure 2.13. The type VitalSigns.

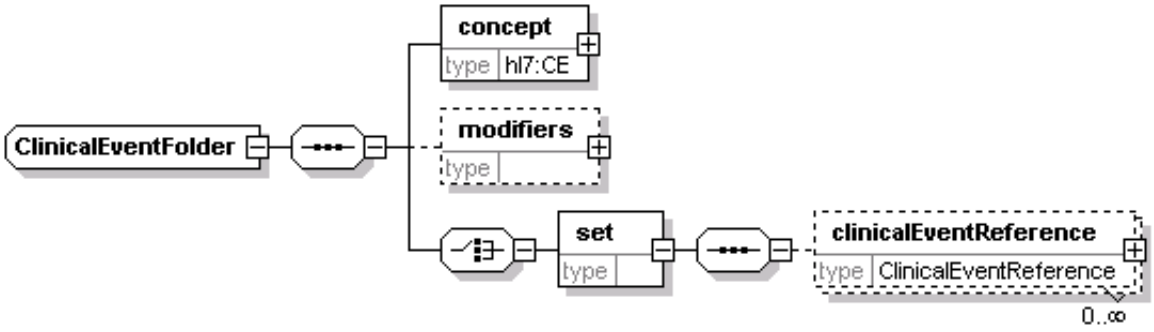


Figure 2.14. Collections of existing ClinicalEvents can be organized by reference within a type such as ClinicalEventFolder.

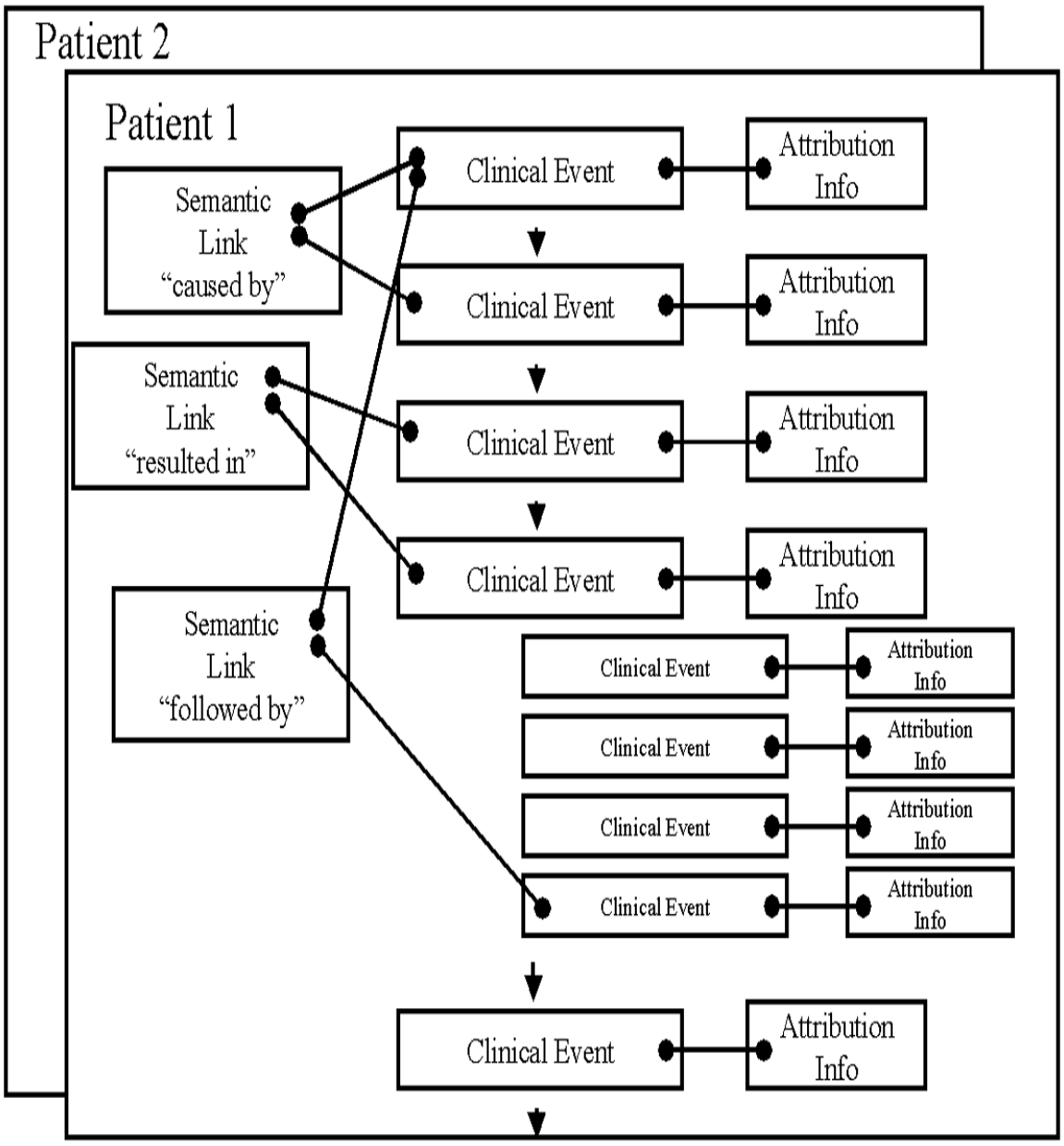


Figure 2.15. The electronic patient's record viewed as a series of Clinical Events.

as a ClinicalEvent, the software that stored the ClinicalEvent, and any updated history information.

The Clinical Event Model also contains a structure called SemanticLink, which allows a ClinicalEvent to be semantically linked to another ClinicalEvent, through a named relationship. For example, a ThroatCulture ClinicalEvent that was positive for *Streptococcus pyogenes* may be linked to an Order ClinicalEvent for Penicillin, through the relationship resulted in. Each SemanticLink is a one-to-one relationship, but any ClinicalEvent can be linked multiple times as either the relationship source or destination, thus creating many-to-many relationships. Not only can SemanticLinks link ClinicalEvents within a patient's record but they can also link from one patient's record to another. For example, a microbiology culture ClinicalEvent may link to a diagnosis of a nosocomial infection ClinicalEvent, which may link to a culture ClinicalEvent for each patient in the hospital.

It should be emphasized that SemanticLinks and AttributionInfo are themselves subclasses of ClinicalEvent. And, although not shown in the diagram (Figure 2.14), SemanticLinks have an attribution link to their own AttributionInfo structure like other ClinicalEvents. AttributionInfo, although a ClinicalEvent, is restricted from having an attribution link to another AttributionInfo to prevent unnecessary recursion.

In this third-generation model, we have removed all the inflexible attributes from the previous model, and have replaced them with modifiers or as part of AttributionInfo. In fact, when you examine a deeply nested ClinicalEvent structure (Figure 2.5), it is easily seen that every element is either a subtype of ClinicalEvent, or is a subtype of the HL7 version 3 data type, ANY. It is the leaf nodes in our recursive model, which always terminate with an HL7 value. Thus, the Clinical Event Model allows us to semantically organize HL7 values with a very simple design, allowing for very easy database modeling and software maintenance. For example, there is only a need for one JAVA class to represent all ClinicalEvent restrictions. And, by conforming to the HL7 values, it will be easy to transform data to and from HL7 messages to communicate with other systems.

The new ClinicalEvent type now serves both the combined function of our previous Event and Observation. By combining this functionality, any ClinicalEvent can now be nested within another ClinicalEvent to create deeply nested detailed clinical models. Pre-

viously, once an Event was modeled, that became the absolute top of the type and it could not be reused in other Events.

With a solid structure and formalism in place, we set out to correct the confusion that had developed over time with the existing attributes such as obsId. The reason modelers were misusing obsId is because they did not have the attributes they needed to express their needs. First, we renamed obsId with the name concept, to more accurately reflect its purpose. Next, we examined the mistakes that had occurred over time. Frequently, the value of a categorizing domain was placed in obsId such as Diagnosis in a DeliveryComplicationObs. To allow modelers to represent this information, we created the universal modifier Classification, which is a set of Class. This allows modelers to represent as many categories as they need for each type, and leaves concept free for its intended purpose. Another frequent value of obsId was Donor or Family History, which was used to represent results from others, yet stored in the patient's electronic medical record. To solve this problem, we added the universal modifier Subject which can take values such as patient/self, mother, baby, donor, or potential donor.

In this new model, the decision to change our modeling language from ASN.1 to XML Schema was based upon several factors. First, XML instance data are both human- and machine-readable where ASN.1 instance data are only machine-readable. Second, the abundance of tools and database options for XML will aid in development and deployment of our design, and we expect XML support to continue to grow. XML also integrates easily with web applications which use HTML. And finally, XML is the implementation choice for HL7 version 3 [24], to which our new model conforms at the base data type level. The only drawback we see with XML versus ASN.1 is that the instance data are larger for XML than for ASN.1, which means more network traffic and more data storage space is needed. We have made the decision never to alter the semantics of XML Schema, as we did using ASN.1. Once a standard is altered, it is no longer a standard, and the benefits of global development disappear. When we altered ASN.1, we developed our own C ASN.1 tools, which made it impossible to easily move to another programming language. It is only recently that we have begun the laborious process of recreating these tools in JAVA.

Our experience would indicate that a particular style of modeling will be more intuitive for a given purpose than an alternative style. Using different styles for a given type of information is all right as long as synonymy is recognized. Synonymy (or isomorphism) of two models can be determined if the codes used in the model are related by a terminology reference information model like SNOMED CT. In a repository of detailed clinical models, a common identifier that represents a family of isomorphic models could link isomorphic types.

The value of these detailed clinical models will not be realized unless there is a common notation for them, and there is a mechanism for sharing. The most logical place for sharing would be a standard organization like HL7 or CEN, where an open source style of repository could be created.

The next step in our development is to remodel all the existing ASN.1 Event types into the new XML Clinical Event Model. During this process, a value map will be generated between the old and the new models. This map will be used to translate the existing ASN.1 instance data into the new XML instances. The new XML instances will then be loaded into both an XML-enabled relational database and a native XML database for performance testing.

2.5 Acknowledgements

We would like to thank Scott Narus, Roberto Rocha, Paul Clayton, Harold Solbrig, Lee Min Lau, Steve Schrank, Alan Rector, David Markwell, Sam Heard, Paul V. Biron, and Thomas Beale for many fruitful discussions related to the subject of this paper.

CHAPTER 3

CLINICAL ELEMENT MODEL DEVELOPMENT

3.1 Overview

In order to represent detailed clinical data models, we have designed the Clinical Element Model (CEM). When we state "the Clinical Element Model" we are referring to the global modeling effort as a whole, or in other words, our approach to representing detailed clinical data models and the instances of data which conform to these models. The Clinical Element Model is the combination of an Abstract Instance Model and an Abstract Constraint Model. The Abstract Instance Model defines a structure to represent instances of medical data, and the the Abstract Constraint Model defines constraints on values in the Abstract Instance Model (Figure 3.1).

The Abstract Instance Model is a structure which can represent individual instances of collected data; for example, the Systolic Blood Pressure measurement collected on John Doe, on May 13, 2007, at 2:45 P.M. The values in this Systolic Blood Pressure data must conform to the constraints or rules stated in the corresponding Constraint Model for Systolic Blood Pressure.

Both the Abstract Instance Model and the Abstract Constraint Model are described abstractly; thus, they must both be implemented using an Implementation Technology Specification (ITS). Later in this document, we will present an implementation of both the Abstract Instance Model and the Abstract Constraint Model using XML, but it should be understood that a different ITS could use XML, Java, C, or Objective-C. And each of these abstract models could use a different ITS. For example, the Abstract Instance Model could be implemented in Java, and the Abstract Constraint Model Could be implemented in XML. For those familiar with Clinical Element Modeling Language(CEML), CEML is an implementation of the Abstract Constraint Model using XML as a syntax (Figure 3.2).

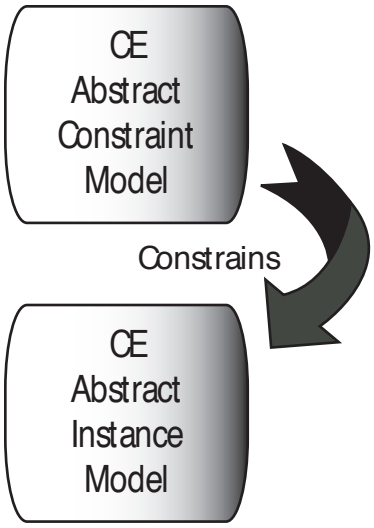


Figure 3.1. CE Abstract Constraint Specification describes the constraints on the CEM Abstract Specification Instance.

In the following sections, I will define each of these these abstract models, which together make up what is called the Clinical Element Model.

3.2 CE Abstract Instance Model

In this section, we will examine the Clinical Element Abstract Instance Model. The Abstract Instance Model defines the structure which is used to represent instances of medical data. An instance of medical data is created each time the patient has infomation added to the medical record. For example, if John Doe had 3 blood pressures taken, then 3 instances of medical data would be added to his medical record. Each of these instances would describe the details of a particular blood pressure. If John Doe then had a serum glucose measurement, then an instance describing this result would be added to the medical record. The patient's medical record thus becomes a collection of thousands of individual instances of medical data. The Abstract Instance Model describes the structure to represent all of these possible instances. We call these instances Clinical Element Instances. In this section, if the term Instance Model is used, we are referring to the Clinical Element Abstract Instance Model.

The Instance Model is a structure designed to hold instance data, and the structure does not change regardless of the type of instance data. Because of this, nonsense data can be

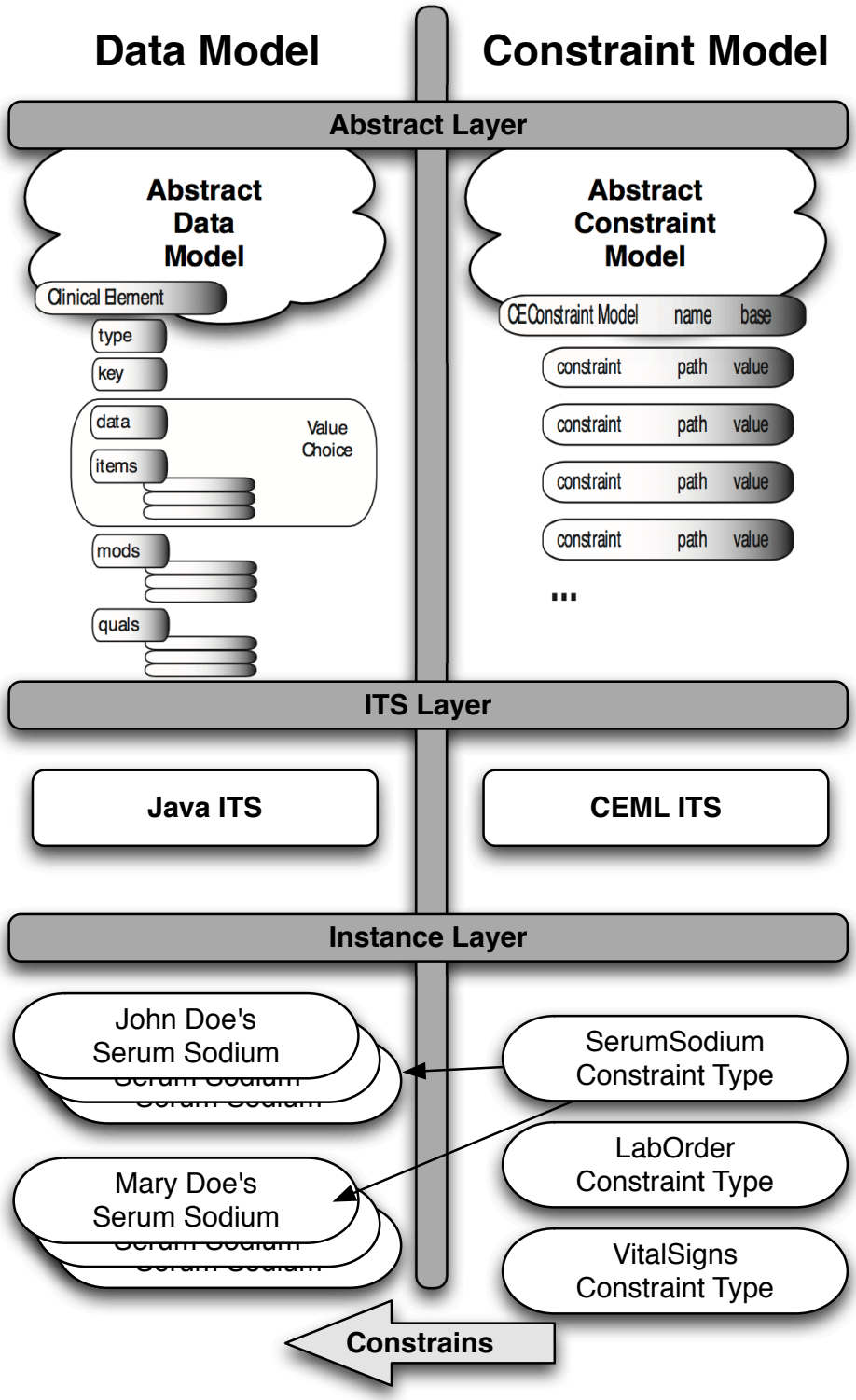


Figure 3.2. The use of the Clinical Element Model involves implementing both the Abstract Instance Model and the Abstract Constraint Model.

stored in a CE Instance, but this is avoided because each CE Instance is linked to a specific constraint specification called a Clinical Element Constraint Type, which is an instance of the Abstract Constraint Model (Figure 3.2).

3.2.1 Introduction

The heart of our approach is a recursive model with the core recursive element being the Clinical Element. Thus, the Instance Model is a tree of Clinical Element nodes. In this section, we will describe the parts of this recursive Clinical Element which make up the Abstract Instance Model.

If we examine the most basic skeleton of the Clinical Element (Figure 3.3), there is a *type*, a *key*, and a *value choice*. The *type* is a coded value which identifies the CE Constraint Type to which the instance will conform. The *key* is a coded value for the real-world concept that is important or key to what the instance is attempting to describe. The value choice is a choice between a *data* property or *items*, where *data* is a derivative of the HL7 version 3 datatype ANY, and *items* is a sequence of one or more Clinical Elements which gives the model its recursive nature. Later, we will learn about the remaining Clinical Element properties such as *mods*, *quals*, and *atts*.

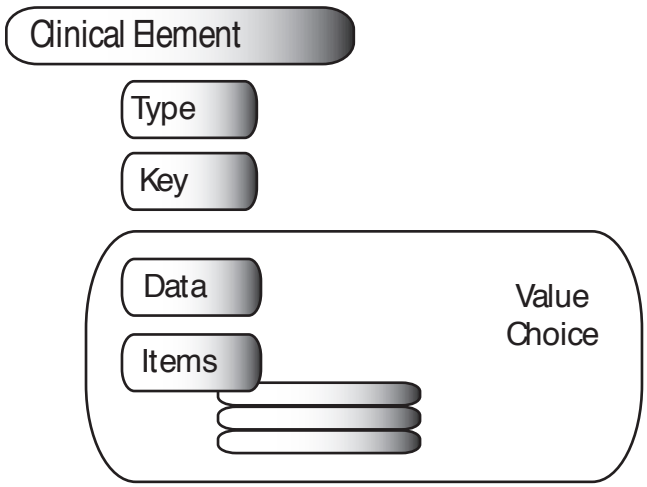


Figure 3.3. Clinical Element Instance Model.

3.2.2 Type

The Clinical Element *type* property is a coded value of type **CNE**, which specifies the CE Constraint Type, known as a **CEType**, to which this instance conforms. The allowable values for *type* consist of the domain of all defined CE Constraint Types. In the CEML syntax, CETypes are defined with the element name **cetype**. So in our implementation of the Abstract Instance Model and Abstract Constraint Model, the property *type* specifies a **cetype** authored in CEML. Examples of the Constraint Types to which *type* specifies would include types such as LabObservation[1], or Order.

3.2.3 Key

The *key* is a coded value represented by a **CWE** datatype. The *key* code is a code for a real-world concept that is important or key to what the model is attempting to describe. An example of a real-world concept is Serum Sodium, which has nothing to do with Clincal Elements or computers. The concept of Serum Sodium exists in the real world and has a known meaning in the field of medicine. It can be said that the *key* code links the Clinical Element Instance to a real-world concept. In Figure 3.4 is an example of some partial instance data that conforms to the constraints specified in the constraint type LabObservation.

[1]LabObservation is a type used to represent Quantitative Lab Observation data such as 140 mEq/L or 4.2 mmol/L.

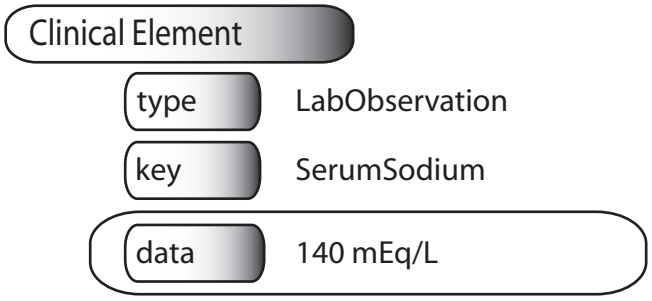


Figure 3.4. Clinical Element Instance with *key* and *data* values, and constrained by *type* LabObservation.

3.2.4 Value Choice

The heart of a Clinical Element Instance is its value, which is the payload. The value is a choice between either the property *Data* or *Items* where the former is a leaf node, and the latter is a list of children Clinical Elements. The next two sections will discuss these in more depth.

3.2.5 Data

The *data* property is represented by an HL7 version 3 datatype, and is used to represent values such as numbers, strings, and codes. At Intermountain Healthcare, we are actually using a subset of all the allowed HL7 datatypes and in fact, we have modified these slightly. For detailed information regarding the datatypes, please refer to the Appendix. If a Clinical Element Instance instantiates the *data* attribute instead of *items* attribute, this Clinical Element node becomes a leaf node in the Clinical Element Instance tree.

Clinical Element Figures

To reduce the size of figures for this section, we are going to typographically place values for the Clinical Element Instance property *type* in the place we have been putting the word Clinical Element. An example of this can be seen in Figure 3.5. Another typographic form that will be used to conserve page space will move the value for the *key* code next to the the *type* name. This can be seen in Figure 3.6. Do not be confused by these diagrams into thinking the constraint types indicate structure, because the constraint types only limit values in the single structure which is the Clinical Element Instance.

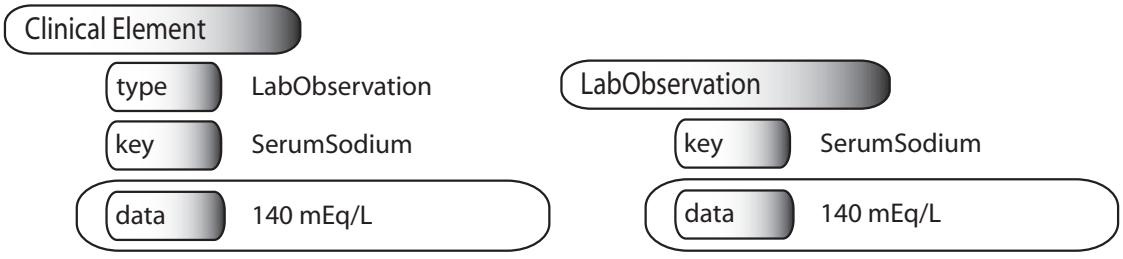


Figure 3.5. Type placement to reduce figure size.

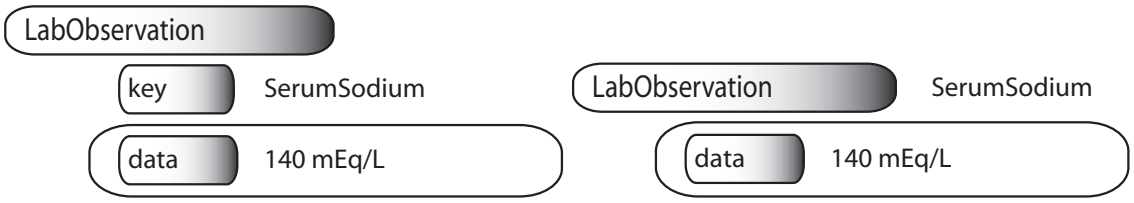


Figure 3.6. Key code placement to reduce figure size.

3.2.6 Items

The Abstract Instance Model defines the *items* property as a sequence of child Clinical Element nodes. This is the functionality that gives the Clinical Element Instance its recursive nature with the ability to represent complex nested data. An example of a Clinical Element Instance that uses *items* is seen in Figure 3.7. Here we have instance data that conforms to the constraint type BloodPressurePanel, which allows two child instances in *items*: one that conforms to the constraint type SystolicBloodPressure and a second that conforms to the constraint type DiastolicBloodPressure.

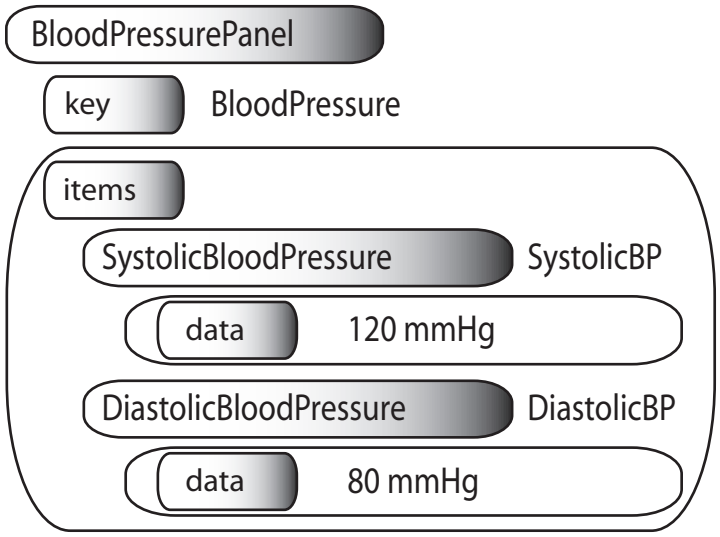


Figure 3.7. Instance Data with Items that conforms to the constraint type BloodPressurePanel.

3.2.7 Quals and Mods

The Abstract Instance Model also defines two other collections of Clinical Element nodes which serve to alter the meaning of the instance. These two lists of Clinical Elements are called *quals* and *mods*, which stands for Qualifiers and Modifiers. They are named to represent the extent to which they alter the meaning of the instance. In Figure 3.8 it is shown where the properties *quals* and *mods* fit into the Abstract Instance Model. A Clinical Element Modifier that is in *mods* alters the meaning of the instance to such an extent that one can never use the instance data without considering the effect of the Modifier. A Clinical Element Qualifier that is in *quals* is considered to add information to the value choice and does not actually change the meaning of the value choice in a way that makes it dangerous to ignore this change. However, it is the case that sometimes even qualifiers can not be ignored. For example, a qualifier that indicates a systolic blood pressure was taken during a stress test. If the context of the stress test is ignored, then the meaning of the systolic blood pressure would be misinterpreted.

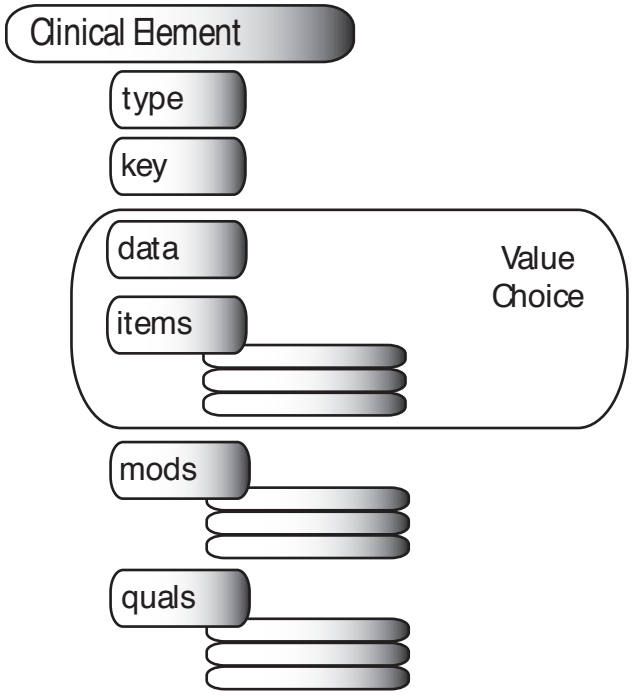


Figure 3.8. CE Abstract Instance Model with Mods and Quals.

In Figure 3.9 is an example of adding a qualifier to the instance constrained as a BloodPressurePanel in Figure 3.7. In Figure 3.9, we specify the BodyPosition of the patient as "Sitting" when the Blood Pressure was measured. This qualifier applies to the value choice, which is *items* in this case, which means the qualifier applies to SystolicBloodPressure and DiastolicBloodPressure.[2] In Figure 3.10 is an equivalent instance example of how the BodyPosition qualifier affects SystolicBloodPressure. Here this BodyPosition applies to the *data* attribute containing 120 mmHg which is the actual measurement. So this is a simple example demonstrating the scope of a qualifier, and how a qualifier within a panel applies to the children in the panel.[3]

[2]Specific rules stating how qualifiers and modifiers affect the children contained in *items* will be explained later.

[3]The children of a panel are either statements or other panels.

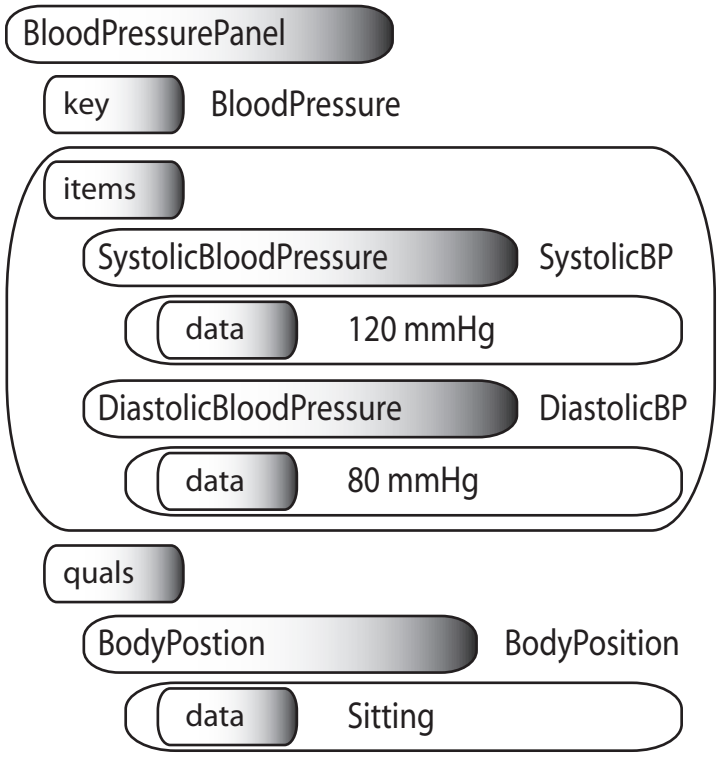


Figure 3.9. CE Instance representing a Blood Pressure Panel measured in Sitting Position.

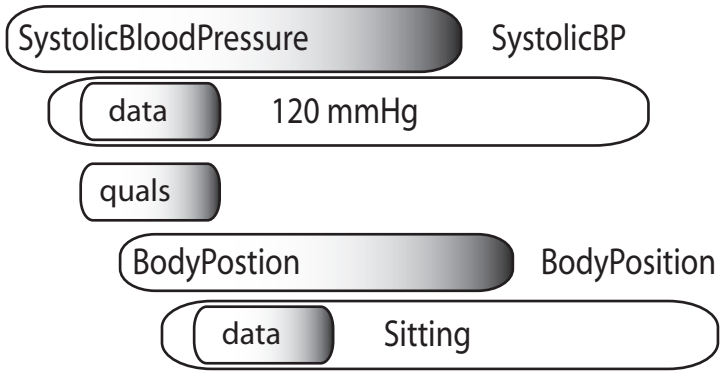


Figure 3.10. CE Instance representing a Systolic Blood Pressure measured in Sitting Position.

3.2.8 Attribution

The Abstract Instance Model also defines another collection of Clinical Element Instance nodes which is not shown in Figure 3.8. This collection exists at the same level as *quals* and *mods* and is called *atts* for attributions, the difference being that it can only contain subtypes of Attribution. An Attribution has a specific structure which defines an action, and the who, where, why, and when information regarding that action. An example of an attribution is Collected which could exist in a LabObservation, and Collected would contain information such as when the sample was collected, who collected it, and why it was collected.

3.2.9 Instance ID

The Abstract Instance Model defines the property *instanceId* which is not shown in Figure 3.8. This identifier is used to reference individual instances of stored patient data. The identifier is defined to be unique for each Clinical Element Instance node; thus, it must be unique across the enterprise, but we recommend that it be globally unique across all enterprises. If it is not globally unique, then this will cause difficulty importing patient data from one institution to another, because the imported identifiers may have already been used in the target system.

3.2.10 Alternative Data

The Abstract Instance Model defines the property *alt* to be a choice between a **CWE**, **PQ**, or an **ED** datatype. The purpose of *alt* is to allow the collection of unexpected or alternative data representations in an instance. For example, suppose Systolic Blood Pressure data were being collected into instances, and those instances were constrained by a constraint type called SystolicBloodPressure which required data to be a **PQ**. But then the system received a Systolic Blood Pressure measurement with a coded value of "HIGH." Since the Abstract Instance Model supports all data, this coded value of "HIGH" could be put into the *data* property as a **CWE**, but then the instance would fail validation by the constraint type SystolicBloodPressure. So instead, the coded value of "HIGH" can be placed into a **CWE** within *alt*, and "null" can be placed in the *data* property, with a corresponding null flavor. *nullFlavor* is a property of the HL7 version 3 datatypes. Figure 3.11 shows an example where the expected *data* section instead has a *nullFlavor* of Not Applicable (NA), and the coded value of HIGH is put in the *alt* section.

3.3 Abstract Instance Model Specification

The properties of CEInstance defined by the Abstract Instance Model are listed in Table 3.1. The CHOICE construct listed in the table indicates that one datatype is chosen from the set. The SEQUENCE construct indicates 0 to Many of the indicated type.

[4] Additionally, there is a choice between the properties *data* and *items*, and one or the other must exist.

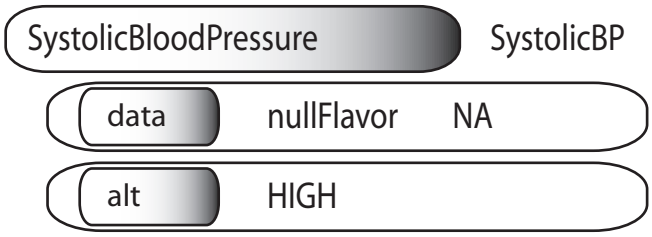


Figure 3.11. Instance Data demonstrating the *alt* property.

Table 3.1. The properties of CEInstance

Property	Cardinality	Type
type	1	CNE
key	1	CNE
data	0-1	CHOICE<CWE, CO, ST, PQ, IVLPQ, RTOPQ, TS, II, INT, REAL, ED>
items	0-1[4]	SEQUENCE<CEInstance>
quals	0-1	SEQUENCE<CEInstance>
mods	0-1	SEQUENCE<CEInstance>
atts	0-1	SEQUENCE<CEInstance
instanceId	1	ST
alt	0-1	CHOICE<CWE, PQ, ED, ST>

3.4 CE Abstract Constraint Model

The Clinical Element Abstract Constraint Model defines the allowable constraints that may be placed on the CE Abstract Instance Model. It is important to understand that any information, even nonsense patient data, can be instantiated in a CE instance, so it is the role of the Constraint Model to ensure that instances contain medically meaningful data. In the Abstract Constraint Model, the constraints are defined abstractly, but in a later section, CEML is defined, which is an XML ITS of the CE Abstract Constraint Model.

The properties of the constraint model are very simple. In Figure 3.12, it can be seen that it is actually just a named collection of individual constraints. Each individual constraint constrains a part of the instance model. Examples of constraints are the constraint of the data value to less than 500, the constraint of the data value to greater than zero, or a constraint of units to the coded concept of "mmHg."

First, there is a *name* property which is used to name the collection of constraints. This named collection of constraints is called a CE Constraint Type, or a **CEType**. Next is the *base* property which can be optionally used to import another named collection of constraints as a starting point; (base) is a reference to another **CEType**. And finally, there is a list of one to many *constraint* structures, where each states the part of the instance model

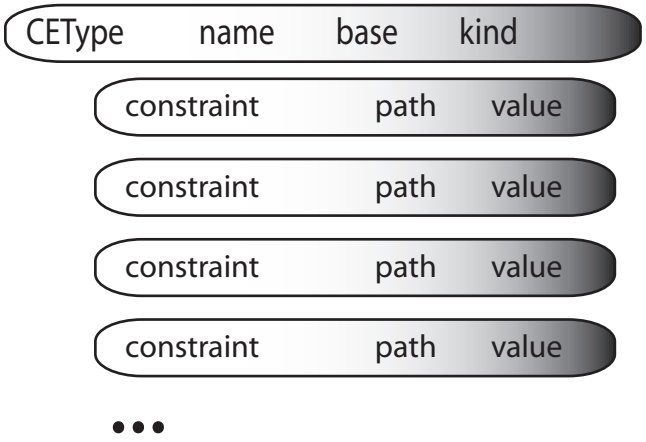


Figure 3.12. The CE Abstract Constraint Model describes the constraints on the CE Abstract Instance Model.

to constrain, and how to constrain it. The properties of the constraint model or **CEType** are listed in Table 3.2.

3.4.1 Name

The *name* property represents the unique textual identifier for this collection of constraints; it is the name of the **CEType**. Some textual examples could include **VitalSignsPanel**, **SystolicBloodPressure**, and **AbnormalFlag**. A named collection of constraints is called a CE Constraint Type or **CEType**. So VitalSignsPanel is the name of a **CEType** which defines the constraints for instance data which would represent a vital signs panel.[5]

[5]It should be noted that the name of a **CEType** is meaningless, and VitalSignsPanel could have been called VSignsPanel, VitalsPanel, or anything the modeler wished.

Table 3.2. CEType Properties

Property	Cardinality
name	one
base	zero to one
kind	one
constraint	one to many

3.4.2 Base

The *base* property identifies an external named collection of constraints, or more specifically, an external **CEType** that is used as the starting point for this new collection of constraints. For example, when defining the constraints for SerumSodium, the base may be set to LabObservation which imports all the general constraints of a LabObservation, and then the specifics of a SerumSodium can be specified in the constraint section.

3.4.3 Kind

The *kind* property declares a functional category for the defined **CEType**. The options, shown in Table 3.3, include "noninstantiable," "component," "statement," "panel," "modifer," and "attribution."

3.4.4 Constraint

The *constraint* structure represents individual constraints on the Abstract Instance Model. The constraints are where the Abstract Constraint Model is tied to the Abstract Instance Model. Each constraint consists of a *path* and a *value*. The *path* property identifies a

Table 3.3. Allowable values for kind

Value	Description
noninstantiable	No instantiations will be constrained directly with this **CEType**. An incomplete definition that must be further defined to become a statement, panel, or component.
component	**CEType** will be used to constrain part of a storable instance, such as an internal qualifier.
attribution	**CEType** will be used to constrain an attribution.
modifier	**CEType** will be used to constrain a modifier
statement	**CEType** will be used to constrain a complete storable instance using *data*
panel	**CEType** will be used to constrain a complete storable instance using *items*, and the *items* must be either statements or panels.

specific location in the CE Abstract Instance Model tree, and the *value* property is the value this location is constrained to.

3.5 Understanding CEM Paths

An important factor to understanding the Abstract Constraint Model is understanding the allowable paths that can be constrained. The follow sections detail the allowable paths and explain the corresponding values. A complete understanding of the CE Abstract Instance Model, as well as a complete understanding of the datatypes, is essential to understanding these paths, because the paths are paths into the Instance Model and into the datatypes. When a path gets to the level of a datatype, see the Appendix to learn about all the allowable properties for that datatype.

The following section will explore the more commonly used paths, but is not a complete list, as this section will not list every property of every allowable datatype. The properties within a datatype are divided into value and rule types. A value type is simply constraining a direct instance property within the datatype, and to validate one must simply make sure the value in the instance matches the constraint. A rule type is constraining one of the rule properties that were added to the datatypes, and logic must be used to understand and enforce this constraint.

3.5.1 Key

The Abstract Instance Model declares the property *key* as a **CWE**, so all the properties within **CWE** are available for constraint. Table 3.4 lists the most common properties of **CWE** that are used within *key* for constraint. The path *key.code* constrains the *key* property to have a *code* with the specified value. The path *key.domain* constrains the *key* property to have a *code* within the specified domain. See the Appendix for additional datatype properties.

3.5.2 Data

The *data* property is constrained to hold one of the allowed datatypes as indicated in the Abstract Instance Model. To indicate the datatype, the path **data.type** is used, and then

Table 3.4. Examples of constraint paths for key

Path	Type
key.code	value
key.domain	rule

a lowercase version of the datatype is used as a value, such as cwe, co, st, pq, ivlpq, rtopq, ts, ii, int, real, or ed.[6]

Depending on what datatype is selected, the path will continue on with that datatype, followed by the allowable properties of that datatype. The following will demonstrate a few possible paths using **CWE** and **PQ** as an example. Also note that this is not a complete list of all the properties of either **CWE** or **PQ**. See the Appendix to see a complete list of properties for every datatype.

If *data.type* was chosen to have a value of "cwe," then the path *data.cwe* would be available for further constraint. The path could be continued at this point with any of the properties in **CWE**. Table 3.5 lists the most common paths using **CWE**. The path *data.cwe.code* is used to constrain the *cwe* datatype instance to have a *code* with the specified value. The path *data.cwe.domain* is used to constrain the *cwe* datatype instance to have a *code* within the specified domain.

Another example is if *data.type* was chosen to have a value of "pq," then the path *data.pq* would be available for further constraint. The path could be continued at this point

[6]It is also possible to use datatype choices here, which is a comma delimited list of the individual datatypes. Please see Section 3.15 for more information.

Table 3.5. Examples of constraint paths for data.cwe

Path	Type
data.cwe.code	value
data.cwe.domain	rule

with any of the the properties in **PQ**. Table 3.6 lists the most common paths using **PQ**. The path *data.pq.unit.code* constrains the *unit* property within the *pq* datatype instance to have a *code* with the specified value. The path *data.pq.unit.domain* constrains the *unit* property within the *pq* datatype instance to have a *code* within the specified domain. The path *data.pq.normal* declares the unit that should be used for the normalized value in the persistent datastore.

3.5.3 Sequences

The CE Abstract Instance Model declares *items*, *quals*, *atts*, and *mods* to be a list of 0 to many Clinical Element Instance nodes. These are the points of recursion in the CE Abstract Instance Model. In the Abstract Constraint Model, a path is defined to indicate constraints on this sequence, and the recursion allows us to continue the path to the *key*, *data*, etcetera that exist in the recursed CE Instance node. The syntax to constrain a CE Instance node within one of these lists is defined by beginning the constraint *path* with either "item," "qual," "att," or "mod," followed by a unique identifier, and then assigning the *value* to an existing defined **CEType**, as shown in Figure 3.13. If the Abstract Instance Model was followed strictly, the true path should be "items," "quals," "atts," and "mods," but in use, we have found the singular form to be more readable and understandable for most. The unique identifier has no meaning so it can be any textual string, but the person creating the unique identifier usually follows the convention to use a lowercase version of the referenced **CEType**.

Identifiers have two properties named *type* and *card*. The *type* property assigns a referenced **CEType** to the identifier. The *card* property constrains the cardinality of the

Table 3.6. Examples of constraint paths for data.pq

Path	Type
data.pq.unit.code	value
data.pq.unit.domain	rule
data.pq.normal	rule

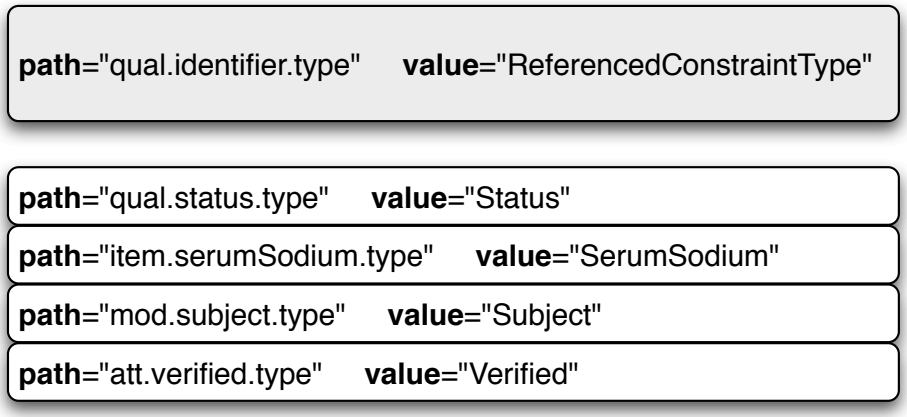


Figure 3.13. A unique textual identifier is used to constrain a particular CE Instance node within *items*, *quals*, *atts*, and *mods*.

referenced **CEType**, and is a rule-based constraint rather than a value-based constraint. The path to assign the cardinality is indicated in Table 3.7 and the allowable values for *card* are indicated in Table 3.8.

The paths **item.*identifier***, **qual.*identifier***, **att.*identifier***, and **mod.*identifier*** can also be continued down into the CE Instance recursively, and thus, anything described prior in this section can be appended on to the identifier path.

Table 3.7. Paths to constrain the *card* property

Path	Type
item.*identifier*.card	rule
qual.*identifier*.card	rule
mod.*identifier*.card	rule
att.*identifier*.card	rule

Table 3.8. Allowable values for the property *card*

Value	Description
0	Zero
1	Exactly One
0-1	Zero to One
0-M	Zero to Many
1-M	One to Many

3.6 Kind

Clinical Element Constraint Types (**CETypes**) may be classified into several categories. This classification is identified by using the property *kind* in a Clinical Element Constraint Definition. Currently, there are 6 possible values for *kind*, shown in Table 3.9.

3.6.1 Statement

A statement is a complete assertion about a particular aspect, characteristic, or condition of a patient. A statement can be thought of as a complete sentence, such as "The patient John Doe had a Hematocrit of 38 percent on June 1, 2007." When a Clinical Element Constraint Model is designed to model a statement, the *kind* property is set with a value of

Table 3.9. The values of the property *kind*

Value	Description
statement	The definition defines a complete assertion
panel	The definition defines a collection of complete assertions
component	The definition defines a portion of an assertion, qualifier, or attribution
attribution	The definition defines an attribution
modifier	The definition defines a modifier
noninstantiable	An incomplete definition that must be further defined to become a statement, panel, or component.

"statement." Since a statement is a complete assertion, it can stand on its own in a patient's medical record. Because of this, any data instances which conform to a **CEType** designated as a statement are storable on their own in the electronic patient record.

The individual parts of the statement, including modifiers and qualifiers, which also conform to their own **CETypes**, cannot exist on their own. This is because these individual parts that make up the statement are not meaningful by themselves out of context of the statement.

Two types of statements exist, which are simple statements and compound statements. The basic difference between the two is that a simple statement contains a single datatype value, and a compound statement is made up of a collection of dependent individual values. This is shown in Figure 3.14

A simple statement is a statement whose meaning is conveyed by a single clinical value, with associated modifiers and qualifiers. In The Clinical Element Model, a simple statement has the value choice of data rather than items. This means that the value is an HL7-like data type, such as a **PQ** or a **CWE**. An example of a simple statement is a hematocrit lab result.

A compound statement is a statement whose meaning is conveyed by multiple clinical values, with associated modifiers and qualifiers. The meaning of the compound statement is dependent on a set of elements with values being interpreted together within the context of

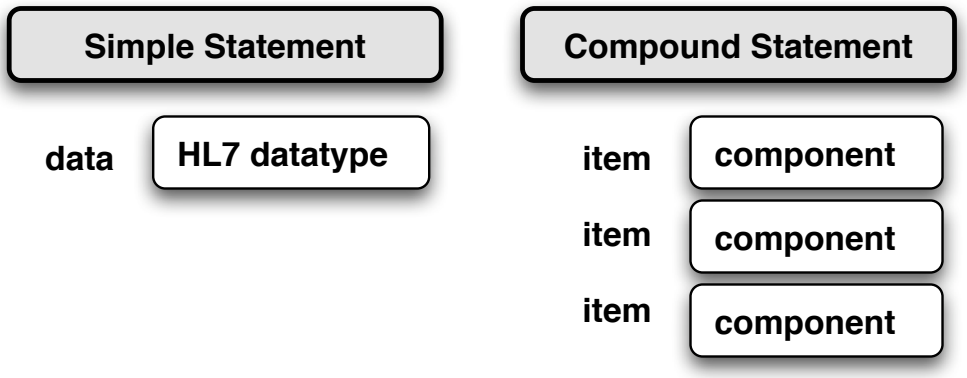


Figure 3.14. A Simple Statement versus a Compound Statement.

the collection. In the Clinical Element Model, a compound statement has the value choice of items rather than data. Each item within a compound statement must have its own *kind* property with a value of "component," as they cannot be statements or panels, because they can never exist on their own. An example of a compound statement is a pharmacy order. A pharmacy order is so complex that there is no simple datatype that is the true value of the order. Instead, the value is multiple items including such information as dose, frequency, and drug. Moreover, one cannot view this collection as a panel, because these pieces of information cannot be understood individually and instead must always be viewed together to create the order.

3.6.2 Panel

A panel represents a common grouping of clinical observations. It is a collection of statements or other panels that could each exist independently outside the panel. A Chemistry 7 lab result is an example of a common lab panel. A Chemistry 7 contains statements representing Serum Sodium, Serum Choloride, and other measurements, which could each exist independently of the enclosing panel. If the goal is to build a **CEType** to represent a panel, the *kind* property is set with a value of "panel." When a **CEType** is designated as "panel," then all its *items* must have their *kind* with a value of "statement" or "panel." The *kind* value of the *items* is what differentiates a panel from a compound statement, as shown in Figure 3.15.

3.6.3 Component

A component is a CE Constraint Type that is only used within another Constraint Type and on its own does not constitute a complete statement. A data instance which conforms to a Clinical Element Constraint Type designated as a component cannot be persisted alone in the electronic patient record. Instead, the data instance must only be persisted as an internal part of another data instance which conforms to a statement or panel. Examples of components include dates, times, and measuring devices. Components can be used as qualifiers or as items in a compound statement or compound component. Thus, it is easy to

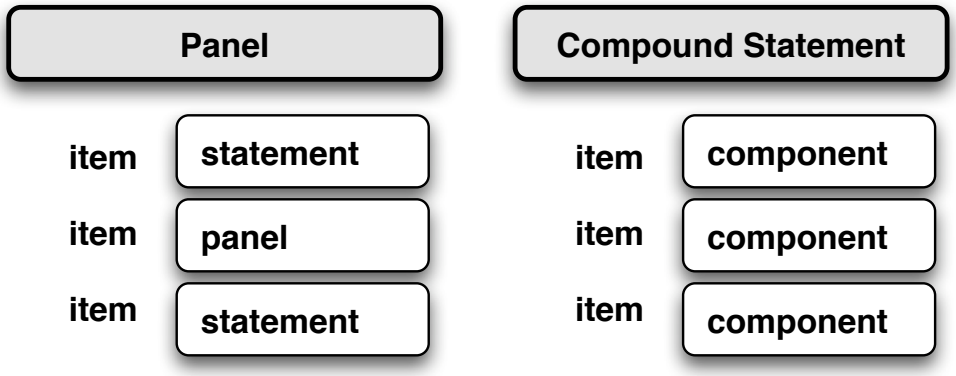


Figure 3.15. A Panel versus a Compound Statement.

see that it would be nonsensical to store a measuring device such as a blood pressure cuff in a patient record outside the context of a blood pressure statement.

A simple component is a component whose meaning is conveyed by a single clinical value, with associated modifiers and qualifiers. In The Clinical Element Model, a simple component has the value choice of data rather than items. This means that the value is an HL7-like data type, such as a **PQ** or a **CWE**. An example of a simple component is a *Status*.

A compound component is a component whose meaning is conveyed by multiple clinical values, with associated modifiers and qualifiers. The meaning of the compound component is dependent on a set of elements with values being interpreted together within the context of the collection. In the Clinical Element Model, a compound component has the value choice of items rather than data. Each item within a compound component must have its own *kind* property with a value of "component," as they cannot be statements or panels, because they can never exist on their own. An example of a compound component is *SpecimenDescription*.

3.6.4 Noninstantiable

A noninstatiable Clinical Element Constraint Type cannot be instantiated with patient data even within a statement or panel. It is only used as a modeling tool as a parent type

from which subtypes can be created. For example, we model *Order* as noninstantiable, and thus, to use this type, modelers are forced to create specific subtypes such as *LabOrder*, *OrderMedPCA*, or *OrderNursing*.

3.6.5 Modifier

A modifier is similar to a component, except that they can only be used in *mods* within the Clinical Element Instance Model. In the past, we did use component for **CETypes** destined to be used in *mods*, but this gives our compiler an extra check to make sure modelers are not making mistakes. To designate a **CEType** as a modifier, set the *kind* property to "modifier."

3.6.6 Attribution

An attribution is similar to a component, except that they can only be used in *atts* within the Clinical Element Instance Model. In the past, we did use component for **CETypes** destined to be used in *atts*, but this gives our compiler an extra check to make sure modelers are not making mistakes. To designate a **CEType** as an attribution, set the *kind* property to "attribution."

3.7 Modifiers and Qualifiers

Modifiers and qualifiers are themselves Clinical Elements nested within a Clinical Element Instance. They both have similar functionality, in that both add additional information to the value. The difference is that with modifiers, the additional information changes the meaning of the data, and hence we say they "modifiy" the data. Qualifiers, on the other hand, simply add information, so we say they "qualify" the data.

Let us look at the difference between a modifier and qualifier with an example. Say we have a Clinical Element Constraint Type which represents the laboratory test for Protein C, which is a protein involved in coagulation. An example of a qualifier for this Protein C model could be the technician who drew the blood from the patient. The qualifier adds extra information but does not change the meaning of the Protein C model in relation to the Patient. An example of a modifier would be the subject whose blood was tested. It may

seem that the subject is always the patient, but this is not the case. A patient may have relatives that are tested for Protein C and these results are stored in this patients record. In this case, the subject would identify the relative.

When an instance of patient data is examined, existing Modifiers must always be considered, because they significantly change the meaning of the Clinical Element Instance. In the example above, if the Subject modifier was ignored, it could appear that a Protein C value was from the patient, when in reality it was from a father or mother. The qualifier in this example can be ignored, because the person who collected the blood sample does not change the interpretation of the value.

It should be noted that in some cases, a qualifier can change the interpretation of the value, and cannot be ignored. For example, if a Systolic Blood Pressure was recorded and a qualifier was used to indicate that this measurement was taken in the context of a stress test, then to ignore this qualifier, one would assume the patient had a high Systolic Blood Pressure. Because of cases like this, it may be wise in the future to divide qualifiers into two separate categories: ones that can be completely ignored, and others that should probably be considered.

Currently, we have only defined three modifiers: *CertaintyOfExistance*, *CertaintyOfOccurance*, and *Subject*. These are described below.

3.7.1 Negation or Certainty Modifier

Originally, the model contained a modifier for negation that was called *Negation*. In further review of this issue, it was determined there are really two shades of negation. These two shades of negation are called *CertaintyOfExistance* and *CertaintyOfOccurance*. Both of these are children of a supertype called *Certainty*.

CertaintyOfExistance is defined as the degree of certainty that what is in data is true. The range of values are "No, Probably Not, Maybe, Might Be, Probably, Affirmative." An example of its use would be in a Diagnosis model to indicate statements such as "The patient probably has Multiple Myeloma."

CertaintyOfOccurance is defined as the degree of certainty that a procedure or action was performed. The range of values are "Absolutely Not, Unlikely, May Have, Might

Be, Absolutely Has." An example of its use would be in a Procedure model to indicate statements such as "The patient absolutely has had an Appendectomy."

The absence of certainty defaults to "Affirmative" and "Absolutely Has," respectively. The negation modifiers are only allowed in leaf node Clinical Element Instances. Leaf Node Clinical Elements are those that have an HL7 datatype in contrast to *items* as the value choice.

3.7.2 Subject Modifier

Another important modifier is called *Subject*. This modifier indicates who the data in the Clinical Element is about. In the absence of a *Subject* Modifier, *Subject* defaults to self. This modifier allows us to add Clinical Elements in a patient's record that contain data about family members, household members, or donors.

3.8 Attribution

Attribution is used to define an action along with the details of that action, including the who, what, where, when, and why the action was performed. The qualifiers for Attribution can be seen in Table 3.10. Attribution is more commonly known in the informatics community as Provenance [25].

Table 3.10. The qualifiers of Attribution

Value	Description
startTime	When the action started
endTime	When the action ended
participant	A participant in the action
patientLocation	The patient's location during the action
providerLocation	The provider's location during the action
reason	The reason for the action

3.8.1 Modeling

Attribution is modeled as a noninstantiable **CEType** and thus is never used directly, but instead, *Attribution* is subtyped for further use. Some examples of subtypes of *Attribution* are *Observed*, *Verified*, *Created*, *Reported*. From these subtype names, it can be seen that our convention is to name subtypes of *Attribution* with a verb in the past tense indicating the action that is intended.

The *Attribution* model can be seen in Figure 3.16.[7] It is modeled with a coded value in the data section, which is constrained in subtypes to always have a value of the action performed. As previously mentioned, the *kind* property is declared as "noninstantiable" which means that no instances can be created using *Attribution* itself. When subtypes are created, the kind property is redeclared within the subtypes as "attribution," and these subtypes can then be used within other instantiable **CETypes**.

An example *Observed* subtype can be seen in Figure 3.17. Other examples of subtypes include *Verified*, *Created*, and *Reported*. When *Attribution* is subtyped, the *data.cwe* property is constrained to a specific action or a domain of actions. Figure 3.18 shows an example of an instance conforming to the **Observed** subtype which was previously declared in Figure 3.17.

3.9 Semantic Links

Once clinical element instances are stored in the patient information system, we need the ability to link one instance with another instance and then assign a relationship type to this link. This type of link is called a semantic link because the relationship is a concept with a computable definition.

An example of a need for a semantic link would be linking a Throat Culture instance that was positive for Streptococcus pyogenes to the resulting Order for Penicillin. Clinical element instances are linked via their instance identifiers, by a relationship code such as "result in" or "caused by."

---

[7] _ECID appended to a text string indicates this string maps to a code in the ECIS Terminology Server.

```
<cetype name="Attribution" kind="noninstantiable">
   <key domain="Action_KEY_ECID"/>
   <data type="cwe"/>
   <qual name="startTime" type="StartTime" card="0-1"/>
   <qual name="endTime" type="EndTime" card="0-1"/>
   <qual name="participant" type="Participant" card="0-M"/>
   <qual name="patientLocation" type="PatientLocation" card="0-1"/>
   <qual name="providerLocation" type="ProviderLocation" card="0-1"/>
   <qual name="reason" type="Reason" card="0-M"/>
   <qual name="comment" type="Comment" card="0-M"/>
   <constraint path="data.cwe.code" value="Action_DOMAIN_ECID"/>
</cetype>
```

Figure 3.16. Attribution model defined in CEML.

```
<cetype name="Observed" base="Attribution" kind="attribution">
        <constraint path="data.cwe.code" value="Observed_ECID"/>
</cetype>
```

Figure 3.17. Observed subtyped from Attribution in CEML.

3.9.1 The Difference between a Semantic Link and a Qualifier

A qualifier differs from a semantic link, in that a qualifier is designed as an internal part of the source or target model, and the instance data for that qualifier is stored as an internal part of the instance. On the other hand, a semantic link is physically separate from the source and target model, and no change to the source and target instance is needed to create a link between them.

In almost every case where a semantic link could be used, it is possible that qualifiers could have been modeled instead, and the same goes for the reverse case. For example, in Figure 3.19, *ThroatCulture* could have been modeled with a qualifier called *ResultingMedOrder*, and the *Order* could have been modeled with a qualifier to indicate the reason it was ordered.

Because of this, the modeler needs to keep the aspects of each approach in mind:

- A qualifier cannot be a statement, but a semantic link can link to a statement.

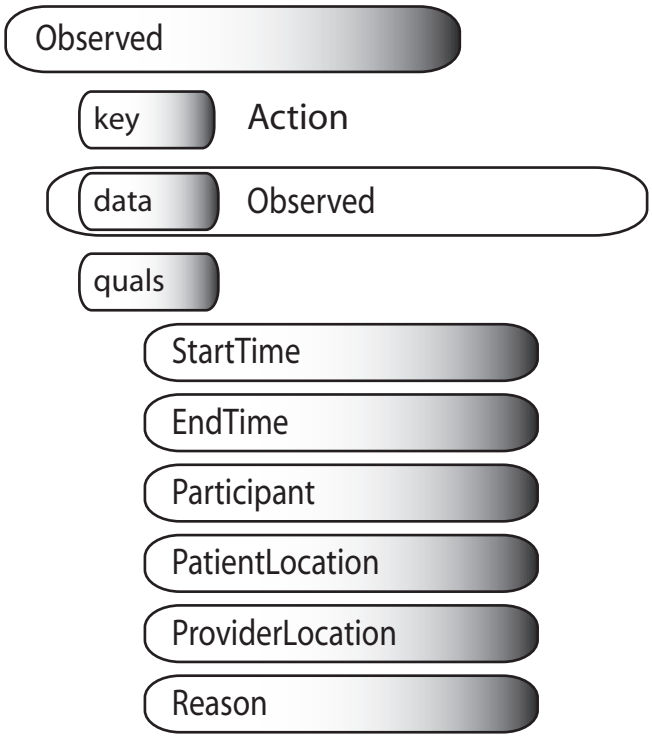


Figure 3.18. A CE Instance conforming to the Attribution subtype called Observed.

- Semantic links are by their nature bidirectional.
- Qualifiers do not really work well for joining two existing instances.
- Semantic links do not require a change to stored data instances.
- A semantic link should never change the meaning of the source or target instance; thus, it should always be safe to ignore semantic links.

3.9.2 Tightly versus Loosely Coupled

We have decided not to make the distinction between tightly coupled and loosely coupled semantic links for the time being, but it is described here for discussion.

During the modeling process, there is sometimes a debate as to whether one should choose a qualifier or a semantic link to model a given concept. If this debate is settled and the modelers chose a semantic link, then this semantic link is slightly different than other semantic links that did not require such a debate. This semantic link is almost a qualifier, and almost made its way directly into the model. In order to signify that this semantic link

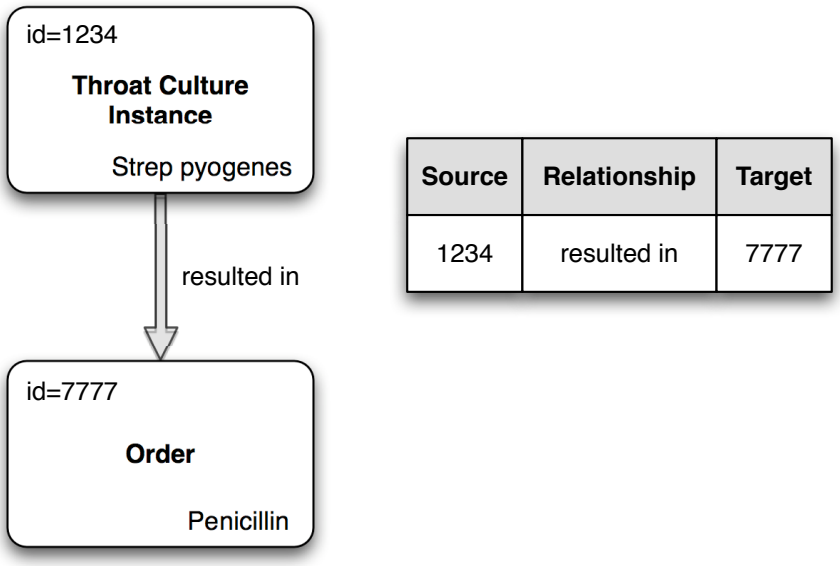


Figure 3.19. A Throat Culture Instance positive for Strep pyogenes is semantically linked to an Order for Penicillin.

is important when considering the model, a semantic link can be coupled as either "loose" or "tight." This could be a binary value, but it would also be possible to use a scale, such as a range from 0 to 10. An advantage of this is that queries could automatically bring back important semantically linked instances.

To limit the complexity of semantic links, we have decided not to make this distinction for the time being. Instead, we leave the responsibility in the hands of the query. Important semantic links can be included in queries. A benefit of this is that all semantic links are treated identically, and it is always known that you must query a semantic link to retrieve it.

3.9.3 Target as Concept

Although we did not implement this, we have discussed the possibility that the target of a semantic link could actually be a coded concept as opposed to only another instance. This would allow users to label stored instances with a relationship and a target concept. An example of this would be as follows. A physician is conducting a research project and he is collecting patient samples for laboratory evaluation. After the lab result is stored,

he could link the stored lab result with the relationship “was collected for" and the coded concept target “Dr. Smith’s Research Project.”

3.9.4 Modeling

Semantic links are created between stored instances of data, so the question arises as to whether we allow users to link any two instances together and with any relationship. If this was allowed, the danger is that users could create nonsensical semantic links between instances. Instead, we have decided to put semantic link constraints into our constraint language, which limit which models can be linked to which, and with what relationship. The properties we need to identify in a semantic link are listed in Table 3.11 and the properties we may need to identify in the future are listed in Table 3.12.

In Figure 3.20 is an example of the CEML syntax for a semantic link constraint. Each link is given a name, similar to how qualifiers, modifiers, and items are given a name. The allowed relationship is defined in the *relation* property with a code from the terminology server. At the moment, we do not have support for a domain of allowed relationships, and each relationship must be specifically defined. The number of allowed links is defined with the *card* property. Individual constraint values within the target instance are specified with the *target* element, which contains a *path* and *value* property analogous to the *constraint* element.

Table 3.11. The properties we need to identify in a semantic link constraint

Value	Description
Target instance properties	type, key, data value, etc
Relationship	This is the forward relationship between the source and target
cardinality	The number of semantic links allowed

Table 3.12. Other properties we may need to identify in a semantic link constraint.

Value	Description
Target concept	If the target is a concept
Strength	This is the coupling strength which is either "loose" or "tight"

```
<link name="myLinkName" relation="resultedFrom_ECID" card="0-1">
    <target path="type.code" value="PAIN_MODEL_ECID"/>
    <target path="key.code" value="Assertion_KEY_ECID"/>
</link>
```

Figure 3.20. The CEML Syntax to constrain allowable semantic links.

3.9.5 Inverse Relationships

Although a semantic link is only the forward relationship, it would be useful for the implementation to have both a forward and reverse relationship. We have decided to have this work automatically. It is the requirement of the terminology server that every relationship code will have an inverse relationship. In this way, the modeler only worries about the forward relationship.

3.10 Scope

This section deals with the issue of whether instantiated qualifiers, attributions, and modifiers in panels apply to the child statements and panels within that panel instance. An example of this issue can be seen in Figure 3.21, which shows an instance conforming to a BloodPressure Panel. This panel contains the two statements SystolicBP and DiastolicBP, as well as a qualifier for BodyLocation at the panel level. The issue at hand occurs when one examines SystolicBP or DiastolicBP outside the context of this Panel, and the question is whether BodyLocation is applicable to the individual statements.

There are three scenarios we consider:

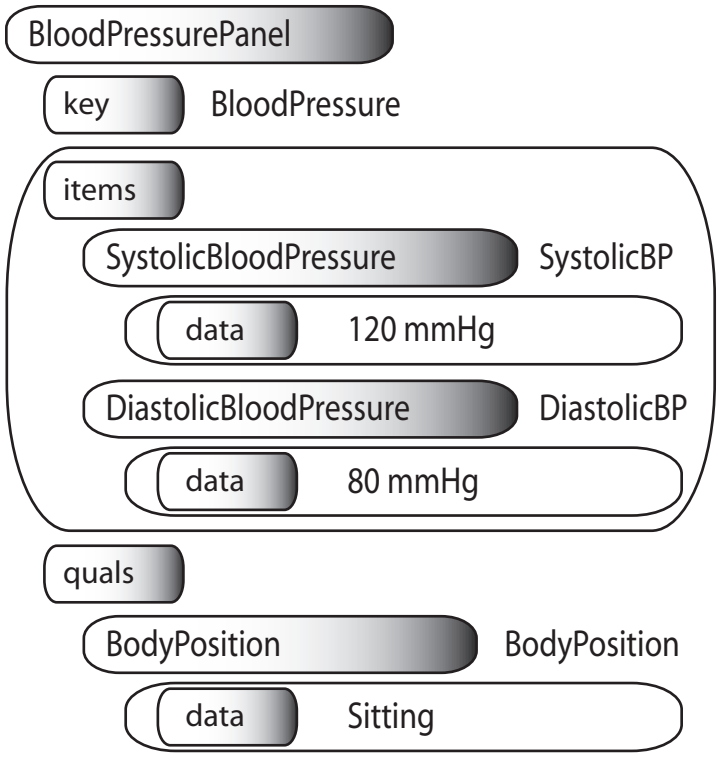


Figure 3.21. A CE Instance of BloodPressure with a qualifier at the panel level.

1. In the "override" case, the instantiated qualifier in the panel will apply to the child statement. But, if the child statement already has an instantiated qualifier of this type, then this local qualifier overrides the qualifier from the panel.
2. In the "local" case, the instantiated qualifier in the panel will not apply to the child statement.
3. In the "additive" case, the instantiated qualifier in the panel will apply to the child statement. If the child statement already has an instantiated qualifier of this type, then the qualifier from panel will apply alongside the existing qualifier.

3.10.1 Modifiers

The default scope for modifiers is "override." At the moment, the only two modifiers we have defined are *Subject* and *Negation*. *Negation* would never be used in a panel due to modeling style rules, so it is not a use case. *Subject* is used in panels, and we would

always want it to follow the scope rules of "override." So in the end, there is no use here for anything but "override."

3.10.2 Attributions

The default scope for attributions is "override."

3.10.3 Qualifiers

The default scope for qualifiers is "override."

3.10.4 Modeling

There are two basic approaches to model scope, one which is deterministic by **CEType**, and the other which is nondeterministic, and we may use either or both. In the deterministic approach, the *scope* property is declared at the **CEType** level, so that every time the **CEType** is used as a qualifier, modifier, or attribution, the **CEType** retains that declared *scope*. In the nondeterministic approach, the *scope* property is not declared at the **CEType** level, and thus, it is required to declare the *scope* when the **CEType** is used as a qualifier, modifier, or attribution. In either case, the value "override" would be the default if the *scope* was not declared. Examples of the deterministic and nondeterministic approach are shown respectively in Figures 3.22 and 3.23.

3.11 Root

The Clinical Element of type *Root* was the default base class for all clinical element constraint types. When a **CEType** is declared, it can be declared de novo, or the *base* property can be used to declare a parent **CEType**. If no *base* is declared, the **CEType** was given a *base* of *Root*.

```
<cetype name="MyType" kind="component" scope="additive"/>
```

Figure 3.22. Deterministic assignment of Scope.

```
<qual name="myQualifier" type="MyQualifier" card="0-1" scope="additive"/>
  or
<constraint path="qual.myQualifier.scope" value="additive"/>
```

Figure 3.23. Nondeterministic assignment of Scope.

3.11.1 History

Initially, *Root* had 6 qualifiers, but over time, we realized we did not want most of these qualifiers in every **CEType**. In the end, only one qualifier remained in *Root*, and this was *Comment*. The original 6 qualifiers we had in *Root* were *Subject*, *Classification*, *Actuality*, *Negation*, *Aggregate*, and *Comment* and the reasons for their removal are shown in Table 3.13.

3.11.2 Implementation

Engineering needs have required us to have the qualifier *Comment* moved out of the **Qualifiers** section, and be made a sibling of **Qualifiers** and **Modifiers**. Because of this, we no longer have any of the original qualifiers in *Root*, so de novo Clinical Element Constraint types inherit no constraints at the moment. Because of this, *Root* is currently deprecated.

Table 3.13. Reason for removal from **Root**

Value	Description
Subject	Only needed in Statements and Panels
Classification	Renamed to Label, which was subsequently removed because this is now handled with Semantic Links
Actuality	Not being used at the moment, so temporarily removed
Negation	Decision to only add this as needed, because its use does not make sense for all types, especially panels
Aggregate	Need to discuss, as this may still belong here, or this could become part of the base model.
Comment	Remains

However, more recently, we have found a need to add implementation-specific qualifiers to groups of Clinical Element models, and we are returning to the idea of *Root*, but making it more powerful, so that it can target specific subgroups of Clinical Elements such as just statements or just panels.

3.12 CEML

Our implementation of the Abstract Constraint Model is called Clinical Element Modeling Language or CEML. This ITS is defined using an XML syntax with specific elements and attributes which conform one to one to the constraint model. As of this writing, we have two forms of CEML, one that strictly follows the Abstract Constraint Model which we call "Strict CEML," and another form for modelers that has some shortcuts or macros to ease the authoring purposes which we call "Authoring CEML."

3.13 Strict CEML

Previously, we have described the Abstract Constraint Model as a collection of paths and values. Strict CEML is simply an XML representation of that collection of paths and values. An example of a Clinical Element Constraint Type defined in Strict CEML can be seen in Figure 3.24.[8]

[8] _CODE appended to a text string indicates this string maps to a code in some unknown Terminology Server.

```
<ceml>
  <cetype name="MyType" base="MyBaseType" kind="statement">
    <constraint path="key.code" value="MyModelRealWorldConcept_CODE"/>
    <constraint path="data.type" value="pq"/>
    <constraint path="qual.myQualifier" value="MyQualifier"/>
    <constraint path="qual.myQualifier.card" value="0-1"/>
  </cetype>
</ceml>
```

Figure 3.24. Strict CEML example of a Constraint Type called MyType.

3.14 CEML Authoring Syntax

This section will describe the Authoring CEML Syntax. It should be understood that everything that is allowed in Strict CEML is also allowed in Authoring CEML, but Authoring CEML adds additional shortcuts or macros, which can replace some of the constraint statements of Strict CEML. An example of a Clinical Element Constraint Type defined in Authoring CEML can be seen in Figure 3.25.

3.14.1 Root Element <ceml>

The **ceml** element is the root element for a CEML authored definition. It has no attributes, and the only allowable elements are one **header**[9] element and then the one **cetype** element, as seen in Table 3.14. The filename that contains this CEML definition should match the *name* attribute of the contained **cetype** element. Figure 3.26 shows a basic example.

3.14.2 The Element <header>

The **header** element is used to store authoring information regarding the authoring lifecycle of the **CEType**. Initially, this was a temporary solution until we had an authoring database server implemented. But currently, we have no plans to remove **header**.

[9]The **header** element is not displayed within figures for simplicity.

```
<ceml>
  <cetype name="MyType" base="MyBaseType" kind="statement">
    <key code="MyModelRealWorldConcept_CODE"/>
    <data type="pq"/>
    <qual name="myQualifier" type="MyQualifier" card="0-1"/>
  </cetype>
</ceml>
```

Figure 3.25. A Simple Authoring CEML example defining the constraints of a Clinical Element Type named *MyType*.

Table 3.14. Properties of <ceml>

Property	Type	Cardinality
header	element	one
cetype	element	one

```
<ceml>
  <cetype name="MyType">
    ...
  </cetype>
</ceml>
```

Figure 3.26. The backbone of a CEML definition. This definition would be stored in a file named MyType.xml.

3.14.3 The Element <cetype>

The **cetype** element is used to define the constraints of a specific Clinical Element Type. This element has 3 attributes, *name*, *base*, and *kind*, and an example of their use can be seen in Figure 3.27. The properties of **cetype** are listed in Table 3.15.

3.14.4 The Attribute *name* of <cetype>

The *name* attribute represents the required unique textual identifier for the **CEType** that is being modeled. This name must be unique within the context of all **CETypes**, and is eventually submitted to the vocabulary server to receive a unique concept code. Some examples include **VitalSignsPanel**, **SystolicBloodPressure**, and **AbnormalFlag**.

```
<ceml>
  <cetype name="MySystolicBP" base="Observation" kind="statement">
    ...
  </cetype>
</ceml>
```

Figure 3.27. A possible example of a Systolic Blood Pressure **CEType** definition inheriting the constraints of Observation.

Table 3.15. Properties of <cetype>

Property	Type	Cardinality
name	attribute	one
base	attribute	zero to one
kind	attribute	one
key	element	zero to one
data	element	zero to one
qual	element	zero to many
mod	element	zero to many
att	element	zero to many
item	element	zero to many
constraint	element	zero to many
link	element	zero to many

3.14.5 The Attribute *base* of <cetype>

The *base* attribute identifies an optional parent **CEType** from which this **CEType** will inherit. The **cetype** being defined inherits all parts of the *base*, but may override any part by restriction. If *base* is absent, then no constraints are inherited.[10]

3.14.6 The Attribute *kind* of <cetype>

The required *kind* attribute declares whether this **CEType** is a statement, panel, component, attribution, modifier, or is noninstantiable, as shown in Table 3.16. This represents the function of the **cetype** and affects the allowable persistence of instances in the datastore.

3.14.7 The Element <key> of <cetype>

The **key** element is a shortcut used to constrain the *key* of the CE Instance. One should remember that according to the Abstract Instance Model, *key* is defined as a **CWE**, so all the properties of **CWE** are available for constraint. The **key** element provides a shortcut to

[10] In the past, an absent base would inherit from a built-in **CEType** call **Root**. See the section titled Clinical Element Root for some historical information regarding this.

Table 3.16. Allowable values for <cetype> kind attribute

Value	Description
statement	A complete stand-alone medical sentence
panel	A collection of medical sentences
component	Part of a medical sentence
modifier	Modifying part of a medical sentence
attribution	Part of a medical sentence describing an Action
noninstantiable	A preliminary definition which must be subtyped for use

the modeler for two of the possible constraint paths, which are *key.code* and *key.domain* shown in Table 3.17. Examples of their use can be seen in Figure 3.28.

3.14.8 The Element <data> of <cetype>

The **data** element is a shortcut used to constrain the *data* of the CE Instance. According to the Abstract Instance Model, *data* can contain any of our datatypes, and a constraint is used to limit this to one or more datatypes. The **data** element provides a shortcut to the modeler for the constraint path *data.type* which can have the allowable values shown in Table 3.18. This shortcut can be seen in Figure 3.29. In addition, if the *data.type* is constrained to a **CWE**, then the *data.cwe.domain* constraint can also be collapsed into this shortcut shown in Figure 3.30. Both shortcuts are listed in Table 3.19.

Table 3.17. Properties of <key>

Property	Type	Cardinality
code	attribute	choice of one
domain	attribute	choice of one

```
<constraint path="key.code" value="RealWorldConcept_CODE"/>
<key code="RealWorldConcept_CODE"/>

<constraint path="key.code" value="RealWorldConcept_DOMAIN_CODE"/>
<key domain="RealWorldConcept_DOMAIN_CODE"/>
```

Figure 3.28. Constraint shortcuts for *key.code* and *key.domain*

Table 3.18. Allowable values for <data> *type*

Value	Description
cwe	Coded With Exceptions
cne	Coded No Exceptions
co	Coded Ordinal
pq	Physical Quantity
ivlpq	Interval Physical Quantity
rtopq	Ratio Physical Quantity
st	String
ed	Encapsulated Data
ii	Instance Identifier
int	Integer
real	Real Number
ts	Coded Simple

3.14.9 The Recursive Elements <qual>, <item>, <att>, and <mod>

The syntax for constraining quals, items, atts, and mods all use the identical shortcut syntax structure, but use the <qual>, <item>, <att>, and <mod> tag, respectively. Here we describe how to define <qual>, but the reader should realize that <item>, <att>, and <mod> follow the same pattern.

The **qual** element is the shortcut for the constraints of a qualifier. It contains 3 attributes, *name*, *type*, and *card*, as can be seen in Table 3.20. The *name* attribute is a unique textual identifier for this qualifier to be used in paths, and it is a shortcut for the path *qual.identifier*. The *type* attribute identifies the referenced **cetype** for this qualifier, and is a shortcut for the path *qual.identifier.type*. The *card* attribute specifies the cardinality of this qualifier, and

```
<constraint path="data.type" value="pq"/>

<data type="pq"/>
```

Figure 3.29. Constraint shortcut for *data.type*

```
<constraint path="data.type" value="cwe"/>
<constraint path="data.cwe.domain" value="MyDomain_CODE"/>

<data type="cwe" domain="MyDomain_CODE"/>
```

Figure 3.30. Constraint shortcut for *data.type* with domain constraint

Table 3.19. Properties of <data>

Property	Type	Cardinality
type	attribute	one
domain	attribute	zero to one

is the shortcut for the path *qual.identifier.card*. An example of this shortcut can be seen in Figure 3.31 and the allowable values are listed in Table 3.21.

3.14.10 The Element <constraint>

Regular constraints as defined in Strict CEML are still allowed. It must be remembered that Authoring CEML just contains additional shortcuts in addition to what is defined by the Abstract Constraint Model. For example, instead of using the **key** tag to constrain the key, it is still possible to use a plain old constraint, as seen in Figure 3.32.

3.14.11 The Element <absence>

Many times, we want default values to be indicated when a qualifier or modifier is not present in a CEInstance. For example, the modifier **Subject** defaults to a coded value

Table 3.20. Properties of <qual>

Property	Type	Cardinality
name	attribute	one
type	attribute	one
card	attribute	one

```
<constraint path="qual.myQualifier.type" value="MyQualifier"/>
<constraint path="qual.myQualifier.card" value="0-1"/>

<qual name="myQualifier" type="MyQualifier" card="0-1"/>
```

Figure 3.31. Constraint shortcut for *qual.identifier.type* and *qual.identifier.card*.

Table 3.21. Allowable values for *card* attribute in <qual>

Value	Description
0	Zero
0-1	Zero to One
0-M	Zero to Many
1	Exactly One
1-M	One to Many

of "Patient" when it is not present. To represent this, there is an **absence** tag with the properties of *path* and *value* shown in Figure 3.33.

3.14.12 The Element <link>

In Figure 3.34 is an example of the CEML syntax for a semantic link constraint. The **link** tag is used to represent the allowable semantic links. Each link is given a name, similar to how qualifiers, modifiers, and items are given a name. The allowed relationship is defined in the *relation* property with a code from the terminology server. At the moment, we do not have support for a domain of allowed relationships, and each relationship must

```
<ceml>
  <cetype name="MyType">
    ...
    <constraint path="key.code" value="MyType_ECID"/>
  </cetype>
</ceml>
```

Figure 3.32. An example of using a constraint instead of the key tag.

```
<ceml>
  <cetype name="Subject">
    ...
    <absence path="data.cwe.code" value="Patient_ECID"/>
  </cetype>
</ceml>
```

Figure 3.33. An example of using an absence tag.

```
<link name="myLinkName" relation="resultedFrom_ECID" card="0-1">
    <target path="type.code" value="PAIN_MODEL_ECID"/>
    <target path="key.code" value="Assertion_KEY_ECID"/>
</link>
```

Figure 3.34. The CEML Syntax to constrain allowable semantic links.

be specifically defined. The number of allowed links is defined with the *card* property. Individual constraint values within the target instance are specified with the *target* element, which contains a *path* and *value* property analogous to the *constraint* element.

3.15 Data Choice

Previously, we have not allowed choices for data types, and we forced the modeler to create two separate constraint types, one with each of the data types required. But we have reversed this decision, and we have introduced a syntax to allow data type choices within a single **CEType**.

For example, this allows a **CEType** to contain a choice between a **PQ** and a **CWE** datatype. This is useful when creating parent types such as a generic lab observation that will then be subtyped to more specific labs, as some specific labs report their values as a **PQ** and others as a **CWE** datatype.

The new syntax requires no change to the the CEML Schema, as the only change is in the value of the *type* property within *data*. In order to allow a choice, one must simply list each data type of the choice in an alphabetical comma delimited list, with no spaces. A list of our currently used choices is in Table 3.22. Figure 3.35 shows the declaration for a choice between **CWE** and **REAL**. Note that "cwe" must be placed before "real" because of the alphabetical rule.

3.15.1 Alphabetical Rule

The alphabetical rule for choices of data types functions to create a unique string for each choice type, so it is easy for us to identify all **CWE**, **REAL** choices. Without the alphabetical rule, there would be 2 possible strings which represent this same choice. And if the choice contained 3 types, then we would have 6 possible strings.

Table 3.22. The current list of Datatype Choices.

Value
cne,st
cwe,pq,st
cwe,real
cwe,st

```
<data type="cwe"/>

<data type="cwe,real"/>
```

Figure 3.35. Declaring a datatype value choice.

3.15.2 Use Requirements

A datatype choice can only be used when the following two conditions are true.

1. All of the components of the model (qualifiers and modifiers) must be valid for a CE Instance that uses any of the possible data types in the choice.
2. The domains of those components are not constrained differently depending on the data type. If either of these rules does not apply, the data type choice should not be used.

If the above conditions are not both true, then instead, multiple **CETypes** should be created, each with their own datatype, for example, if there is a **CEType** with a data choice between **CWE** and **PQ**, and this also has two qualfiers, *qual1* and *qual2*. If *qual1* were only valid for a CE Instance using **CWE** and *qual2* was only valid for a CE Instance using **PQ**, then this would fail the first condition. So, instead of creating a **CEType** referencing both *qual1* and *qual2* with *data* constrained to a *type* of "cwe,pq," two different **CETypes** should be created; one with *data* constrained to **CWE** and with a reference to *qual1*, and the other with *data* constrained to **PQ** and with a reference to *qual2*.

Another example would be if *qual1* is valid for both a CE Instance of *type* **PQ** and also for a CE Instance of *type* **CWE**. However, the *data* in *qual1* needs to be constrained to "domain1" when a CE Instance uses **PQ**, and the *data* in *qual1* needs to be constrained to "domain2" when a CE Instance uses **CWE**, then this would fail the second condition. So, instead of creating a single **CEType** with *data* constrained to a *type* of "cwe,pq," two different **CETypes** should be created: one with *data* constrained to **CWE** and with *qual1*'s *data* constrained to "domain1," and the other with *data* constrained to **PQ** and with *qual1*'s *data* constrained to "domain2."

3.16 CEML Tutorial

To better understand how to define a **CEType** with CEML, it is probably easiest to just give a few simple examples and then describe the example. These examples will be using the Authoring CEML syntax. First, let us start by describing a type for a Blood Pressure Panel in Figure 3.36. Remember that in a Clinical Element Instance, the value choice can contain either a list of child Clinical Element Instance nodes or an HL7 datatype. In this

```
<ceml>
  <header>
        ...
  </header>
  <cetype name="BloodPressurePanel" kind="panel">
    <key code="BloodPressurePanelKey_CODE"/>
    <item name="systolicBloodPressure" type="SystolicBP" card="0-1"/>
    <item name="diastolicBloodPressure" type="DiastolicBP" card="0-1"/>
  </cetype>
</ceml>
```

Figure 3.36. A Blood Pressure Panel with no Qualifiers

first example, we create a **CEType** that constrains the instance to a value choice of *items* which contains two children: one for systolic blood pressure and one for diastolic blood pressure.

Let us examine the parts of this **CEType** definition in Figure 3.36. Every definition is defined using a **ceml** element as the root element. Inside of the **ceml** element, we have one **header** element and one **cetype** element. We will not be discussing the **header** element in this section, so let us focus on the **cetype** element.

The **cetype** element is the basis of the Clinical Element Constraint Type. The *name* attribute of **cetype** will act as a globally unique name for the **cetype** we are creating with this definition. This **cetype** has been declared a "panel" using the *kind* attribute. A list of the allowable attributes in **cetype** can be seen in Figure 3.37.

Next, inside the **cetype**, other constraints can be declared using various elements. In our example in Figure 3.36, there are two types of elements within the definition. The first is a **key** constraint and the second is two **item** constraints.

- **name** - The name of the cetype being declared.
- **base** - The name of a parent cetype.
- **kind** - Identifies the function of this cetype.

Figure 3.37. Attributes of cetype

In the **key** element, the allowable key code for an instance can be restricted to a particular code, as it is here using the *code* attribute. An alternative to using the *code* attribute is to use the *domain* attribute which restricts the allowable key code in an instance to be within a certain domain of codes. The attributes of the **key** element are listed in Figure 3.38.

Next, in our BloodPressurePanel example in Figure 3.36, there are two **item** constraints. These **item** constraints declare the child Clinical Elements that are required or allowed in the BloodPressurePanel we are defining. Each of the **item** constraints has a *name*, *type*, and *card* attribute. The *type* attribute defines the referenced **cetype** to constrain the child, and the *name* attribute assigns a local name to this type. This is just like the normal variable-type assignments used in most programming languages. The *card* attribute is set to "0-1" for both of the examples, and this defines the number of each child type that can occur in the CE Instance. The attributes available in an **item** constraint are listed in Figure 3.39.

3.16.1 Data

In the last example, we saw a **cetype** defined where the value choice was two child Clinical Elements rather than an HL7 datatype. So, in this next example seen in Figure 3.40, we define a Systolic Blood Pressure **cetype** that uses an HL7 datatype. In the **data** definition, there is an attribute called *type* which is used to declare the HL7 datatype to be used. In this example, it has been declared to be an **HL7:PQ**.

3.16.2 Qualifiers

The examples in Figures 3.36 and 3.40 only declared constraints of the *key* and value choice (*data* or *items*). Next, in Figure 3.41, we have added qualifier constraints pertinent to the systolic blood pressure. These **qual** constraints have the same attributes as the **item** constraints as seen in Figure 3.42.

- **code** - Restricts all instances of this type to have this key code.
- **domain** - Restricts all instances of this type to have a key code within this domain.

Figure 3.38. Attributes of the key constraint

- **name** - The local name of this item.
- **type** - The Clinical Element type of this item.
- **card** - Restricts the number of occurrences for this item.

Figure 3.39. Attributes of the item constraint

```
<cetype name="SystolicBP" base="Observation" kind="statement">
  <key code="SystolicBPKey_CODE"/>
  <data type="pq"/>
</cetype>
```

Figure 3.40. A Systolic Blood Pressure definition with no Qualifiers

```
<cetype name="SystolicBP" base="Observation" kind="statement">
  <key code="SystolicBPKey_CODE"/>
  <data type="pq"/>
  <qual name="methodOrDevice" type="MethodOrDevice" card="0-1"/>
  <qual name="bodyPosition" type="BodyPosition" card="0-1"/>
  <qual name="vascularBodySite" type="VascularBodySite" card="0-1"/>
  <qual name="breathingPhase" type="BreathingPhase" card="0-1"/>
</cetype>
```

Figure 3.41. A Systolic Blood Pressure definition with Qualifiers

These constraints for qualifiers will allow the CE Instance to contain information about the measured systolic blood pressure, such as indicating the body position of the patient during the measurement, and what method or device was used to make the the systolic blood pressure measurement.

3.16.3 Modifiers

In Figure 3.43 is an example of a modifier constraint pertinent to the systolic blood pressure or any clinical element. This **mod** constraint has the exact same attributes as the **qual** structure. The modifier in the example is called Subject, and is used to represent the

- **name** - The local name of this qual.
- **type** - The cetype of this qual.
- **card** - Constrains the number of occurrences for this qual. Elements.

Figure 3.42. Attributes of the qual Constraint

```
<cetype name="SystolicBP" base="Observation" kind="statement">
  <key code="SystolicBPKey_CODE"/>
  <data type="pq"/>
  <qual name="methodOrDevice" type="MethodOrDevice" card="0-1"/>
  ...
  <mod name="subject" type="Subject" card="0-1"/>
</cetype>
```

Figure 3.43. A Systolic Blood Pressure definition with a Modifier

subject of this systolic blood pressure. In the CE Instance, it would have a value of "self" to indicate that the measurement was taken on the patient.

3.17 Understanding Instance Data

We have discussed that CEML is used to author the Clinical Element Constraint Types which contain rules that constrain the values in a Clinical Element instance. But what is instance data? People outside of computer science seem to have a hard time understanding what we mean by instance data. Instead of a formal definition, it is easier to understand instance data by example.

In the following example, an English sentence is used to state a definition or constraint rather than CEML. The constraint is followed with examples of many instances which conform to the constraint, each also represented as an English sentence.

Constraint A blood pressure panel contains a Systolic Blood Pressure and a Diastolic Blood Pressure each with a numerical value and with units of mmHg. The values should never be negative. In addition, you can specify the patient's body position.

Instances ...

- 120/80 mmHg
- 142/101 millimeters Mercury
- 114/68 mmHg while patient was sitting
- 132/96 mmHg while patient was standing

From this example, it should be clear that you have ONE constraint definition and you can then have an UNLIMITED number of instances which conform to the definition. Every time a blood pressure measurement is taken on a patient, a new instance is generated. These instances are then stored and make up a patient's electronic medical record, available for later retrieval.

3.17.1 Constraints and Instances

There are many existing formalisms that allow one to state a constraint definition, and then create many instances that conform to this definition (Figure 3.44). For example, XML Schema can be used to declare a constraint definition and then many XML documents can be created as instances which conform to the XML Schema definition. ASN.1 source is also used to declare a definition and then instances exist as BER strings. Java source code is used to declare a definition or Class, and then many instances of that class can be created.

3.17.2 Serialization

Frequently, you will hear the term serialization used when reference is made to instances. When many instances are created that conform to a definition, these instances must be able to be used by computer software. There also must exist a way to send this instance over the network as well as a way to store it.

An XML document is as an example of serialized data that conform to an XML Schema. An ASN.1 BER string is an example of serialized data that conform to an ASN.1 source definition. It is our intent that Clinical Element Model data can be serialized in many forms and ultimately conform to a single CEML definition.

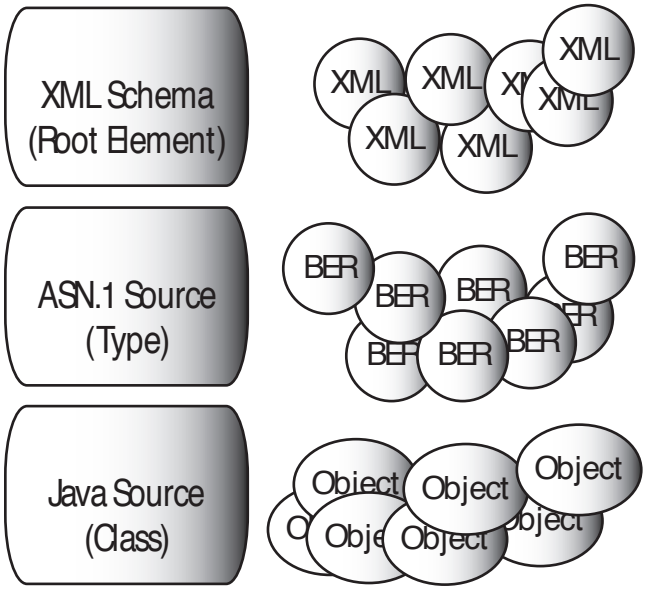


Figure 3.44. Formalisms to define constraints/rules for Instances.

CHAPTER 4

TRANSFORMATION OF DETAILED CLINICAL MODELS

4.1 Introduction

A transformation is the process of taking a structured instance of data and then processing the data to create an instance of data with a new structure, which is depicted in Figure 4.1. The input source can be external data such as an HL7 message, or even an institution's internal data representation. There are many situations that result in the need for a transformation. First, a need for transformation occurs when an interface receives an HL7 message and it must be transformed into the institution's internal data structure. Another need for a transformation occurs in the opposition situation, where an institution's internal data structure must be exported as an HL7 message. Lastly, there are many intra-institution forms of clinical data which may need conversion from one form to another.

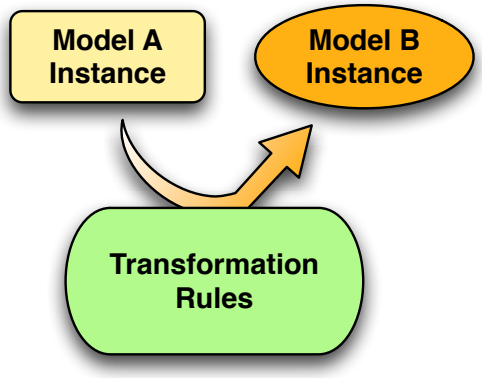


Figure 4.1. Instance Data Transformation.

Our internal data model is called The Clinical Element Model. The use of the Clinical Element Model results in patient data instances being created and stored in accordance with a particular Clinical Element Constraint Model called a CEType. Due to the permanent storage of these instances, the CETypes are also called our storage models.

Despite the hopefully well thought out design of these models, there are still many use cases which necessitate an intra-institution transformation. These include the need for a simple code change in stored data instances, structural changes to constraint models, and finally, the need for alternative transient forms[1] of a model to facilitate ease of use. Transformations can either occur in real time as needed, or be executed against all stored instances in the data store to create a permanent change. We begin by discussing these use cases for transformation, and in the process discover the requirements we need in a transformation process.

4.2 Transformation Use Cases

In this section, the use cases for transformation will be examined.

4.2.1 External to Internal

External to Internal Transformation is the use case of an interface engine. The common use case is an HL7 message comes from an external system and must be stored in the internal system in another format. This is depicted in Figure 4.2.

4.2.2 Internal to External

Internal to External Transformation is the use case when the internal system must export a common format to an external system. The common use case is an HL7 message is generated from the internal representation. This is depicted in Figure 4.3.

---

[1] CEMorph is the term given for our alternative constraint models to represent data instances for transient use.

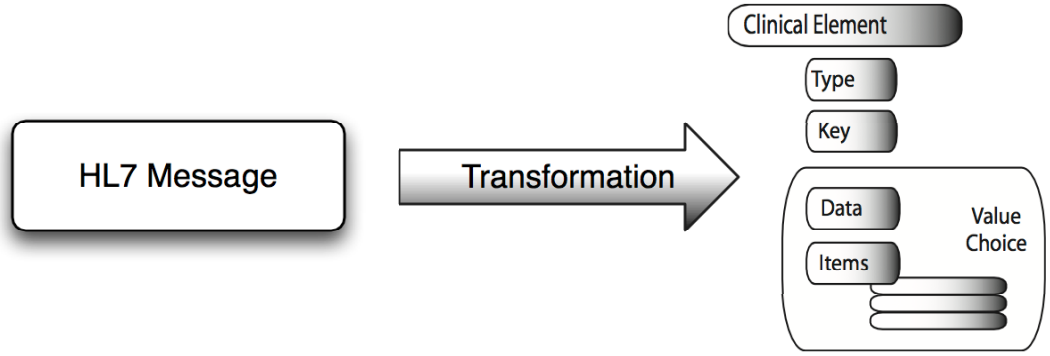


Figure 4.2. External to Internal Transformation.

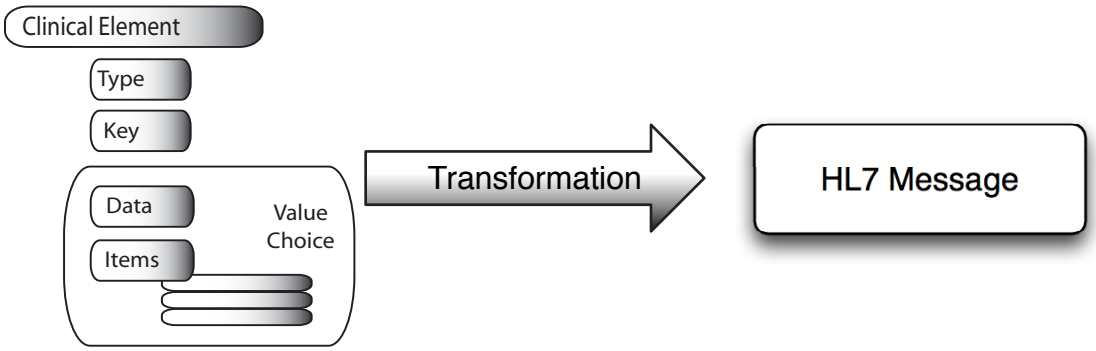


Figure 4.3. Internal to External Transformation.

4.2.3 Internal to Internal

Internal to Internal Transformation is the use case when the internal system requires a change. This is depicted in Figure 4.4. There are three common situations where these internal transformations are needed: when a code changes, when the structure of a CEType changes, and to support transient alternative forms of data which we call CEMorphs. These are listed in Table 4.1 and discussed in the following paragraphs.

One common internal change is the need for a code change. The Clinical Element Data Model creates a structural framework into which codes can be placed. These codes are then validated by a particular Clinical Element Constraint Model called a CEType. The codes

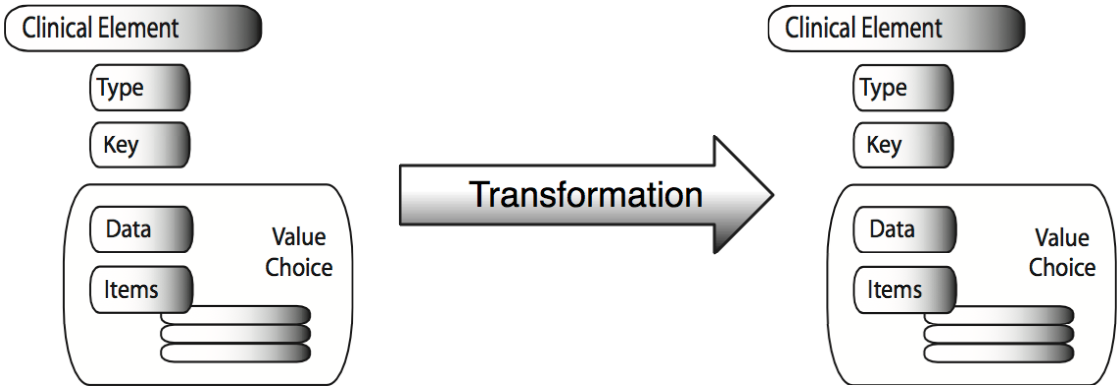


Figure 4.4. Internal to Internal Transformation.

that are used to represent a concept within this structural framework will sometimes change, and this change affects all data instances that have been stored using the old code. There are two ways to handle this situation of a changed code. First, the vocabulary server can dynamically handle the translation of the code in real time, or secondly, a transformation engine can handle the change, either in real time or the stored data can be permanently transformed and all instances containing the old code are permanently updated.

Another common internal change is a the need for a structural change. When a CEType needs to be changed in such a way that it requires structural changes, then the vocabulary server will no longer suffice to handle the outdated data instances, and a transformation engine is required. Again, the transformation engine can either do this in real time or the stored data instances can be permanently transformed.

There are three types of possible transformations that may need to take place. The difference between these lies only in the directional association such as one-to-one, one-to-many, or many-to-one.[2]

- CEType to CEType Transformation
- CEType to CETypes Transformation
- CETypes to CEType Transformation

[2]In a previous decision, it was decided to not allow one-to-many, or many-to-one transformations involving CEMorphs, but this decision did not apply to CEType to CEType transformations.

Table 4.1. Types of Changes in a Transformation

Use Case	Description
Code Change in CEType	Often the code for a concept is changed. This can often be transiently handled by the vocabulary server and stored instances do not need to be changed, but a transformation can also be used to permanently update all stored instances.
Structural Change in CEType	Modeling is difficult to get correct on the first attempt, and when a structural update is required, a transformation is needed for any stored instances.
CEMorph	No model is ever actually correct, and different users may prefer one design over another model's design. Transformation of a storage instance into various transient instances allows different users to work with the data as they see fit. CEType Models are storage models, and instances that conform to these models can be transformed into transient forms defined by CEMorph models.

One issue you must deal with in these transformations is data loss from the original instance. This occurs because the newly designed model defines less information than the original model. This issue must be addressed at the time when a new model is being designed; thus, a thorough knowledge of transformation issues is required when models are updated and redesigned in a backward incompatible way.[3]

A corollary to the above issue is when the new model strictly requires information that the original data instances do not contain, and whether or not this information can be obtained from some other source. Again, this issue must be addressed during the remodeling phase.

---

[3]It is possible to update a model by addition, which has no affect on existing stored instances, and thus does not require a transformation.

Another issue that comes into play is maintaining unique instance identifiers through the transformation. It is very important if other data are linked to the data being transformed by the instance identifier, one example being through semantic links. This is much easier to deal with in a one-to-one transformation, but becomes difficult with one-to-many and many-to-one transformations.

In addition to our storage form models called CETypes, we also have models which will hold temporary instance data, and these are called CEMorphs. The CEMorph is modeled identically to a CEType with the same CEML formalism to describe the model. The only difference is that one is designated as a CEMorph, and the other is designated as a CEType.

During the modeling phase, for any particular clinical concept, many alternative methods to model the concept are explored. The problem is that no model will ever be fully correct, because the various potential users of the model may prefer one alternative over another. To rectify this situation, we have stated that for any conceptual model, we will designate one model as the permanent storage form, and this will be modeled as a CEType. But in addition, alternative forms of the model will exist and be modeled as CEMorphs, and data instances will only transiently exist in these CEMorph forms. Thus, transformations are required to transform the instance data from the CEType form to a CEMorph form and vice versa. Each CEType can have many CEMorph alternative forms, and in the end, we have an isomorphic family of models for each CEType storage form.

Since CEType to CEMorph instance transformations occur in real time, we do not want to deal with the complexities involved with one-to-many and many-to-one transformations. For this reason, any CEType to CEMorph transformation can only have a one-to-one directional association.[4]

One common user of CEMorphs will be user interface designers. User interface widgets will, of course, be bound to data instances of the various CEType Models. But it is often the case the UIWidget designer would like to deal with the model in a way that makes it difficult to use the data as it exists in a CEType. In these cases, the user interface designer

[4]As previously stated, one-to-many and many-to-one transformations are allowed in CEType to CEType transformations. However, that use case involves a model undergoing permanent change, thus dealing with the complexity involved with the transformation will only cause difficulties temporarily, until all instances have been converted.

would bind the widget to a CEMorph type, and all the transformation issues are taken care of behind the scenes.[5] If the UIWidget designer only wishes to not show some of the data from a CEType instance, this is not a reason to design a CEMorph model, as a UIWidget is not required to display all data in a CEType instance.

4.2.4 External to External

External to External Transformation can be seen in an interface engine. The common use case is an HL7 message comes from an external system and must be preconditioned in some way before the HL7 message is sent to another part of the interface. This was seen at Intermountain Healthcare in the interface for microbiology data. 3M had built an HL7 interface that the Intermountain developers did not want to rewrite, so to handle decomposition of microbiology data, the decomposition was handled in the HL7 message prior to forwarding it to the 3M service. This is depicted in Figure 4.5.

4.3 Composition and Decomposition

Transformation is an easier task when the source form and the target form are similarly structured, but this is rarely the case. The most difficult situation arises when either the source model or target model contains an aggregate of multiple parts from the other model.

A coded concept can be assigned to any real-world concept. These real-world concepts may be atomic concepts such as **ARM** and **LEFT**. Or, these real-world concepts may be

[5]Programatically, a CEMorph instance needs to record what CEType instance it originated from, if it will be used for updates.

Figure 4.5. External to External Transformation.

composed of multiple atomic concepts to create a single composed concept such as **LEFT ARM** and **RIGHT ARM**.

Composition occurs when multiple coded concepts are combined into a single coded concept that contains the combined meaning of the individual concepts. Composition occurs when the individual concepts of **LEFT** and **ARM** are combined into the single coded concept of **LEFT ARM**. Decomposition occurs when a single coded concept is divided into multiple coded concepts which represent parts of the original single coded concept. Composition and Decomposition are summarized in Table 4.2.

The problem with composed concepts is that they do not lend themselves to query and comparison between instances of stored data. For example, if two instances of **SystolicBloodPressure** were stored, one with a body location of **LEFT ARM** and the other with **RIGHT ARM**, then a search for **SystolicBloodPressure**s taken on the **ARM** would yield no results, despite the fact that both these measurements were taken on the arm.

It is the responsibility of the modeling team to design a **SystolicBloodPressure** that meets the needs of the users. They may decide that it is unimportant to decompose body location and thus model body location with only a value set of decomposed codes. If this is the case, then any composed codes that come from an external interface or even an internal user interface must be decomposed before they can be stored. The reverse is also true; if **SystolicBloodPressure** is modeled with composed codes, but is receiving decomposed codes from an external interface, then these decomposed codes must be composed before they can be stored.

Since understanding the problems of composition and decomposition are a prerequisite for understanding model transformation, the following sections will explore composition and decomposition before returning to the more general problem of model transformation.

4.4 Current Decomposition

Currently, Intermountain Healthcare handles concept decomposition for the use case of transforming incoming HL7 messages from external systems. Intermountain has not implemented any form of composition at this point.

Table 4.2. A few simple definitions

Term	Defition
Composition	The combination of multiple coded terms into a single aggregate code.
Decomposition	The division of a single aggregate code into multiple coded terms

Intermountain has implemented a rule-based system for decomposition. When this system is presented with an incoming code and a given interface context, it will return a rule that describes how to decompose the concept into parts and where to place each part in either a specific HL7 version 2 message or in a specific storage ASN.1 message. Examples can be seen in Figures 4.6 and 4.7. A specific example showing the decomposition of Left Arm into an HL7 message can be seen in Figure 4.8

This is implemented by using a special decomposition database table within the terminology server, as can be seen in Figure 4.9. This table is called COMP_DECOMP, but Intermountain has only used the table to implement decomposition.[6] The columns in this table used to implement decomposition are described in Table 4.3.

[6]Although composition was never implemented by Intermountain, this table was created with composition in mind, as can be seen by the columns TO_ATOM_NCID and TO_MOLECULE_NCID.

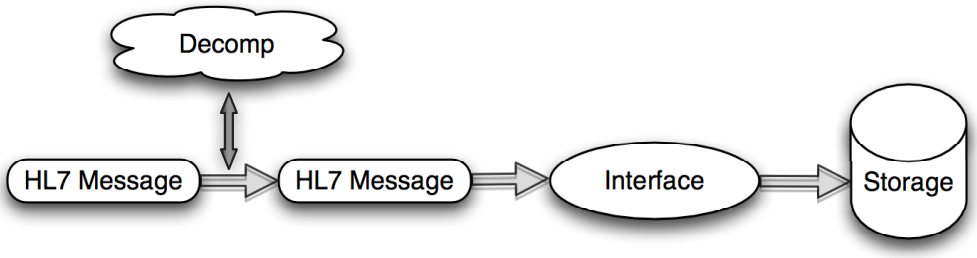


Figure 4.6. Decomposition targeting HL7.

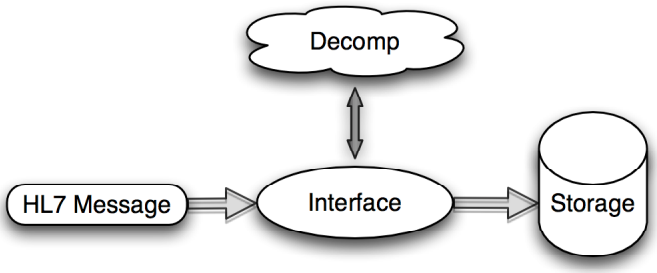


Figure 4.7. Decomposition targeting ASN.1.

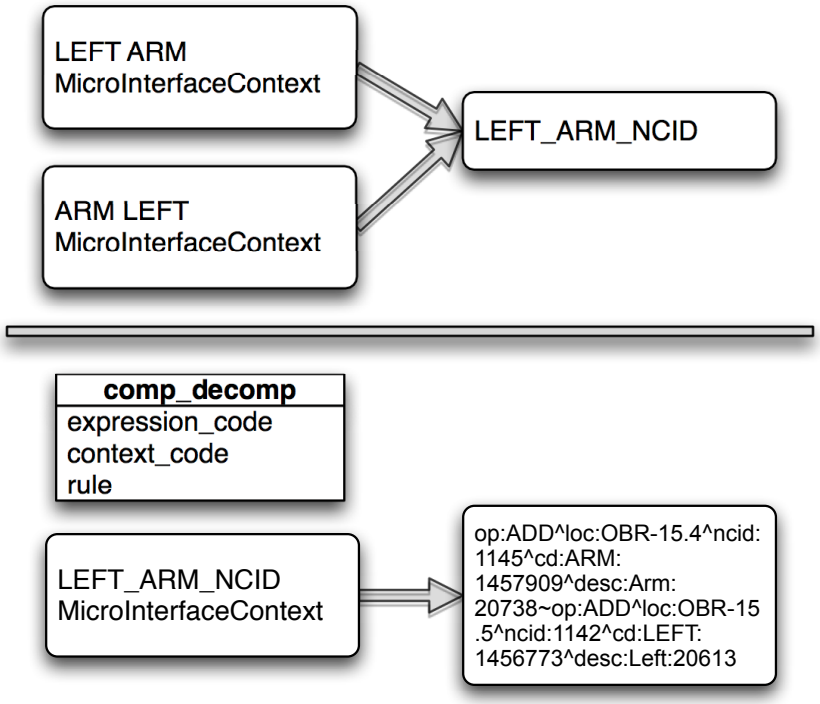


Figure 4.8. Precoordinated Left Arm decomposed into an HL7 Message.

Column Name	ID	Pk	Null?	Data Type	Default	Histogram
EXPRESSION_NCID	1	1	N	NUMBER (20)		Yes
ASN1_TYPE_NCID	2		Y	NUMBER (20)	NULL	Yes
MESSAGE_CONTEXT_NCID	3	2	N	NUMBER (20)		Yes
TYPE_OF_EXPRESSION_NCID	4	3	N	NUMBER (20)	6490614	Yes
TO_ATOM_NCID	5	4	N	NUMBER (20)	1022	Yes
TO_MOLECULE_NCID	6	5	N	NUMBER (20)	1023	Yes
CREATED_BY_NCID	7		Y	NUMBER (20)		Yes
ENTERPRISE_NCID	8		Y	NUMBER (20)	211	Yes
STATUS_NCID	9	6	N	NUMBER (20)	1024	Yes
RULE	10		Y	VARCHAR2 (4000 Byte)		Yes
COMMENTS	11		Y	VARCHAR2 (4000 Byte)	NULL	Yes

Figure 4.9. Comp Decomp Table.

Table 4.3. Columns for Decomposition

Column	Definition
EXPRESSION_NCID	The composed concept code
MESSAGE_CONTEXT_NCID	The context of the decomposition, such as the Microbiology Interface
RULE	The rule that contains the decomposition parts and where they are placed within the target model.

The reason that some rules target HL7 messages is because 3M had built interfaces which accept HL7 messages and target an ASN.1 message. Rather than rewriting these interfaces to handle decomposition, Intermountain chose to handle decomposition in the HL7 message before passing the message to the 3M service. One problem here is that the interface coder has little control over where the decomposition parts are placed in the target ASN.1 model. Basically, the rule writer knows where 3M writes an OBR segment to in the ASN.1, so the rule writer takes the decomposed parts and assigns them to the appropriate OBR segment. These mappings can be seen in Table 4.4.

Where Intermountain was responsible for writing the interface, the rule-based decomposition system would target the ASN.1 model directly, which gives the decomposition rule writer more direct control of where the parts go.

Table 4.4. Micro HL7 OBR to ASN.1 Mapping

OBR	ASN.1
15.1	SpecimenType
15.4	BodyAnatomy
15.5	BodySiteModifier
15.6	SpecimenModifier

4.5 Decomposition Research

There are a few options for decomposition which are described in the following sections. One of these options could be chosen or a combination of a few could be chosen.

4.5.1 Relations

Decomposition can be implemented with concept relations in a vocabulary server. The Composed Code will have multiple entries in a 3-part relationship table, with the fields source concept, relationship concept, and target concept.

One way to implement relations is when the relationship concepts are given meaning such as **hasBodyPart** and **hasLaterality**, which can be seen in Figure 4.10. This scheme allows one to ask questions of the composed concept, such as retrieving the laterality of the concept. It is also possible to retrieve all the relationships for a concept.

One problem that can be seen in Figure 4.10 is that a relationship such as **hasBodyPart** can return more than one target concept. In this example, it can be seen that both **Leg** and **Lower Leg** will be returned for **hasBodyPart**. Another problem is that if you want to retrieve all the atoms that make up the composite, it is impossible because the composite concept can have many more relations than the ones that simply describe the atoms of the composite.

Another option is to create relation concepts such that they return the atoms of a composite, rather than denoting meaning, as shown in Figure 4.11. The problem here would be that it would take a further step to ascertain the meaning of the target concept.

Source Concept	Relation Concept	Target Concept
Left Lower Leg	hasBodyPart	Leg
Left Lower Leg	hasLaterality	Left
Left Lower Leg	hasBodyPartQual	Lower

Left Lower Leg	hasBodyPart	Lower Leg
Left Lower Leg	hasBone	Tibia

Figure 4.10. Meaningful Relation Concepts.

Another option that solves the previous problem is to precoordinate atom or molecule with a meaningful relationship concept which results in such relationships as **hasBodyPartAtom**.

There are some complex composites such as **ABCESS 4TH TOE OF LEFT FOOT**, that if you break the composite into all atoms, then you will no longer understand the relationship between the atoms. For example, does **4th** go with **FOOT**?

4.5.2 Model Populating Rules

This is the current implementation of Intermountain's decomposition engine. A composed code points to a rule. The rule contains a syntax that contains the atoms and or molecules, and their target location in a specific model, which in our case would be a CEType.

This is the simplest and most straightforward approach to the problem. As long as the rule writer does not make a mistake, it guarantees that the composite will be written correctly to the target model.

One drawback is that the rule is only good for the target model, and if you want decomposition for some other use, then you must write a new rule for a new target, and place that within a new context. Another problem occurs when building rules for two

Source Concept	Relation Concept	Target Concept
Left Lower Leg	hasAtom	Leg
Left Lower Leg	hasAtom	Left
Left Lower Leg	hasAtom	Lower

Leg Lower Leg	hasMolecule	Lower Leg

Figure 4.11. Atom and Molecule Relations.

different interfaces; if even one composite rule is handled differently, then all the rules must be copied and placed in a new context. One solution to this duplication problem would be to allow multiple contexts to register with a single rule, and in this way, the odd rule would be written and no duplication would be needed for the other existing rules.

4.5.3 Generic Rules

One could think of these as a combination of Rules and Relations, but instead of existing in a relationship table, all the relations are formatted into a parsable rule or string. And even better, the rule can contain hierarchy so that even composites such as **ABCESS 4TH TOE OF LEFT FOOT** can be handled correctly.

4.5.4 Analysis

I think relations are too risky, as things would quickly become too complicated as interface and transformation writers had to handle all the exceptions that may occur with various composites that currently exist, and all those that will be created in the future.

Model Rules are easy, and they work. They are not elegant, but they will get the job at hand accomplished. One downside is that if a model changes, all the rules must be changed, since the rules target the old model.

In trying to develop a demonstration syntax for Generic Rules, I stumbled onto a simple answer after months of thinking about this whole problem. We already have a generic syntax to describe structured data, its the language we use to describe detailed clinical models. So basically, the answer would be to store a CE Instance as the generic rule for a composite code. And it is not so much a rule anymore, but rather a piece of data that the interface writer can use. In order for this to work, we would need to allow storage of stand-alone qualifier instances in the rule repository, but I see that as not a problem. An example can be seen in Figure 4.12.

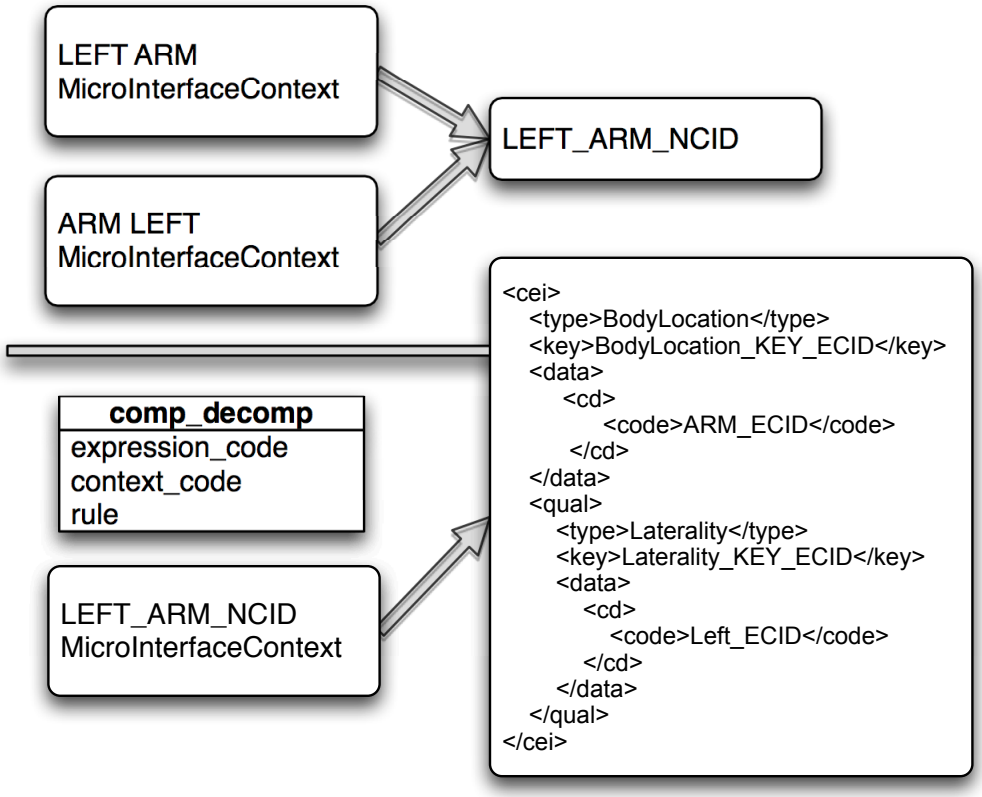


Figure 4.12. Generic Rule.

4.6 Composition Research

Composition results in a harder problem than decomposition. If one tries to use a relation model, the query to retrieve the composite becomes burdensome. The key to any solution is too generate a unique search key from multiple concepts that will then point to the composite concept. I list three solutions to the problem, the first being Ascending Strings followed by Concept Prime, with the final solution to be presented at the end of this section.

4.6.1 Ascending Strings

Ascending Strings creates a search term by ordering the multiple concept identifiers into one string. The trick of ascending strings is to order the UUID for each concept into one string with ascending values. In this manner, there will be only one key created for any collection of multiple concepts. One problem with this technique is the length of the key generated.

4.6.2 Concept Prime

With the Concept Prime technique, each concept is assigned a unique prime number which is called the Concept Prime. The key for the composite is then created as the multiplicative product of the individual concept primes, as shown in Figure 4.13.

A benefit with this technique is that the key remains relatively small when compared to Ascending Strings. For example, even with a terminology repository of 50 million concepts, each with an assigned prime, the resulting MAX key of the five largest primes is only 46 digits long. This is still larger than Oracle's number column limit of 38 digits, so a text field index would be necessary for the key. The key is thus much smaller than with Ascending Strings, but the problem here lies with the complexity to set this system up, such as assigning unique prime numbers.

4.6.3 Working Solution

Ascending Strings and Concept Prime both have a common problem. They both suffer from ambiguous composition, where concepts such as LEFT ARM RIGHT LEG, and

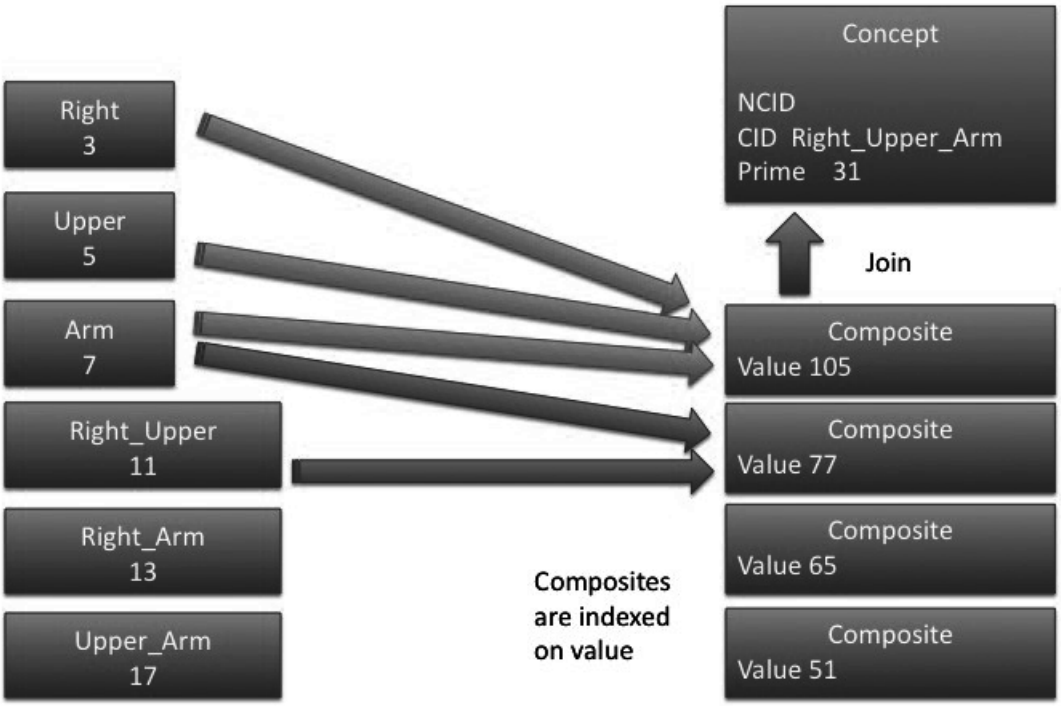


Figure 4.13. Concept Prime Composition.

LEFT LEG RIGHT ARM both result in the same key being generated, as shown in Figure 4.14.

Trying over and over to solve this problem, I finally realized that it would be possible using the Ascending Strings technique, if contextual structure was kept intact. Thus, a syntax would be required, at which point I realized the problem was already solved. For decomposition, I had recommended the use of Generic Rules which contain CE Instances. Thus, the solution is to index this generic rule column, which allows one to build a post-coordinated CE Instance and return the composite concept, as shown in Figure 4.15. Thus, there is one table to rule them all.

The only requirement here is that a strict serialized CEInstance textual syntax is finalized, and it is adhered to character for character so that an Oracle Text index will work in reverse. But since code will generate the serialized CEInstance textual syntax, this should not be a problem. It would also be possible to use a less verbose syntax just for this purpose.

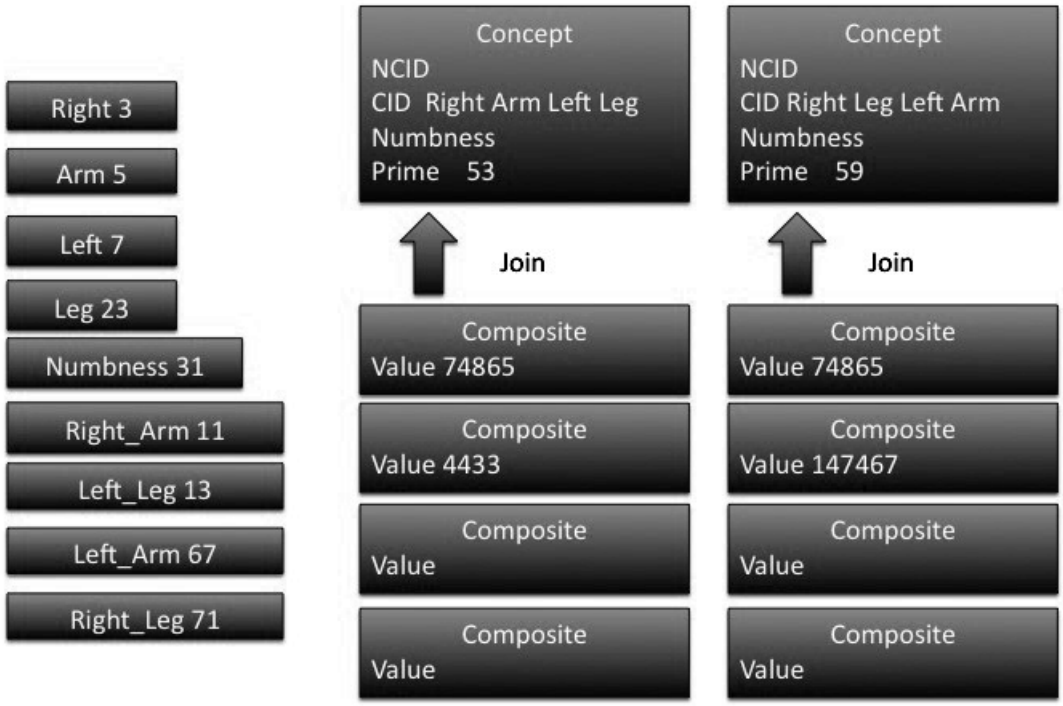


Figure 4.14. Ambiguous Composition.

A lingering problem is that for a transformation, sometimes we need composition that does not quite make sense from a model perspective. For example, say one needed to compose DeepTendoReflex_KEY_ECID with Patella, as compared to DeepTendonReflex model with Patella. This lingering problem basically means we will need to create models that in some cases are only created to compose and decompose.

4.7 Explicit vs. Semi-Explicit Transformation

When defining the mapping in a transformation, it is possible to do this completely explicitly, where all the mapping information exists in the transformation rules. Or another option is to define the transformation mapping semi-explicitly where some of the mapping information exists in the transformation rules, and some exists externally in the terminology server. The following examples are not meant to be a syntax, but are intended to address the issues our syntax, called Clinical Element Transformation Syntax (CETL), will need to

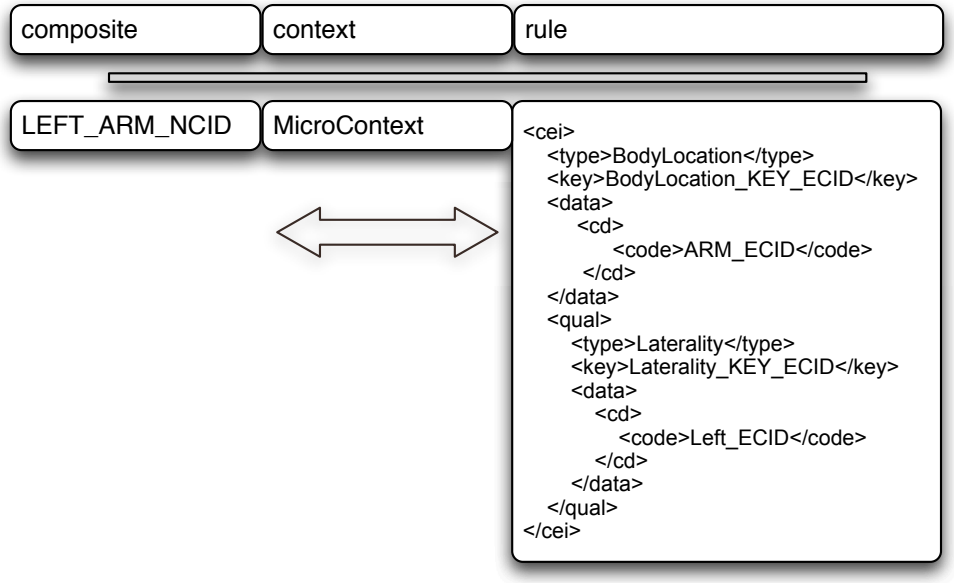


Figure 4.15. Bidirectional Comp/Decomp Table.

address. Based on the above description, we have created the terms Explicit Transformation and Semi-Explicit Transformation.

4.7.1 Explicit Transformaton

With an explicit transformation, all the information for the transformation is represented in the rules of the transformation. No external information is needed, as depicted in Figure 4.16.

4.7.2 Semi-Explicit Transformation

Semi-explicit transformation uses external information in the terminology server in addition to the rules in the transformation map, as depicted in Figure 4.17. The terminology server requires a decomp table with the parts of decomposition labeled. These labels will be used in the transformation rules.

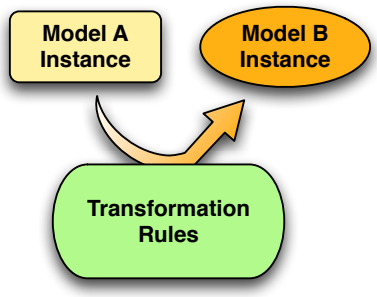


Figure 4.16. Explicit Transformation.

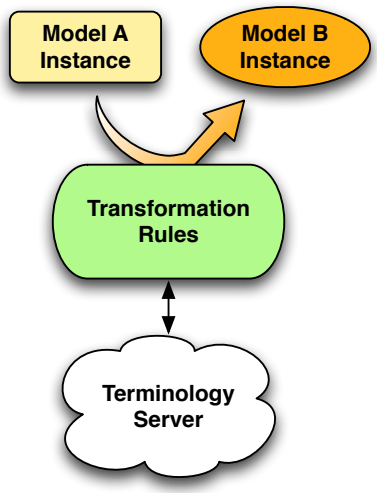


Figure 4.17. Semi-Explicit Transformation.

4.7.3 What's Next?

In the following sections, we will declare some example constraint models, and then describe what is needed in a transformation language to handle the transformation between data instances that conform to these models. For brevity, the models will be described with a stripped-down pseudo Clinical Element syntax. The transformations will also be

described using pseudo-language whose purpose is to illustrate the various problems we will encounter. Later in this document, we will define a true syntax based on our findings here. In the transformation examples, we will be using the term *src* to indicate the source data instance and the term *dst* to indicate the destination instance. The destination instance is the data instance that is created as a result of the transformation. The path statement used to reference subtrees is the same as described in the Clinical Element Modeling Language.

4.8 Transformation Examples

4.8.1 Case 1: Key + Qual to Determinate Precoordinated Key

Use case 1 will use the models shown in Figure 4.18 which lists two constraint models for deep tendon reflex. Both of these models convey similar information, except the DeepTendonReflex model has a qualifier called *bodyLocation* which gives it the ability to describe the deep tendon reflex of different body locations such as the ankle, triceps, or patellar tendon. The model called PatellarDeepTendonReflex, on the other hand, is limited by its precoordinated *key* to only describe the deep tendon reflex of the patellar tendon.

```
DeepTendonReflex
    key  DeepTendonReflex_KEY_ECID
    qual bodyLocation
    data co

BodyLocation
    key  BodyLocation_KEY_ECID
    qual laterality
    data cwe

PatellarDeepTendonReflex
    key  PatellaDeepTendonRelex_KEY_ECID
    qual laterality
    data co
```

Figure 4.18. Models for Use Case 1 and 2.

The transformation shown in Figure 4.19 is an explict transformation of DeepTendonReflex to PatellarDeepTendon Reflex. This transformation has a *src*[7] of DeepTendonReflex and a *dst* of PatellarDeepTendonReflex. The transformation involves combining the *key* and the *qual bodyLocation* into the precoordinated *key* value for patellar deep tendon reflex.

In this transformation, not all data instances that conform to the DeepTendonReflex constraint model can be transformed. It is quite easy to see that only those instances that have a bodyLocation of PatellarTendon can be transformed into an instance of PatellarDeepTendonReflex. Because of this situation, our transformation language requires the ability to check whether an instance can be successfully transformed according to its data content. In the example, lines 1 and 2 describe the required source and destination models. Line 4 is a check whether the data instance contains the key code for deep tendon reflex. Line 4 in this case is redundant, because the DeepTendonReflex constraint model only allows this one value for the key.code, but this is not always the case. Line 5 is a check of the source data instances bodyLocation qualifier to verify this deep tendon reflex was taken on the patellar tendon. At this point, there is enough information to verify a transformation can take place, and a new PatellarDeepTendonReflex instance is created. In Line 7, the key code in the destination data instance is set with the precoordinated code for patellar deep tendon reflex. In Line 8, the data section is copied from the source to the destination

[7]The term *src* will be used to indicate the source data instance and the term *dst* will be used to indicate the destination instance. The destination instance is the data instance that is created as a result of the transformation.

```
src = 'DeepTendonReflex';
dst = 'PatellarDeepTendonReflex';

CHECK (src.key.code == 'DeepTendonReflex_KEY_ECID')
CHECK (src.qual.bodyLocation.data.cwe.code == 'PatellarTendon_ECID')

dst.key.code = 'PatellarDeepTendonReflex_KEY_ECID';
dst.data.pq = src.data.pq;
```

Figure 4.19. Explicit transformation for Case 1.

instance. This example does not include all qualifiers, and is thus not fully complete, but is meant to highlight the main issues.

This transformation is an example of an explicit transformation. No information is needed from the terminology server. This does not mean it would not be possible to use the terminology server to precoordinate the key and the qualifier value, but it is rather pointless, since the result is always the same. However, this would not be the case in a situation where the result of the composition is nondeterminant, and we will come to examples of this later.

Even though we have stated that there is no reason to use a semi-expiclit transformation for this use case, we will still show in Figure 4.20 what a semi-explicit transformation could look like.

This semi-explicit transformation has basically the same mechanics as the the explicit example, except that a call is made to the terminology server to compose two codes into a single precoordinated code. This is seen in line 7, where a variable *terminologyReturnValue* is declared, and then a value is assigned to this variable. The value assigned is a result of the function COMPOSE, which is a call to the terminology server. The COMPOSE function is given the 2 codes to precoordinate, but it is not enought to give the 2 codes; you must also state in what domain context you want the terminology server to consider these codes. This is indicated in the form of DOMAIN.CODE. PhysicalExam_DOMAIN_ECID.DeepTendonReflex_KEY_ECID indicates DeepTendonReflex_KEY_ECID in the do-

```
src = 'DeepTendonReflex';
dst = 'PatellarDeepTendonReflex';

CHECK (src.key.code == 'DeepTendonReflex_KEY_ECID')
CHECK (src.qual.bodyLocation.data.cwe.code == 'PatellarTendon_ECID')

var terminologyReturnValue =
        COMPOSE( PhysicalExam_DOMAIN_ECID.DeepTendonReflex_KEY_ECID +
                BodyLocation_DOMAIN_ECID.PatellarTendon_ECID );

dst.key.code = terminologyReturnValue;
dst.data.pq = src.data.pq;
```

Figure 4.20. Semi-explicit transformation for Case 1.

main of PhysicalExam_DOMAIN_ECID, and BodyLocation_DOMAIN_ECID.PatellarTendon_ECID indicates PatellarTendon_ECID in the domain of BodyLocation_DOMAIN_ECID. The result of the COMPOSE function is assigned to the variable terminologyReturnValue which is then assigned to dst.key.code in line 11. Again, this is rather pointless in this example, because the result of the COMPOSE function is always the same.

4.8.2 Case 2: Precoordinated Key to Determinate Key + Qual

In use case 2, we will use the same models as previously shown in Figure 4.18, but now the transformation will be in the reverse direction from PatellarDeepTendonReflex to DeepTendonReflex.

In the explicit transformation shown in Figure 4.21, there is a *src* of PatellarDeepTendonReflex and a *dst* of DeepTendonReflex. The transformation involves the decomposition of the code for patellar deep tendon reflex into the *key* for deep tendon reflex and the qualifier *bodyLocation* with a value of patellar tendon.

In lines 1 and 2, the source and destination models are defined. Line 4 is a check whether the data instance contains the key code for patellar deep tendon reflex. Line 4 in this case is redundant, because the PatellarDeepTendonReflex constraint model only allows this one value for the key.code, but this is not always the case. In lines 6 and 7, the decomposition takes place, breaking the key.code into the new key of deep tendon reflex and the bodyLocation qualifier value of patellar tendon. In line 8, the data section is copied from the source data instance to the destination instance.

```
src = 'PatellarDeepTendonReflex';
dst = 'DeepTendonReflex';

CHECK (src.key.code == 'PatellarDeepTendonReflex_KEY_ECID');

dst.key.code = 'DeepTendonReflex_KEY_ECID';
dst.qual.bodyLocation.data.cwe.code = 'PatellarTendon_ECID';
dst.data.pq = src.data.pq;
```

Figure 4.21. Explicit transformation for Case 2.

Again, the semi-explicit transformation shown in Figure 4.22 has basically the same mechanics as the the explicit version, except that two calls are made to the terminology server to decompose the precoordinated code into its parts. This is seen in lines 6 and 8, where the result of the calls are assigned to the two variables, *keyCodeValue* and *bodyLocValue*. In each of the calls to the DECOMPOSE function, the precoordinated code is given as the first argument, and this is followed by the second argument which is the domain of the decomposed part. Again, these calls to the terminology server are pointless, because the result of each call to the DECOMPOSE function will always return the same value.

4.8.3 Case 3: Data + Mod to Determinate Precoord Data

For use case 3, we have a new set of models shown in Figure 4.23. The two models are CoughAssert and NoCoughAssert. CoughAssert is a model that asserts the presence of a cough, but it has a modifier to negate this, and assert the absence of cough. The NoCoughAssert Model asserts only that no cough is present and does not have the ability to state that a cough is present.

A transformation in the case of CoughAssert to NoCoughAssert needs to compose the data value of cough and the modifier negation into the precoordinated no cough. The transformation in the other direction just needs to decompose no cough into its parts. In the

```
src = 'PatellarDeepTendonReflex';
dst = 'DeepTendonReflex';

CHECK (src.key.code == 'PatellarDeepTendonReflex_KEY_ECID');

var keyCodeValue = DECOMPOSE( PatellarDeepTendonReflex_KEY_ECID,
                                PhysicalExam_DOMAIN_ECID );
var bodyLocValue = DECOMPOSE( PatellarDeepTendonReflex_KEY_ECID,
                                BodyLocation_DOMAIN_ECID );

dst.key.code = keyCodeValue;
dst.qual.bodyLocation.data.cwe.code = bodyLocValue;
dst.data.pq = src.data.pq;
```

Figure 4.22. Semi-explicit transformation for Case 2.

```
CoughAssert
    key       Assertion_KEY_ECID
    data      Cough_ECID
    negation  Negation_ECID

NoCoughAssert
    key   Assertion_KEY_ECID
    data  NoCough_ECID
```

Figure 4.23. Models for Use Case 3 and 4.

end, it is doing the same thing as our last example, which is composing and decomposing codes with a result that is determinate. Thus, the explicit and semi-explicit transformation issues for the transformation between these models are identical to the above examples involving DeepTendonReflex and PatellarDeepTendonReflex.

The explict transformation of CoughAssert to NoCoughAssert seen in Figure 4.24 is basically the same transformation as we have seen previously.

Again, since the result of the COMPOSE function is known, the semi-explicit transformation makes no sense in this case, but is still shown in Figure 4.25.

4.8.4 Case 4: Precoordinated Data to Determinate Data + Mod

As previously mentioned, for use case 4, the reverse transformation just needs to break NoCoughAssert into the known parts for CoughAssert.

```
src = 'CoughAssert';
dst = 'NoCoughAssert';

CHECK (src.key.code == 'Assertion_KEY_ECID');
CHECK (src.data.cwe.code = 'Cough_ECID';
CHECK (src.mod.negation.data.cwe.code = 'Negation_ECID';

dst.key.code = 'Assertion_KEY_ECID';
dst.data.cwe.code = 'NoCough_ECID';
```

Figure 4.24. Explicit transformation for Case 3.

```
src = 'CoughAssert';
dst = 'NoCoughAssert';

CHECK (src.key.code == 'Assertion_KEY_ECID');
CHECK (src.data.cwe.code = 'Cough_ECID';
CHECK (src.mod.negation.data.cwe.code = 'Negation_ECID';

var terminologyReturnValue = COMPOSE( PhysicalFinding_DOMAIN_ECID.COUGH_ECID +
                                      Negation_DOMAIN_ECID.Negation_ECID );

dst.key.code = "Assertion_KEY_ECID";
dst.data.cwe.code = terminologyReturnValue;
```

Figure 4.25. Semi-explicit transformation for Case 3.

The explict transformation of NoCoughAssert to CoughAssert is basically the same transformation as we have seen in previous use cases and is shown in Figure 4.26.

Again, since the results of the DECOMPOSE function are known, the semi-explicit transformation makes no sense in this case, but is shown in Figure 4.27.

4.8.5 Case 5: Key + Data to Determinate Precoordinated Data

In use case 5, we have a new set of models shown in Figure 4.28. The two models are HctEval and HctGreaterThan35Assert. HctEval is a model that details the exact hematocrit percentage in the patient, or has the ability to describe greater than and less than any

```
src = 'NoCoughAssert';
dst = 'CoughAssert';

CHECK (src.key.code == 'Assertion_KEY_ECID');
CHECK (src.data.cwe.code = 'NoCough_ECID');

dst.key.code = 'Assertion_KEY_ECID';
dst.data.cwe.code = 'Cough_ECID';
dst.mod.negation.data.cwe.code = 'Negation_ECID';
```

Figure 4.26. Explicit transformation for Case 4.

```
src = 'NoCoughAssert';
dst = 'CoughAssert';

CHECK (src.key.code == 'Assertion_KEY_ECID');
CHECK (src.data.cwe.code = 'NoCough_ECID');

var dataCodeValue = DECOMPOSE( NoCough_ECID, PhysicalFinding_DOMAIN_ECID );
var negationValue = DECOMPOSE( NoCough_ECID, Negation_DOMAIN_ECID );

dst.key.code = 'Assertion_KEY_ECID';
dst.data.cwe.code = dataCodeValue;
dst.mod.negation.data.cwe.code = negationValue;
```

Figure 4.27. Semi-explicit transformation for Case 4.

```
HctEval
    key   Hct_KEY_ECID
    data  > 35

HctGreaterThan35Assert
    key   Assertion_KEY_ECID
    data  HCTGreaterThan35
```

Figure 4.28. Models for Case 5 and 6.

hematocrit percentage. The HctGreaterThan35Assert model can only make the assertion that the hematocrit is greater than 35.

Only a tiny fraction of instances will meet the criteria to be transformed. To be useful, this transformation would need to be called when the instance is known to contain a HCT greater than 35 value.

The explicit transformation for use case 5 is shown in Figure 4.29. There is no reason to create a semi-explicit transformation. Although it could be feasible to create a transformation that calls other transformations based on the HCT value, this is beyond the scope of this initial investigation.

```
src = 'HctEval';
dst = 'HctGreaterThan35Assert';

CHECK (src.key.code == 'Hct_ECID')
CHECK (src.data.pq.value == '35' && src.data.pq.operator == 'GreaterThan_ECID'
        && src.data.pq.unit.code = 'Percent_ECID')

dst.key.code = 'Assertion_KEY_ECID';
dst.data.cwe.code = 'HCTGreaterThan35';
```

Figure 4.29. Explicit transformation for Case 5.

4.8.6 Case 6: Precoordinated Data to Determinate Key + Data

In use case 6, we are again using HctEval and HctGreaterThan35Assert. HctEval is a model that details the exact hematocrit percentage in the patient, or has the ability to describe greater than and less than any hematocrit percentage. The HctGreaterThan35Assert model can only make the assertion that the hematocrit is greater than 35.

This transformation actually makes sense, because all instances will be able to be transformed. In the previous example, only a tiny fraction of instances would have been transformable.

The explicit transformation for use case 6 is shown in Figure 4.30. There is no reason to create a semi-explicit transformation. This is strictly determinate.

```
src = 'HctGreaterThan35Assert';
dst = 'HctEval';

CHECK (src.key.code == 'Assertion_KEY_ECID');
CHECK (src.data.cwe.code == 'HCTGreaterThan35');

dst.key.code = 'HCT_KEY_ECID';
dst.data.pq.value = '35';
dst.data.pq.operator = 'GreaterThan_ECID';
dst.data.pq.unit.code = 'Percent_ECID';
```

Figure 4.30. Explicit transformation for Case 6.

4.8.7 Case 7: Key + Data to Determinate Precoordinated Key

There is not an example where this is determinate, so no transformation is provided for use case 7. The inverse is determinate, and in the next section is a transformation example.

4.8.8 Case 8: Precoordinated Key to Determinate Key + Data

In use case 8, we are using the models BlondHairColor and HairColor shown in Figure 4.31. We are currently not modeling with this style, as we would use BlondHairColorAssertion, but the point is still made. The explicit transformation is shown in Figure 4.32. There is no reason to create a semi-explicit transformation as this is strictly determinate.

4.8.9 Case 9: Key + Qual to Nondeterminate Precoordinated Key

Use case 9 uses two forms of constraint models for deep tendon reflex shown in Figure 4.33. Both of these models convey similar information, except the DeepTendonReflex model has a qualifier called *bodyLocation* which gives it the ability to describe the deep tendon reflex of different body locations such as the ankle, triceps, or patellar tendon. The model called xxxDeepTendonReflex, which is actually meant to represent a set of models with the "xxx" meant to represent the tendon in question, is limited by its precoordinated *key* to only describe the deep tendon reflex of one particular tendon.

I believe that even though this is nondeterminate, the nondeterminate set is always small and thus a transformation can be created. I cannot think of a clinically useful large destination set which would prove this untrue.

```
BlondHairColor
    key   BlondHairColor_KEY_ECID
    data  Present_ECID

HairColor
    key   HairColor_KEY_ECID
    data  Blond_ECID
```

Figure 4.31. Models for Case 8.

```
src = 'BlondHairColor';
dst = 'HairColor';

CHECK (src.key.code == 'BlondHairColor_KEY_ECID');
CHECK (src.data.cwe.code == 'Present_ECID');

dst.key.code = 'HairColor_KEY_ECID';
dst.data.cwe.code = 'Blond_ECID';
```

Figure 4.32. Explicit transformation for Case 8.

```
    DeepTendonReflex
        key  DeepTendonReflex_KEY_ECID
        qual bodyLocation
        data co

    BodyLocation
        key  BodyLocation_KEY_ECID
        qual laterality
        data cwe

    xxxDeepTendonReflex
        key  xxxDeepTendonRelex_KEY_ECID
        qual laterality
        data co
```

Figure 4.33. Models for Case 9.

The explict transformation of DeepTendonReflex to a xxxDeepTendon Reflex is shown in Figure 4.34. This example has a *src* of DeepTendonReflex and a nondeterminate *dst*. The transformation involves combining the *key* and the *qual bodyLocation* into the preco-ordinated *key* value for the unknown deep tendon reflex.

In this transformation, the *dst* is not known, so is set to "null" in line 2. Line 4 contains a declaration of a variable, which will be used to store the code for bodyLocation. Line 6 is a check whether the data instance contains the key code for deep tendon reflex. Line 6 in this case is redundant, because the DeepTendonReflex constraint model only allows this one value for the key.code, but this is not always the case. Line 7 is an assignment of

```
src = 'DeepTendonReflex';
dst = null;

var SRC_bodyLocation;

CHECK (src.key.code == 'DeepTendonReflex_KEY_ECID');
$SRC_bodyLocation = src.qual.bodyLocation.data.cwe.code;

IF($SRC_bodyLocation == 'PatellarTendon_ECID')
{
   dst = 'PatellarDeepTendonReflex';
   dst.key.code = 'PatellarDeepTendonReflex_KEY_ECID';
}
ELSEIF($SRC_bodyLocation == 'TricepsTendon_ECID')
{
    dst = 'TricepsDeepTendonReflex';
    dst.key.code = 'TricepsDeepTendonReflex_KEY_ECID';
}
// etc

dst.data.pq = src.data.pq;
```

Figure 4.34. Explicit transformation for Case 9.

the source data instances bodyLocation qualifier to the variable created in line 4. At line 9 begins the series of checks for the possible bodyLocations for deep tendon reflex. When a match is found, the destination model is set, and then the precoordinated key.code is is set.

This transformation is an example of an explicit transformation. No information is needed from the terminology server. Since there is a short list of possibilities for the destination model, this is a workable solution. I do not think that would be the case if there were hundreds or even thousands of destination possibilities.

Although we have stated that there is no reason to use a semi-expiclit transformation for this example, it is shown in Figure 4.35. This semi-explicit transformation has basically the same mechanics as the the explicit example, except that a call is made to the terminology server to compose two codes into a single precoordinated code. This is seen in line 9, where a variable *terminologyReturnValue* is declared, and then a value is assigned to this variable. The value assigned is a result of the function COMPOSE, which is a call to the

```
src = 'DeepTendonReflex';
dst = null;

var SRC_bodyLocation;

CHECK (src.key.code == 'DeepTendonReflex_KEY_ECID');
$SRC_bodyLocation = src.qual.bodyLocation.data.cwe.code;

var terminologyReturnValue =
          COMPOSE( 'PhysicalExam_DOMAIN_ECID.DeepTendonReflex_KEY_ECID' +
            concat('BodyLocation_DOMAIN_ECID.' + $SRC_bodyLocation));

//  Now to get the correct destination model we run into problems...
//  impossible using basic information from terminology server
```

Figure 4.35. Semi-explicit transformation for Case 9.

terminology server. The COMPOSE function is given the 2 codes to precoordinate, but it is not enough to give the 2 codes; you must also state in what domain context you want the terminology server to consider these codes. This is indicated in the form of DOMAIN.CODE. PhysicalExam_DOMAIN_ECID.DeepTendonReflex_KEY_ECID indicates DeepTendonReflex_KEY_ECID in the domain of PhysicalExam_DOMAIN_ECID, and BodyLocation_DOMAIN_ECID. + $SRC_bodyLocation indicates the code contained in the variable in the domain of BodyLocation_DOMAIN_ECID. The result of the COMPOSE function is assigned to the variable terminologyReturnValue which would then be assigned to dst.key.code. But then we run into problems figuring out what the destination model should be based on the source key.code. The only way to figure out such information in a dynamic fashion would be to use our own server with defining our own functions. Since this is a requirement, we could also use this server to handle the composition and decomposition functions of the terminology server.

Moreover, both the COMPOSE function shown in Figure 4.35 and the need to identify the destination model could be handled with another function with 2 arguments, a compose function name as argument 1 and the source body location as argument 2, which would

then return a single value. Figure 4.36 shows an example of such a function that would serve this purpose.

4.8.10 Case 10: Precoordinated Key to Nondeterminate Key + Qual

There does not seem to be an actual use case for this scenario. This is because the source key is precoordinated and the decomposition is always determinate.

4.8.11 Case 11: Data + Mod to Nondeterminate Precoordinated Data

There is a new set of models for case 11 shown in Figure 4.37. The two models are BodyLocation and BodyLocPrecoord. Both models describe a body location, but BodyLocation describes this in a postcoordinated model, and BodyLocPrecoord describes this in a precoordinated model.

The explicit transformation for use case 11 is not feasible. The number of possibilities such as right arm, left arm, right leg, etc., makes IF-IFELSE statements impractical, and this would be better served by an external service. Thus, a semi-explicit transformation is more practical in this case, because of the huge number of possible combinations of composition. Even this example is somewhat contrived, because body location can be more complex than just a body part and laterality. The semi-explicit transformation is shown in Figure 4.38.

4.8.12 Case 12: Precoordinated Data to Nondeterminate Data + Mod

There does not seem to be an actual use case for this scenario. This is because the source key is precoordinated and the decomposition is always determinate.

4.8.13 Case 13: Key + Data to Nondeterminate Precoordinated Data

For use case 13, we have a new set of models shown in Figure 4.39. The explict transformation of SkinColor to SkinColorXXX is shown in Figure 4.40. This example

```
var termionlogyReturnValue =
     FUNCTION ( DeepTendonReflexKeyBodyLocCompose, $SRC_bodyLocation );
```

Figure 4.36. Function to replace COMPOSE function and identify destination model.

```
BodyLocation
    key         BodyLocation_KEY_ECID
    data        Arm_ECID
    laterality  Right_ECID

BodyLocPrecoord
    key   BodyLocationPrecoord_KEY_ECID
    data  RightArm_ECID
```

Figure 4.37. Models for case 11.

```
src = 'BodyLocation';
dst = 'BodyLocPrecoord';

CHECK (src.key.code == 'BodyLocation_KEY_ECID');

var laterality = src.qual.laterality.data.cwe.code;
var bodyLocationCode = src.data.cwe.code;

var compVal = FUNCTION( 'BodyLocationLateralityCompose', $laterality, $bodyLocationCode);

dst.key.code = 'BodyLocationPrecoord_KEY_ECID';
dst.data.cwe.code = $compVal;
```

Figure 4.38. Semi-explicit transformation for case 11.

has a *src* of SkinColor and a nondeterminate *dst*. The transformation involves combining the *key* and the *data* into the precoordinated *data* value for the unknown assertion model.[8]

The conclusion is that an explicit approach will work if the choices are few, but will not work if the number of choices becomes large. But a semi-explicit approach as shown in Figure 4.41 will always work regardless of the number of possibilities.

4.8.14 Case 14: Precoordinated Data to Nondeterminate Key + Data

There does not seem to be an actual use case for this scenario. This is because the source key is precoordinated and the decomposition is always determinate.

[8]This could also use an unconstrained assertion model, rather than SkinColorXXX.

```
SkinColor
    key   SkinColor_KEY_ECID
    data  Purple_ECID

SkinColorXXX
    key   Assertion_KEY_ECID
    data  SkinColorXXX_ECID
```

Figure 4.39. Models for case 13.

```
src = 'SkinColor';
dst = null;

var SRC_data;

CHECK (src.key.code == 'SkinColor_KEY_ECID');
$SRC_data = src.data.cwe.code;

IF($SRC_data == 'Purple_ECID')
{
   dst = 'SkinColorPurple';
   dst.data.code = 'SkinColorPurple_ECID';
}
ELSEIF($SRC_data == 'RED_ECID')
{
    dst = 'SkinColorRed';
    dst.data.code = 'SkinColorRed_KEY_ECID';
}
// etc

dst.key.code = Assertion_KEY_ECID;
```

Figure 4.40. Explicit transformation for case 13.

```
src = 'SkinColor';
dst = null;

var SRC_data;

CHECK (src.key.code == 'SkinColor_KEY_ECID');
$SRC_data = src.data.cwe.code;

var modelVal = FUNCTION(( 'SkinColorModel', $SRC_data);
var compVal = FUNCTION( 'SkinColorCompose', $SRC_data);

dst = $modelVal;
dst.key.code = 'Assertion_ECID';
dst.data.cwe.code = $compVal;
```

Figure 4.41. Semi-explicit transformation for case 13.

4.8.15 Case 15: Key + Data to Nondeterminate Precoordinated Key

Use case 15 uses a new set of models shown in Figure 4.42. This is not how we are currently modeling, as we would normally create a HairColor Assertion model, but it serves to illustrate the point.

The HairColor instance is being transformed into specific hair color models. Since there are various possible destinations, this is a nondeterminate destination. Again, we find that an explicit transformation as shown in Figure 4.43 will work if the list is short, and that a semi-explicit transformation as shown in Figure 4.44 will always work.

4.8.16 Case 16: Precoordinated Key to Nondeterminate Key + Data

There does not seem to be an actual use case for this scenario. This is because the source key is precoordinated and the decomposition is always determinate.

```
HairColor
    key   HairColor_KEY_ECID
    data  Blond_ECID

xxxHairColor
    key   xxxHairColor_KEY_ECID
    data  Present_ECID
```

Figure 4.42. Models for case 15.

```
src = 'HairColor';
dst = null;

var SRC_data;

CHECK (src.key.code == 'HairColor_KEY_ECID');
$SRC_data = src.data.cwe.code;

IF($SRC_data == 'Blond_ECID')
{
   dst = 'BlondHairColor';
   dst.key.code = BlondHairColor_KEY_ECID';
}
ELSEIF($SRC_data == 'Brown_ECID')
{
    dst = 'BrownHairColor';
    dst.key.code = 'BrownHairColor_KEY_ECID';
}
// etc

dst.data.cwe.code = Present_ECID;
```

Figure 4.43. Explicit transformation for case 15.

```
src = 'HairColor';
dst = null;

var SRC_data;

CHECK (src.key.code == 'HairColor_KEY_ECID');
$SRC_data = src.data.cwe.code;

var modelVal = FUNCTION(( 'HairColorModel', $SRC_data);
var compVal = FUNCTION( 'HairColorKeyCompose', $SRC_data);

dst = $modelVal;
dst.key.code = $compVal;
dst.data.cwe.code = 'Present_KEY_ECID';
```

Figure 4.44. Semi-explicit transformation for case 15.

4.9 CETL Syntax

The Clinical Element Translation Syntax (CETL) is an XML-based transformation syntax. Previously, in Figure 4.19, an explicit transformation was described for the measurement of patellar deep tendon reflex. Here, in Figure 4.45, the CETL syntax for that same explicit transformation is shown.

Based on our previous examples, it is very easy to understand the CETL syntax. It is an XML-based syntax which follows our previous transformation examples very closely. The document type for each transformation is called *cetran*. The elements for *cetran* are listed in Table 4.5. A *cetran* element has one attribute called *name* which is the unique identifier for the transformation. The first two elements of *cetran* are *src* and *dst*, which identify the constraint models for the source and destination data instances.

Next we have the *check* element that has the attribute *value*, which contains a conditional statement. If the conditional statement returns true, the transformation continues; otherwise, the transformation fails.

The next section contains the elements that perform the transformation. The *assign* element has the attribute *value* that contains an assignment statement, and is used to assign a value to a node in the destination tree. The *copy* element contains the *value* attribute, which

```
<cetran name="DeepTendonReflex-PatellarDeepTendonReflex">
    <src value="DeepTendonReflex"/>
    <dst value="PatellarDeepTendonReflex"/>

    <check value="src.key.code == 'DeepTendonReflex_KEY_ECID'"/>
    <check value="src.qual.bodyLocation.data.cwe.code == 'PatellarTendon_ECID'"/>

    <assign value="dst.key.code = 'PatellaDeepTendonRelex_KEY_ECID'"/>
    <copy value="dst.data = src.data"/>
</cetran>
```

Figure 4.45. CETL Syntax Example.

Table 4.5. The properties of *cetran*.

Property	Cardinality	Type
name	attribute	1
src	element	1
dst	element	1
check	element	0-M
assign	element	0-M
copy	element	0-M

contains a copy statement that copies a branch of the source instance into the destination instance.

4.10 XSLT and Transformations

In Section 4.8, transformation examples were covered using a psuedo-syntax, and in Section 4.9, I described the Clinical Element Translation Syntax (CETL). This research was all theoretical and no transformation engine has been built using this research.

To bridge the gap from theory to implementation, I have researched the feasibility of handling transformations using XSLT. I was able to write XSLT transformations for a subset of the examples in Section 4.8. The current interface team is already using XSLT

to handle simple transformations for incoming HL7 messages, though they are not handling comp/decomp at this point.

One feature of XSLT that I relied on was the fact that XSLT can also call external java functions. The ability to call external functions from XSLT is a requirement to handle the complexity of composition and decomposition.

4.10.1 Finding

The XSLT experiment did lead to one new insight. The biggest problem I found with doing real-world transformation, and it may be an unsolvable problem, is what to do with identifiers. For example, in a transformation that results in less CE nodes, there is a loss of identifiers, and in the vice versa transformation, there is a gain of identifiers with a transformation that results in a fuller tree. This change in identifier information causes problems if these identifiers were used to semantically link to other instances, or were used for some other purpose.

I believe the most straightforward solution is to create all new identifiers for the newly structured instance, and any semantic links that should still remain would have to be copied. What I have not researched is the feasibility of duplicating semantic links during the transformation process.

4.10.2 Requirements for XSLT Transformation

1. In order to use XSLT for transformation, there must exist a defined XML serialization of both HL7 messages and CE Instances.
2. A Comp/Decomp engine is a requirement, and it must be accessible through an external function call from within the XSLT transformation.
3. The Transformation must have access to the service to create new identifiers through an external function call.

4.11 Query Transformations

A user can either work with instance data in its original CEType form or in one of its various CEMorph forms. It is very advantageous to provide a query mechanism that treats

CEType data and CEMorph as if they both physically exist in the datastore. In order for this to work, the user should be able to query the data as a CEMorph or CEType in an identical manner. Even though the properties of the CEMorph do not physically exist in the repository, it would be ideal for the user if they could query these virtual properties as if they were the same as a CEType property. In order to accomplish this, we would need a query language that could be transformed to match the physical storage model.

While this may be a goal for the future, in the meantime, we have decided to take a simpler approach. We will require users to query data using properties in the CEType form, but they can specify they would like the returned data to be in a specific CEMorph form. Once the user receives the data instance, they can then treat the data as if it is a CEMorph.

To ease the learning curve of determining the correct query, we could design a tool which could pop up CEType properties for QUERY based on the CEMorph.

Below is a step-by-step listing of what must happen using our simplified approach, which avoids a query transformation.

1. Designer wants to use a CEMorph called PatellarDeepTendonReflex.
2. Transformation already exists in both directions.
3. Designer has to understand the properties of the CEType to which the CEMorph is bound.
4. Designer then writes query based on CEType properties.
5. Query is executed and CEType instances returned.
6. Transformation is applied, and DeepTendonReflex instances converted to PatellarDeep-TendonReflex instances.
7. If update is required, new CEMorph Instances must keep track of from what CEType instance they were derived.

4.11.1 CEQL - Future

In the future, we may develop a Clinical Element Query Language (CEML). This language would use the dot notation used in CEML to indicate CEType and CEMorph properties. A key feature of this syntax would be that it would be abstract with regard

to the CEType storage form. In other words, users could query the properties of either CEType or CEMorphs and the query engine would handle the query transformation.

For example, imagine we design a generic model for deep tendon reflexes, where you specify the body location via a qualifier. It is quite possible users would require a CEMorph model for patellar deep tendon reflex, as seen previously in Figure 4.18.

The query language CEQL would be something like the example shown in Figure 4.46 with the ability of CEMorph queries to automatically transform into CEType queries.

```
SELECT PatellarDeepTendonReflex FROM PATIENT 1234
   WHERE key.code = "PatellarDeepTendonReflex_ECID"

   automatically transforms to ...

SELECT DeepTendonReflex FROM PATIENT 1234
   WHERE key.code = "DeepTendonReflex_ECID"
   AND qual.bodyLocation.data.cwe.code = "Patella_ECID"
   AND qual.bodyLocation.qual.laterality.data.cwe = "LateralityRight_ECID"
```

Figure 4.46. Possible CEQL syntax.

CHAPTER 5

DESIDERATA FOR DETAILED CLINICAL MODELS

5.1 Abstract

Detailed clinical models that separate medical data from their storage mechanism and allow the use of precise semantics have emerged. Many groups are involved in the design of detailed clinical models, all of them addressing the specific requirements of their institutions. Here, using the biomedical informatics literature, we attempt to identify those requirements.

5.2 Introduction

Biomedical informaticists have been computerizing medical records for the past 50 years. During that time, there has emerged a consensus that due to the complexity, extent, and constantly changing nature of medical data, traditional database design, with its tables containing clinically meaningful columns, does not work [20]. In its place, detailed clinical models (DCMs), which separate medical data from their storage mechanism and allow the assertion of precise semantics for those data, have been developed. Moreover, DCMs allow the computable meaning of data to be shared within an institution and to external entities. Indeed, DCMs are essential to achieve semantic interoperability. Many groups are designing DCMs, all trying to fulfill the requirements specific to their institution [20, 4, 3, 26, 27, 28, 29, 30]. In this paper, we identify those requirements as reported in the biomedical informatics literature.

Our inspiration for this paper came from James J. Cimino's "Desiderata for Controlled Medical Vocabularies in the Twenty-First Century" [31]. Two problems that Cimino associated with controlled medical vocabularies are relevant to DCMs. One is that a system's requirements vary with its intended purpose and there are many possible intended purposes,

which he addressed by stating that the requirements must be multipurpose. The second is the difficulty of gathering the various opinions in the literature and unifying them. For the latter problem, we will follow Cimino's lead and apologize for misrepresenting or overlooking opinions in the expectation that they will be addressed in further discussions [31].

As our work progressed, it became difficult to separate the desiderata for the model definitions from the requirements for the runtime environment in which the models are used. By assuming that certain capabilities exist in the runtime environment, however, we were able to remove some requirements. That led us to improve the clarity of our descriptions by categorizing the requirements. (Note, however, that placing a desideratum in one category does not mean that a problem cannot be solved in another category.) The desiderata fell into four categories: (1) definitional capabilities of the language, which includes syntax and the ability to semantically express the structure and content of the model, (2) implementation of the authoring environment, which allows modelers to model, (3) implementation of the runtime system, which allows you to create and use instances of data which correspond to the models, and (4) governance of model creation, which includes the organization and approval process for request, review, and change of model content. Here we discuss desiderata primarily of the first category, definitional capabilities. We also touch briefly on desiderata of implementation of the authoring environment and implementation of the runtime environment.

5.3 Definitional Capabilities of the Language

5.3.1 Comprehensive Model

A basic desideratum for the language describing DCMs is that it be comprehensive and allow models to be defined for all clinical data [32, 33]. We want a system where all data models are expressed in a common language and can exist together in a common repository, and where models in one medical domain can reference models in any other medical domain. We do not want a partial solution in which one language and syntax is used to describe lab results and another to describe pharmacy orders. The effort required to develop and maintain the machinery that will use these models will be extensive, and we do not

want to build these systems redundantly. Some concept modeling languages do not meet the requirement for comprehensiveness because they are not able to model quantitative data and numbers. When considering a language for comprehensive modeling of clinical data, an essential question is what are clinical data? Should the modeling language accommodate only what is measured, such as labs and physical findings, or should it include the ability to describe such things as hospital departments and rooms? Should it include the modeling of population data and also the elements necessary to support research and clinical trials? Our requirement is that the modeling language accommodate all information found in clinical systems, not just the patient-associated data elements. Thus, this desideratum–a restatement of Cimino's assertion that the language must be multipurpose–necessitates more desiderata [31].

5.3.2 User Input Validation and Guide

A primary use of a DCM is to validate and guide user input [20, 3, 30]. It should prevent the storage, for example, of hematocrits of over 100% or body weights given in inches. This was Alan Rector's goal with Galen and Grail–to allow only sensible information [34], and it is the responsibility of the DCM to similarly constrain the semantics of the model. Thus, the models need to express each clinical characteristic in sufficient detail to guarantee that the data can be validated using the definition and result in only sensible information being sent to the storage system. A DCM design should contain the information to allow this validation either after or while the instance is formed (i.e., the value of each attribute is being set). If the model does not contain enough information to validate and guide user input, the information will need to be handled on a case-by-case basis for each model, such as in a user input screen. It would then be an onerous task to keep the thousands of models, especially those that have the ability to change, in sync with the external rules.

5.3.3 Datatypes

At the core of any language are the fundamental datatypes that represent values, such as numbers, codes, text, binary data, and physical quantities. DCMs organize, arrange, and attribute those datatypes into a meaningful structure that can accurately describe the

clinical information being modeled. Thus, a set of concrete datatypes should be formalized, and modelers should use only those [4, 3, 26, 28]. Were modelers to create datatypes for their own needs, the system would become intractable. Most datatypes used in current clinical formalisms have their own structure and are not simple single-field datatypes. For example, the HL7 Physical Quantity datatype has subelements such as numeric value and unit of measure. The desiderata and design of well-formed datatypes for clinical use is beyond the scope of this paper, but all the desiderata assume that the elements in the model ultimately resolve to a well-defined set of primitive datatypes. Thus, we mention datatypes as we discuss the DCM desiderata, but we do not define them.

5.3.4 Terminology

Terms and concepts that are allowed as values for a coded element in a model should exist in a terminology server independent of the model [4, 28, 30, 35, 36, 37]. The modeling language, however, must support a mechanism for associating the allowed values from the server with the coded element in the model. During model creation, modelers examine the real world and the existing models and terminology. Then, for each coded element, they specify the allowed codes or concepts, which is often referred to as a concept domain or a value set. A standard code can be used to represent either a single concept or a collection of concepts from the terminology server. The process of assigning allowed code sets to a coded data element in the model is often referred to as terminology binding.

The model itself does not know about concepts such as meaning, specifications, or concept relationships since those are represented externally in the terminology server. For example, if a field is to be restricted to a pharmacological drug, the modeler could constrain the field to the single standard concept that denotes the set of things that are pharmacological drugs and then rely on the relationships within the terminology server to identify whether an individual drug falls within that domain. Most common modeling languages (Unified Modeling Language, for example) require that drugs be enumerated within the model itself. Since the model is not scalable to large value sets, it becomes difficult to keep the enumerated list consistent with ongoing terminology changes. Moreover, the pharma-

cological drug domain is steadily expanding, so a dynamic binding to an external value set obviates the need to change a model's definition whenever a new drug is introduced.

In static binding, a model is bound to a concept in the terminology server that changes neither in meaning nor value set content. In dynamic binding, on the other hand, the bound concept can change in either of those. Static binding allows for greater accuracy, dynamic binding for greater flexibility. Fortunately, it is possible to treat the terminology server as dynamic, and as long as the data models contain a reference to the particular version of the terminology server at any given state, the binding can be considered functionally static.

Another important issue to consider for coded concepts is whether the models allow exceptions. The need for this arises, for example, if a coded field is constrained to be an antibiotic in the terminology server but a user wants to enter an antibiotic not in its current value set. That breaks the binding rules and is called coded with exceptions. No one can foresee every possibility when modeling and loading the terminology server, and coded with exceptions allows the user to store the needed information for later use.

The specifics of these terminology issues are too detailed for this discussion, but the Core Principles Document produced by the Vocabulary Workgroup of Health Level Seven International (HL7) has a good set of definitions and principles for all aspects of terminology binding [27].

5.3.5 Negation

There should exist a mechanism to express negation of a given vocabulary concept [4, 3, 26]. Given the code for chest pain, for example, it should be possible to state "No." Negation can exist not only as No, but also as a point on a continuous yes-to-no range of certainty. The certainty spectrum includes almost certain, probable, possible, likely, might be, may be, unlikely, or improbable. Moreover, negation can pertain to two kinds of certainties–certainty of existence (the degree of certainty that a concept is true) and certainty of occurrence (the degree of certainty that an action occurred).

If there is no way to express negation, every condition or procedure in the terminology server must be precoordinated with every shade of negation that needs to be expressed, and that leads to a combinatorial explosion of terms in the vocabulary. Moreover, precoordina-

tion can lead to complexity in querying because each precoordinated version of a concept must be accounted for in a query statement.

5.3.6 Subject

It is often necessary to store data about people other than the patient in a patient's electronic medical record [26]. It may be necessary, for example, to store an infant's blood type in the mother's record in cases of hemolytic disease of the newborn or to store organ donor information in a transplant patient's record. Thus, there should be a mechanism to specify the subject of a finding or observation, although the subject will usually be "self" (the patient). The absence of such a mechanism could lead to a combinatorial explosion of the number of models that are required. In transplantation cases, for example, models would need to be designed for every lab result model that is described for the patient, including HLA type, HLA type of donor, and HLA type of father.

5.3.7 Meaning of Absence

For most or all of a patient's data, the subject will be set to "self." Thus, the bandwidth and storage space set aside for the rare case when a donor or relative's data are needed is continuously wasted. That can be avoided if meaning is given to absent elements, and in this case, the meaning of Absence for Subject would be "self." In other words, if an instance of data does not specify the subject, the subject is inferred to be the patient. The ability to specify the meaning of absence of a particular field in a model alleviates the need to store the value of the field in the majority of cases.

5.3.8 Constraints

At the heart of any DCM design is the need for constraints of the patient data to be stored. The different forms of constraints that must be considered are value constraints, domain constraints, co-occurrence constraints, cardinality constraints, and reference constraints. In general, there should be a mechanism to constrain any data element, or any part of a datatype, to a subset of the values that are allowed in a parent model [20, 26, 28].

The simplest kind of constraint is the value constraint. An example would be the constraint of a number to values between 1 and 100 or the constraint of a unit of measure concept to a specific value, such as mm of mercury. A mechanism for constraining the range of possible values is needed for all types of data–numeric, coded, strings, dates, etc. A domain constraint limits the values to specific concepts taken from a list of concepts in an external vocabulary server [3]. An example would permit only a pharmacological drug as a concept or, more specifically, only antibiotic drugs. Co-occurrence constraints are required when one value affects the constraint of another value [30]. For example, if a parenteral medication is specified in an order, administration would necessarily be limited to parenteral routes. Cardinality constraints limit the number of repeating elements within a model [38]. An example is limiting the number of subjects per data instance to one. Common cardinality constraints include zero to one, exactly one, zero to many, and one to many.

Reference constraints are needed in the creation of new models when, for example, a 20-test chemistry panel needs to reference the 20 individual models that the panel comprises. These might include, among others, a serum sodium result and a serum chloride result. Without such constraints, it would be necessary to remodel a test result in each model in which it occurs, and that would lead to redundancy and eventually inconsistency between all the various serum sodium and serum chloride result models.

Two types of reference constraints must be considered–explicit references and semantic links. Although the lines between them can become blurred, they differ mainly in the logical model of how data will be stored. Explicit references are used to model data that will be stored at the time the entire data instance is being stored and are considered part of it. The data are typically gathered at the same time by the same person or device. An example would be the elements of a vital signs panel, which are stored together as a collection. A semantic link reference is used to model a relationship between two instances of data that may be stored at different times, and the relationship between them is asserted later. An example would be an instance of a low hematocrit lab result model. A physician reading the value would prescribe erythropoietin, thus creating an instance of an order model that is stored in the patient's record. A "resulted in" semantic link could then be created that links

these two separate instances: hematocrit lab result (low) resulted in order (erythropoietin). The distinction between an explicit reference and a semantic link, however, is not always so clear. For example, a "reason for" order could be represented as an item in the order model or as a semantic link to a diagnosis record that contains detailed information. Which method to use depends on who asserted the diagnosis and when, and who asserted the association between the diagnosis and the order, and when.

The language should provide a mechanism to describe both types of constraints. In a vital signs panel, where several explicit references are required, the language could be similar to type assignments in common programming languages. We call them constraints here because the modeling language describes a panel as a collection of observations, and a vital signs panel contains constraints because the observations are specific (usually body temperature, blood pressure, heart rate, and respiratory rate).

Constraints are needed for semantic links even though they are external to the source and target data instance for two reasons. First, the semantic links that can exist between two given types do not include all possible relationships, and constraints help to disallow nonsensical connections between them. Second, semantic link constraints allows users and computer systems to know that the link is important and should be considered for creation by users, applications, or decision support algorithms [4, 3, 26].

5.3.9 Model Construction from Existing Models

An important requirement of any DCM implementation is the ability to build new models from existing ones [39]. That can occur by either of two mechanisms–using existing models as elements of new models or using them as parents of new models.

When new models are constructed with existing models as elements (this was illustrated in the discussion of reference constraints), it is necessary to allow the nesting of collections of any depth or breadth [32]. For example, if a model is created for blood pressure and another for heart rate, the two should be allowed to be used within a new vital signs panel model. That is a simple case of breadth, but collections can be nested further, creating depth. For example, the blood pressure model may contain a systolic blood pressure model and a diastolic blood pressure model. Sometimes, when existing models are incorporated

into new models, the meaning of the existing models changes. If a blood pressure model was used within a cardiac stress test model, for example, the blood pressure was probably measured while the patient was exercising on a treadmill, but if the model was used within a vital signs panel model, the blood pressure was probably measured while the patient was at rest. Thus, problems can arise if blood pressures are generically queried from the patient's electronic medical record and are compared without context being taken into account.

When the existing model acts as a parent, the new model can exist as an extension of the parent (for example, observation may be extended to lab observation by adding elements such as specimen type) or a restriction of the parent (for example, a quantitative lab observation may be restricted to a serum glucose observation by restricting allowed values and units). Both are common [4].

5.3.10 Instance Identity

Another requirement for DCMs is the ability to identify instances of specific information, whether they involve diseases, people, or places. For example, if a patient has a myocardial infarction and then another a week later, both events will be entered into the patient record, and it is important to distinguish one event from the other [40].

Because different classes of instances may function differently, an implementation should recognize the differences and remain consistent. While some instances, like social security numbers, have their own built-in instance identifiers, instances like individual myocardial infarctions must be assigned a unique identifier. Some instances of organizations, states, locations, and physicians function like concepts in user interfaces, pick lists, and functional relationships and thus can be assigned a concept identifier and handled by a coded datatype even though, from a pure modeling point of view, they represent instances of things rather than concepts.

HL7 has created a datatype called Instance Identifier that can be used to label instances. And, to be ontologically pure, all the classes mentioned in the preceding paragraph could use the Instance Identifier datatype for their representation, but it is much more convenient in most systems to use coded concepts for organizations, states, locations, and physicians. To remain consistent in this overlap between coded concepts and instance identifiers, we

should always use Instance Identifier when the identifier has no implied semantics such as order numbers, accession numbers, or social security numbers.

5.3.11 Temporal Requirements

Each instance of data can have a specified time stamp, but the granularity of time may differ for various events [41]. A time stamp may exist as a point or an interval in time [42]. Some instances will be known to the microsecond, others will be known only to minutes, hours, days, weeks, years, or decades. Moreover, some timestamps reference conceptual epics and repeating cycles of time, such as infancy, childhood, and adulthood or spring, summer, and fall. A model must have the ability to capture notions of time at any granularity and from any perspective used in clinical medicine.

Although the granularity of time may differ between events, there should still exist the ability to relate those events temporally. The relationship could be an exact time measure, or it could involve uncertainty in time and uncertainty in order [43, 44].

5.3.12 Isosemantic Forms

Adopting a formal modeling language does not remove all modeling flexibility. The modeler still has choices about how much conceptual information can be placed in one field or attribute. Thus, elements of modeling style still exist even in formal modeling languages. For example, a coded field that is used in a model can be pre- or postcoordinated. A precoordinated value would be “right arm”; a postcoordinated value would be “arm,” and the laterality of “right” would be placed in another field. Which is correct will depend on how the model is used. In designing a blood pressure model, for example, if the right arm is the cuff location, the precoordinated form of right arm would work better for a user interface pick list. The postcoordinated form, on the other hand, would work better for data storage, with “right” and “arm” stored separately. The decomposed form in storage allows for more efficient querying of the data when, for example, retrieving blood pressures taken on the arm.

The two blood pressure models described are isosemantic, meaning that they represent the same data but use different degrees of pre- and postcoordination. During the modeling

phase, many alternative structures for the DCM are explored for any clinical concept, and no two of them meet all needs equally because users differ in their preferences. We consider it a desideratum to allow isosemantic forms of any given model within the model set. However, allowing multiple forms of a model to exist for storage will lead to complications.

Because different institutions or systems may model the same clinical data at different levels of granularity, there needs to be a way to map DCMs so that they are semantically interoperable [39]. The mapping's availability to the runtime system's transformation engine enables taking a structured instance of data and processing the data to create an instance of data with a new structure. The need for transformation occurs, for example, when an interface receives an HL7 message that is not compatible with the institution's internal data structure or when an institution's internal data structure must be exported as an HL7 message.

When designing a set of datatypes to use within the language, it is important to consider the design of the datatypes in the interfacing language. For example, if an institution communicates its clinical data to other institutions through HL7 messages, transformations are simpler when both sets of datatypes are congruent in regard to content.

5.3.13 Documentation

An important requirement is the documentation of the overall DCM as well as the internal parts it comprises. An ability for users to include understandable narrative descriptions for all parts of a model is critical to clinical experts, modeling and terminology teams, decision support staff, guideline developers, and all those who will use the models [45].

Clinical experts help design the models and make sure that they solve the real world problems they are supposed to address. This is a back-and-forth iterative process between the expert and the modeling team. It may span years and involve many experts separated in time and location. Once a model is completed, it may be used for years before a new use case arises, and at that point, an entire new team may be responsible for updating the models.

When inferring meaning, users cannot count on the parts of the model being mapped to a standard terminology. During the initial stages of modeling, parts are not yet mapped

and a source of meaning is needed for mapping to occur. Even after terminology mapping, some parts will not have been mapped to a standard term because no domain or concept actually existed. In those cases, the user of the model maps to a local domain or concept that may eventually become standard. When the fields in a model are ambiguous, mappings will be inconsistent and errors will occur [36].

Documentation is even more important with complex models because they can be nearly impossible to understand by pure examination of the individual fields within them. The difficulty arises when an internal part of a model interacts with another internal part, and thus, even if users understand the parts, they will never understand the intended interaction of the parts unless it is clearly documented.

Documentation can occur either in the model definition or as accompanying files. With the latter, it usually lags behind the current model definition and is not available during the modeling phase for which it could be critical. An ideal authoring system would tie separate definition and documentation files together and keep them in sync.

5.4 Authoring System Implementation

The authoring system plays an important role in DCM implementation requirements. It should allow modelers to efficiently design and update models and to easily send the changes to the runtime system. The system should facilitate governance so that the changes can be approved quickly [46], and it should compile and validate the syntax before the changes are submitted.

As the number of models increases, it becomes increasingly important to provide model searching ability because a newly planned model may already exist in some form or there may exist models that should be incorporated into the new one as inclusive elements [46]. Thus, the authoring system should assist modelers who are creating references to existing models. It should also help modelers link concepts to standard terminologies, seamlessly tying them in with the terminology system [47]. It should allow modelers to search for concepts and domains and help them create correct constraints.

Finally, the authoring system should allow the modeler to accurately document the model and its various subelements and how the subelements may interact. The authoring

system is not only for technical modelers, so it must provide a presentation of the models and the associated documentation in a form that is understandable to clinicians, especially during the modeling process when clinicians are directly involved.

5.5 Runtime System Implementation

The runtime system also plays a critical role in DCM implementation requirements. It must be able to accept the model changes provided by the authoring system quickly and apply them to data storage [20, 32, 39, 22]. It is not feasible to rely on a database team to create database tables on demand or to make database changes for each model change, and a runtime system implemented as a specific static model is sure to quickly fall behind users' needs.

DCM implementation should define a basis for efficient querying of complex data [20, 32, 43, 22, 48]. The design should be coordinated with runtime storage so that users can query the data in model form and not in some hidden form devised by the data storage team to handle the needs of a changing DCM model set. The runtime system should provide a mechanism to provide default values for fields within a model. If an organization prefers that all data be explicit and does not use Meaning of Absence, it would help if Subject defaulted to "self" (which would be true for the majority of instances created).

There should be a mechanism to record precisely how the user originally saw the data. If the user was presented with the textual representation "Myocardial Infarction," for example, data storage should not only store the terminology concept code "MI." The system should also have a mechanism to manage received data that does not conform to the constraints of a model [4]. The runtime system should provide a mechanism to access items within a collection as if they were stored individually. For example, a blood pressure measurement could be stored individually, as part of a vital signs panel, or as part of a nursing note. Later, when blood pressure data is retrieved, all blood pressure readings should be appear regardless of their original participation in a collection.

Finally, the runtime system should provide a mechanism to semantically link one instance of patient data to another instance [4, 3, 26]. The two data instances that participate in a semantic link–the source instance and the target instance–are linked by a named

relationship such as "resulted in." The runtime system must therefore store three pieces of information to maintain a semantic link–source instance, target instance, and relationship. Since a semantic link can be created any time after the linked instances have been created, it may also be advantageous to store attribution and uncertainty information along with the semantic link.

5.6 Discussion

DCM desiderata can be viewed from the perspective of the modeling language, the authoring environment, or the runtime environment, but it is often difficult to confine the desiderata to only one of those arenas. For example, an implementation may fail to meet a desideratum of the modeling language but meet it within the runtime system. Even while writing this paper, we sometimes had difficulty deciding in which category a desideratum belonged. For example, we placed the desideratum to record which concept representation was seen by the user in the runtime environment but in our institution, we actually handle that desideratum in the modeling language with the use of coded datatypes derived from HL7 version 3.0 datatypes. The HL7 CD datatype has an original text property that serves to record the representation of a code that was viewed by the user.

Thus far, we have mentioned style (the way of creating a model given a particular DCM language) only briefly. A DCM implementation provides one underlying toolset, but real-world problems can be addressed many ways. Thus, given any particular clinical concept, different modelers often create different models. Some common differences include naming, structure, and value. The best an organization can do to encourage uniformity is to create a set of style guidelines. Those inevitably change as the modelers encounter problems and solutions are discovered, and it is important to meet frequently and discuss problems. Style is too broad a topic to cover here (we intend to do so in a separate paper), but we will illustrate a style issue using the modeling of heart sounds. We could create a single heart sound model with a single coded value where the value is from the domain of allowable heart sounds (e.g., murmur, rub, click). Alternatively, we could create three separate models (HeartSoundMurmur, HeartSoundRub, HeartSoundClick) with values of Present or Absent.

Related to style is the concept of consistency across domains. For example, a specimen description is used in a lab collection as well as in an intake and output collection, and it is desirable to use the same attributes and values for describing volume, color, consistency, and smell for both collection types, and it is up to the modeling team to ensure that that happens if it is appropriate. This obviates unnecessary model duplication and maintenance for the lifetime of the models, and as previously stated, the authoring system should help the modeler avoid such situations.

Another point to consider is that the language and runtime environments and the modeling style should allow graceful model enhancement. To accomplish that, appropriate procedures and mechanisms for handling new versions of models should be in place so that instances of data that already exist in the database do not become invalid or incompatible with new data. When only optional elements are added to a model, it is backward compatible with all the existing data instances in the data store. On the other hand, when new required elements are added or old elements are required to be absent or data is restructured, the change is nonbackward compatible. The options are to transform the existing data so all data are consistent with the new model or to allow data from the two versions of the model to coexist in the database.

An important fundamental practice that has emerged in the various standards is the designing of DCMs as constraints on instances of an underlying reference model [20, 22]. Using that methodology, a small to medium reference model that will hold all instances of clinical data can be physically implemented in a datastore. Thus, the new DCM creation does not require a change to the physical datastore, a result that fulfills the desideratum of the runtime system being able to accept model changes quickly. Another benefit is that this allows all clinical data instances to be stored in a similar manner, and that fulfills the need for an efficient query mechanism.

Probably the most important desideratum of them all is that the system be implementable in common programming languages [49]. Characteristics of a proposed DCM formalism, such as multiple inheritance, can lead to problems in the real world. We need to fulfill as many of the desiderata as possible, but not at the cost of an over-designed, unimplementable system.

We hope that these version 1.0 DCM desiderata will aid in the objective evaluation of modeling languages and, ultimately, to modeling languages that are highly functional and interchangeable. We also hope that high quality modeling languages will, in turn, allow us to focus on the issues of modeling style that will result in a comprehensive shared public library of DCMs that provide the basis for true semantic interoperability among clinical systems.

CHAPTER 6

THE CLINICAL ELEMENT MODEL

6.1 Abstract

Intermountain Healthcare has a long history of designing and developing detailed clinical models. Here we describe our latest model, the Clinical Element Model, and evaluate whether it meets the requirements for detailed clinical models that we described in Chapter 5.

6.2 Introduction

Detailed clinical models (DCMs) define medical observations and events in a computable representation. Intermountain Healthcare has a long history of designing and developing these models, starting with PAL (PTXT Application Language)[2] and going on to the Clinical Event Model [3, 4], which used Abstract Syntax Notation One (ASN.1) as a formalism. After the Clinical Event Model, in an attempt to enhance and generalize our DCMs, we tried using XML Schema as a formalism [26], but we came to recognize three limitations. First, the syntax was overly complex and verbose (XML Schema had been designed to be broadly applicable, but we needed only a few of its features). Second, it did not allow expression of some of our needs and thus would require the use of a secondary formalism. And finally, the use of this formalism implied that data instances would always exist in XML.

Since our brief excursion with XML Schema, we redefined our underlying model and designed a syntax tailored specifically for it. Here we describe our updated DCM, the Clinical Element Model, and evaluate whether it meets the requirements for DCMs presented in Chapter 5.

6.3 Development

6.3.1 The Clinical Element Model

When speaking of "The Clinical Element Model" we are referring to the global modeling effort as a whole. In other words, clinical element modeling encompasses our approach to representing detailed clinical models and the data instances that conform to those models. The Clinical Element Model combines an Abstract Instance Model, which defines a structure to represent instances of medical data, and an Abstract Constraint Model, which defines constraints on values in the Abstract Instance Model. We describe the two models in detail below.

The Abstract Instance Model defines a recursive structure called the Clinical Element. The Clinical Element can store individual instances of collected data such as systolic blood pressure data collected on John Doe on May 13, 2007, at 2:45 P.M., and those data must conform to the constraints or rules stated in the corresponding Constraint Model for Systolic Blood Pressure.

The abstract models actually are abstract, meaning that to actually use the Abstract Instance Model and the Abstract Constraint Model, both must be implemented in an Implementation Technology Specification (ITS) using, for example, XML, Java, C, or Objective-C, and each of the models can use a different ITS. Our current ITS is Java for the Abstract Instance Model and an XML syntax we call Clinical Element Modeling Language (CEML) for the Abstract Constraint Model (Figure 6.1).

6.3.2 Clinical Element Abstract Instance Model

The Abstract Instance Model defines the logical structure that is used to represent instances of medical data. An instance of medical data is created each time patient information is added to a medical record. For example, if John Doe's blood pressure is taken 3 times, 3 instances of medical data would be added to his record, each describing the details of one of the measurements. Then, if his serum glucose concentration is measured, an instance describing that measurement would be added to his record. Thus, the patient's medical record becomes a collection of many individual instances of medical data. The

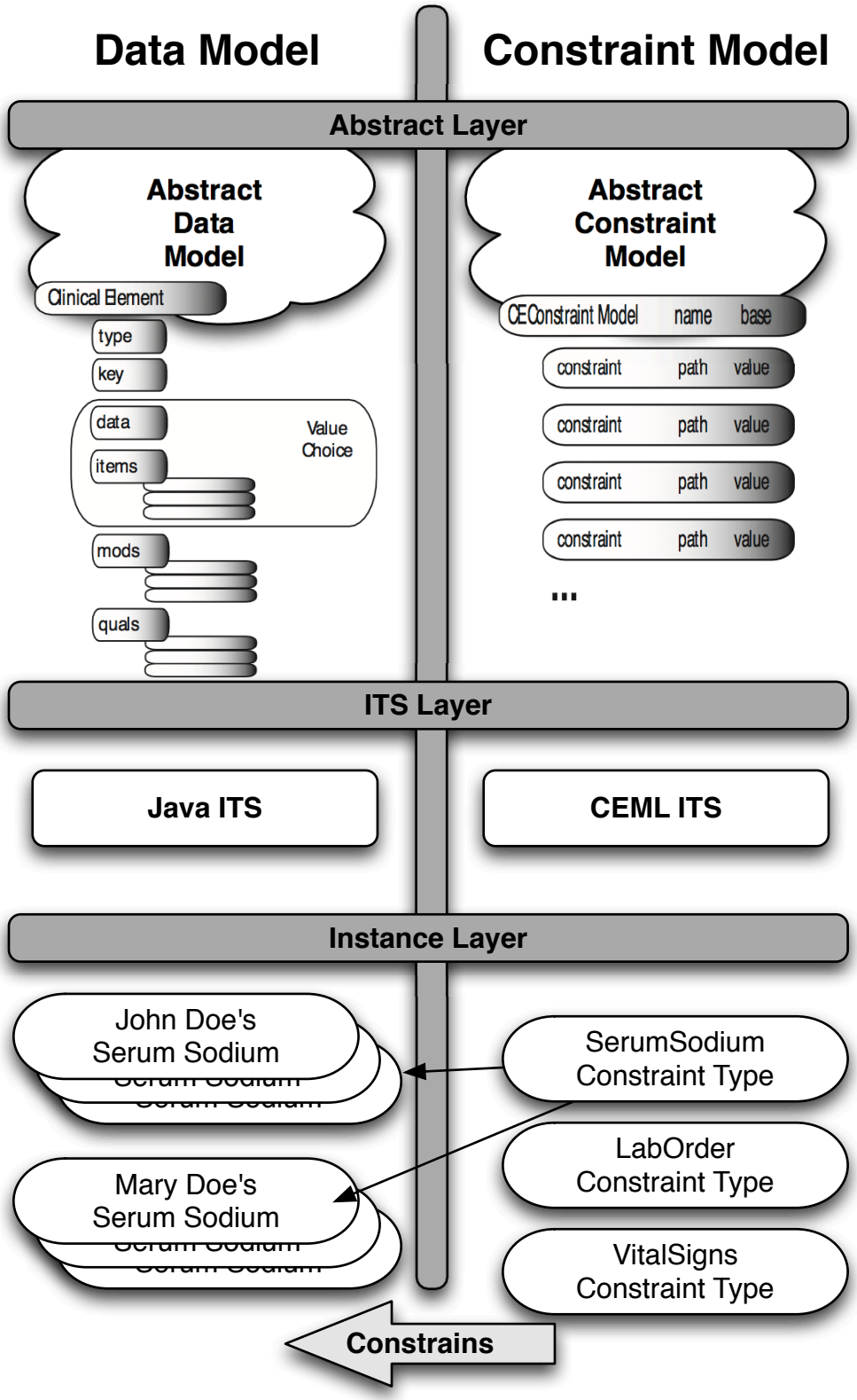


Figure 6.1. The use of the Clinical Element Model involves implementing both the Abstract Instance Model and the Abstract Constraint Model using an Implementation Technology Specification.

Abstract Instance Model describes the structure that needs to be implemented in the runtime system that is used to store all of the possible Clinical Element (CE) Instances.

The Abstract Instance Model is designed to hold any kind of instance data, and its structure remains constant regardless of the type of instance data it contains. Nonsense data can also be stored as a CE Instance, but that can be avoided if each CE Instance is linked to a specific CE Constraint Type, which is an instance of the Abstract Constraint Model (Figure 6.1).

The Abstract Instance Model is a recursive model with the core recursive element being the CE. Thus, the Abstract Instance Model is a tree with CE nodes. The basic CE (Figure 6.2) contains a type, a key, and a value choice. Type is a coded value that identifies the CE Constraint Type to which the instance will conform. Key is a coded value for the real-world concept (a concept from a standard terminology or ontology) that is important to the entity the instance is attempting to describe. Value choice is a choice between a data property or a collection of items. Data is a derivative of the HL7 version 3 datatype ANY [50] and is used to represent values such as numbers, strings, and codes. When a data choice is exercised, the value becomes a node in the CE tree (Figure 6.3). Items is a sequence of one or more CEs that give the model its recursive nature, enabling it to represent complex nested data. Figure 6.4 shows instance data that conform to the constraint type BloodPressurePanel, which allows two daughter instances in items: one that conforms to the constraint type SystolicBloodPressure and one that conforms to the constraint type DiastolicBloodPressure.

The Abstract Instance Model also defines two collections of CEs that serve to alter the meaning of the instance. These are called quals (for qualifiers) and mods (for modifiers) (Figure 6.5). A mod alters the meaning of the instance to such an extent that the newly modified instance data cannot be used without considering the effect of the mod. Examples of mods include Subject (to whom the data pertain) and Negation. A qual, on the other hand, adds information to the instance but does not necessarily change its meaning. Quals typically add information about how the data were collected (the device used, the patient position, etc.). The additional context information in quals can often be ignored in queries, and it is up to the user to decide whether quals are relevant to a particular use case.

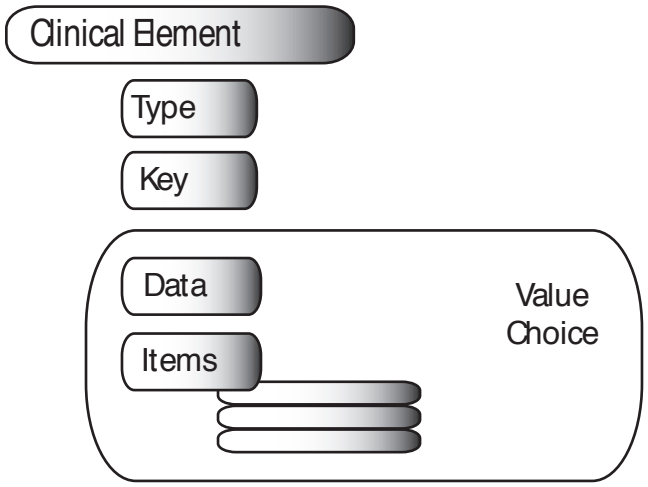


Figure 6.2. Clinical Element Abstract Instance Model.

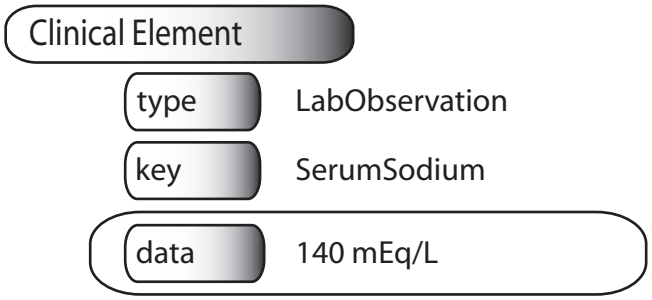


Figure 6.3. Clinical Element Instance with key and data values, and constrained by type LabObservation.

We have created a rule that states that if a qual is instantiated in a panel or collection, it is inherited by the items in the panel or collection. Figure 6.6 demonstrates how the BodyPosition qual, which is instantiated at the level of BloodPressurePanel, has a value of "Sitting," and that value is inherited by both SystolicBloodPressure and DiastolicBlood-Pressure. Thus, if SystolicBloodPressure were queried from the database, it would be returned with the BodyPosition qual, as seen in Figure 6.7, even though it may not have been stored that way. We also devised a mechanism that turns off this type of inheritance behavior.

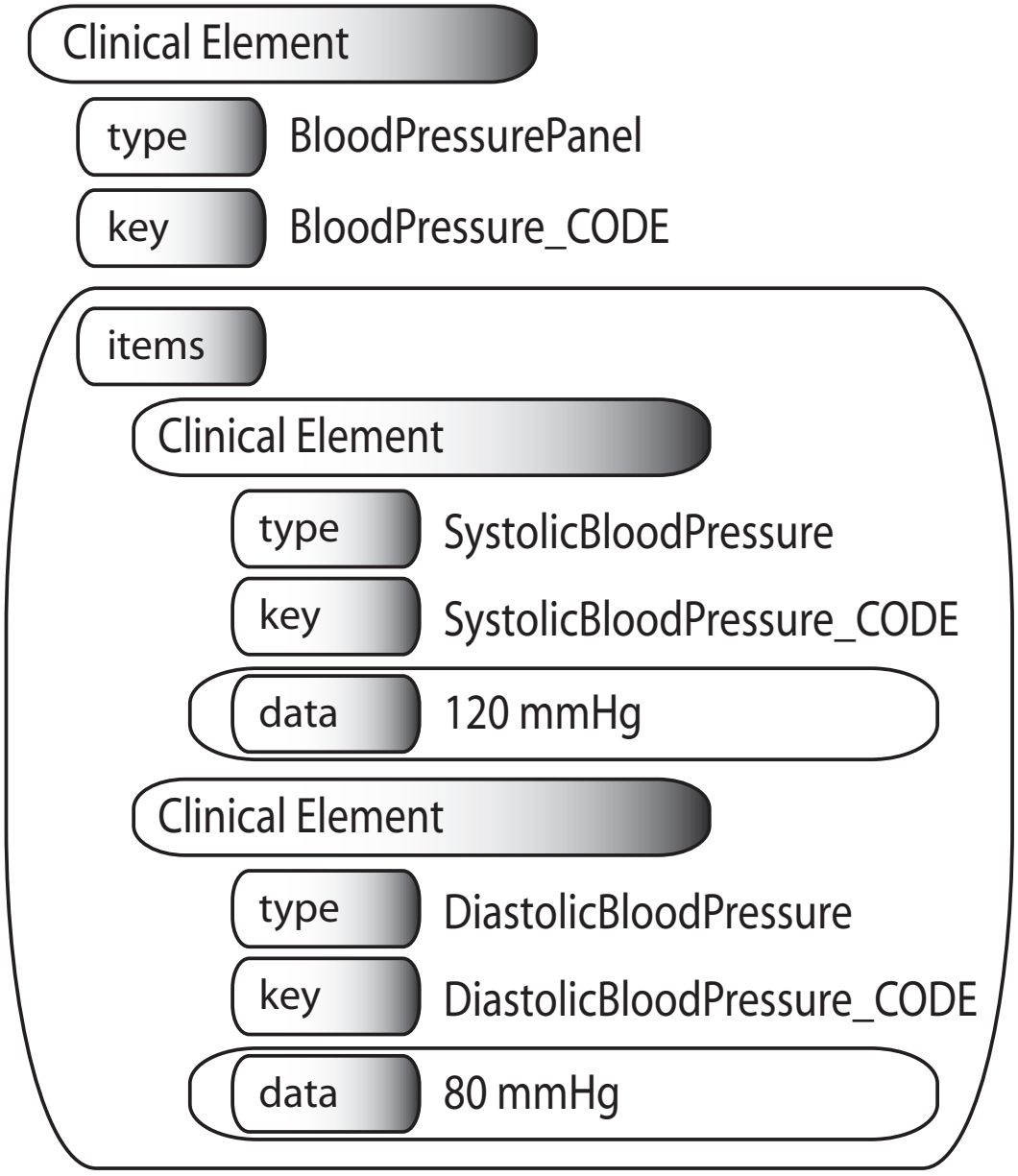


Figure 6.4. Clinical Element Instance with items that conforms to the constraint type BloodPressurePanel.

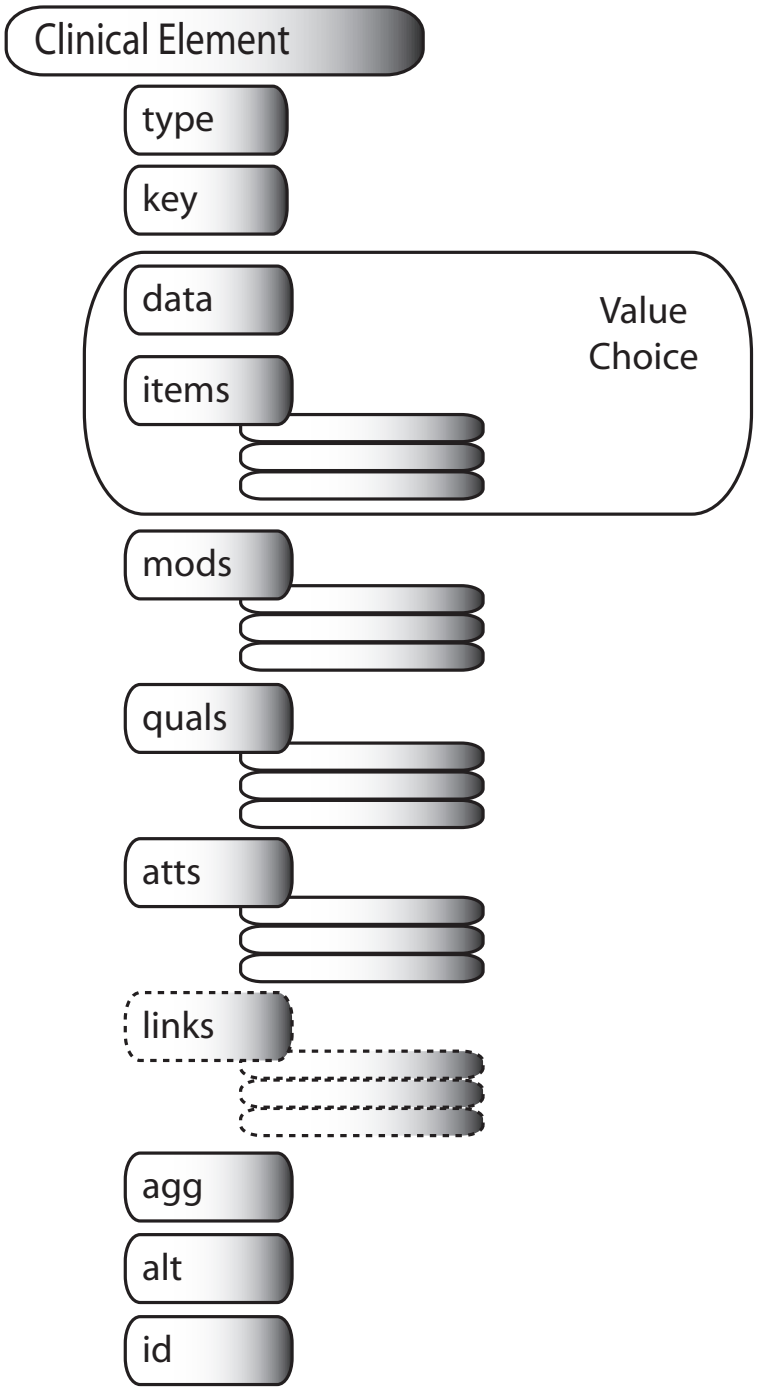


Figure 6.5. Clinical Element Abstract Instance Model in full.

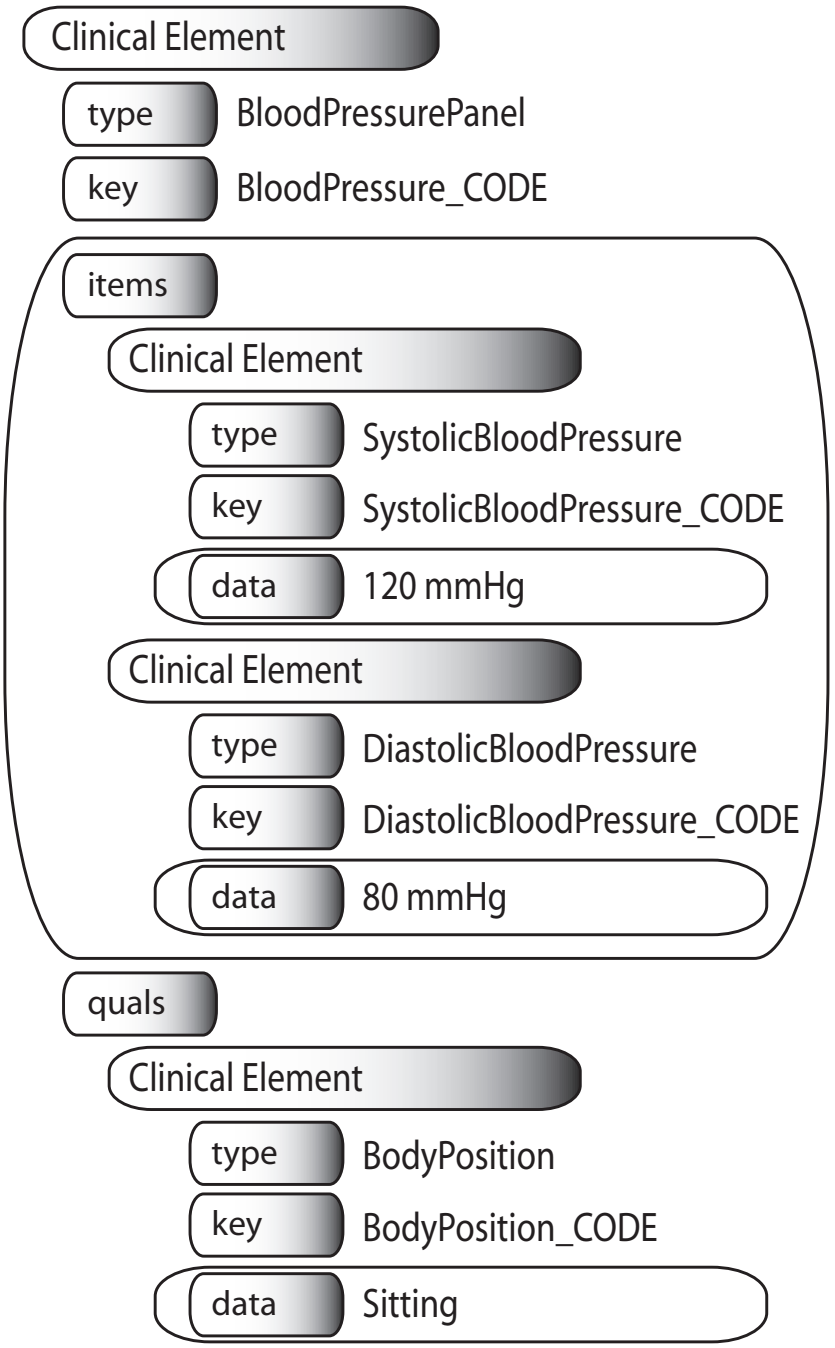


Figure 6.6. Clinical Element Instance representing a Blood Pressure Panel with the addition of a qual representing the body position "Sitting."

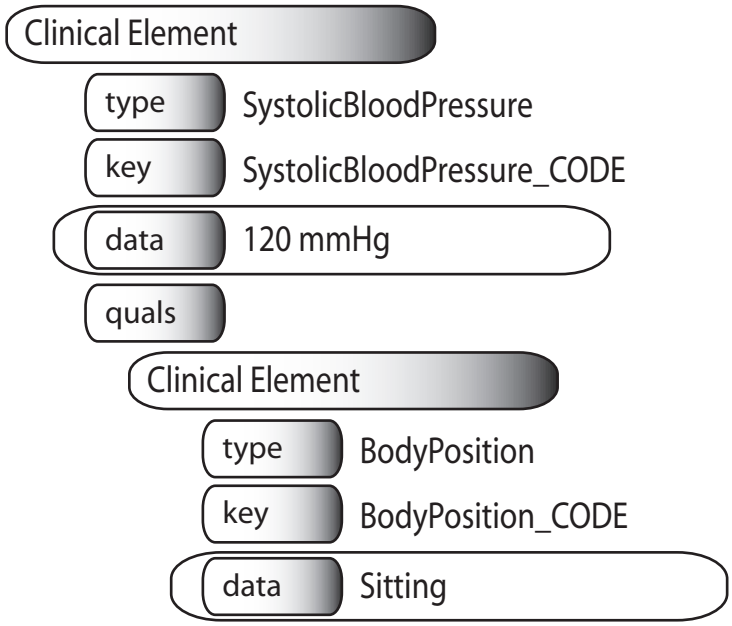
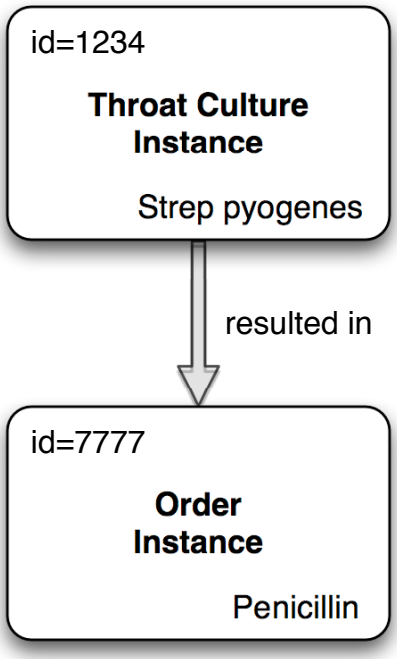


Figure 6.7. Clinical Element Instance representing a Systolic Blood Pressure with the addition of a qual representing the body position "Sitting."

The Abstract Instance Model also defines a recursive collection called Attributions, which defines the who, where, when, and why of a given action. Examples that we have modeled include Observed, Reported, Performed, and Verified.

Another defined property is id, a unique identifier for each CE Instance node. When given the value of id, it should be possible to uniquely identify a particular instance of patient data across institutions. To guarantee that no two instances use the same id, we use a Globally Unique Identifier (GUID) to populate this property.

The Abstract Instance Model describes links as a virtual property that allows access to a collection of independently stored CE Instances. The linked CE Instances and the semantic meaning of the relationship are not stored within the source CE Instance itself, but in an undefined external manner implemented by the runtime system. A restriction we place on the runtime system is that the CE id property be used to identify which CE Instance is being linked since that is defined to be unique across institutions. Additionally, since each CE node within an instance tree has an id value, semantic links can be created linking only parts of an instance. Figure 6.8 shows an example of two independently stored CE Instances representing a Throat Culture Instance and an Order Instance that are



Source	Relationship	Target
1234	resulted in	7777

Figure 6.8. A Throat Culture instance is semantically linked to an Order instance with the relationship "resulted in."

semantically linked with the relationship "resulted in," and the information maintaining the link is stored externally as a tuplet. We also allow links to connect an instance and a coded concept. For example, a lab result instance could be labeled as belonging to a clinical trial that is a concept in the terminology server.

The Abstract Instance Model defines the property alt to allow for the collection of unexpected or alternative data representations in an instance. Suppose, for example, that Systolic Blood Pressure data were being collected with the generated instances being constrained by a constraint type called SystolicBloodPressure that required data to be stored in a physical quantity datatype (PQ) consisting of a decimal number and a coded unit of measure, but a Systolic Blood Pressure value was entered with a coded value of "High." Since the Abstract Instance Model supports any of the defined data types, "High" could be placed into the data property as a coded value but the instance would fail validation. To prevent that, "High" can be placed into a coded field within alt, and an HL7 null flavor value [50] can be placed in the data property. This is illustrated in Figure 6.9, where alt is

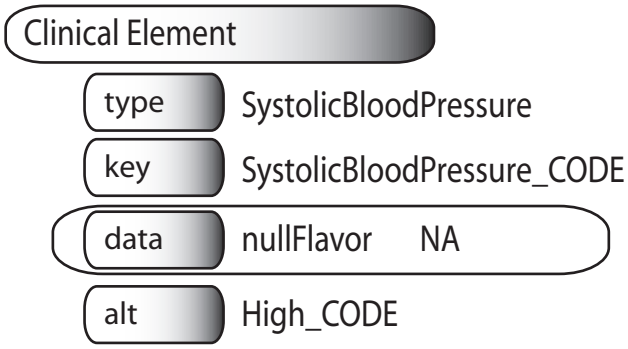


Figure 6.9. Instance data demonstrating the alt property.

defined as a choice between a coded physical quantity and a binary datatype that can store anything that is not a code or physical quantity.

The Abstract Instance Model defines the property agg as an aggregate of the information conveyed by value choice, quals, and mods. Figure 6.10 shows an example of a CE Instance node for BodyLocation with a Laterality qual, which together convey "Left Leg." The agg property in this example contains a single code for the precoordinated meaning.

6.3.3 CE Abstract Constraint Model

In the previous section, we described the CE Abstract Instance Model and how it is used to represent instances of clinical data. It should be re-emphasized that the general nature of the model allows nonsensical patient data to be instantiated in a CE Instance, and it is the role of the Abstract Constraint Model to define constraints on the general model that ensure that instances are meaningful and valid. The Abstract Constraint Model is defined abstractly, and our current implementation uses XML as an ITS, which is referred to as Clinical Element Modeling Language (CEML). We actually define two forms of CEML– one that structurally follows closely to the form of the Abstract Constraint Model and one we call Authoring CEML, which is easier for modelers to read and edit.

The Abstract Constraint Model defines the CE Constraint Type (CEType), and Figure 6.11 illustrates its properties. The CEType consists of a name, a base, and a kind property, and it contains a collection of constraints used to validate data instances. The name property is used to name the CEType and is analogous to a type name in traditional programming

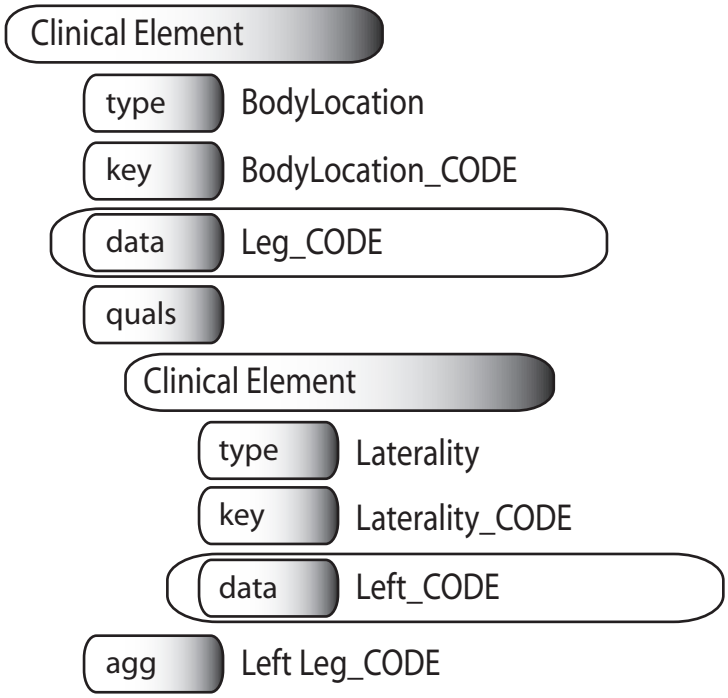


Figure 6.10. Instance data demonstrating the agg property.

languages. It is also used to link CE Instances to the corresponding CEType (Figure 6.12). The optional base property is used to inherit constraints from a parent CEType.

The kind property declares the CEType to be a "statement," "panel," "component," "mod," or "attribution." Statements and Panels represent independently storable data instances, while the other types represent only a portion of a storable instance. A Statement is a complete assertion about a particular aspect, characteristic, or condition of a patient. It can be thought of as a complete sentence such as "The patient John Doe had a Hematocrit of 38 percent on June 1, 2007." A Panel represents a common grouping of Statements that can exist independently outside the panel, and common synonyms include "battery" and "collection." A Chemistry 7 lab result is an example of a common lab panel.

Finally, there is a set of one or more constraint structures, each of which consist of a path property that identifies a specific path to a node in a CE Instance as defined by the Abstract Instance Model and a value property that identifies the value to which this specific node must be constrained. As shown in Table 6.1, the path "data.pq.unit.code" is

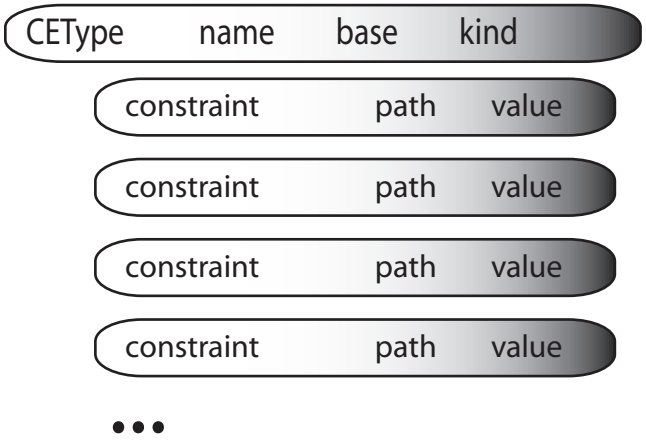


Figure 6.11. The CE Abstract Constraint Model describes the CE Constraint Type (CEType) that constrains the CE Abstract Instance Model.

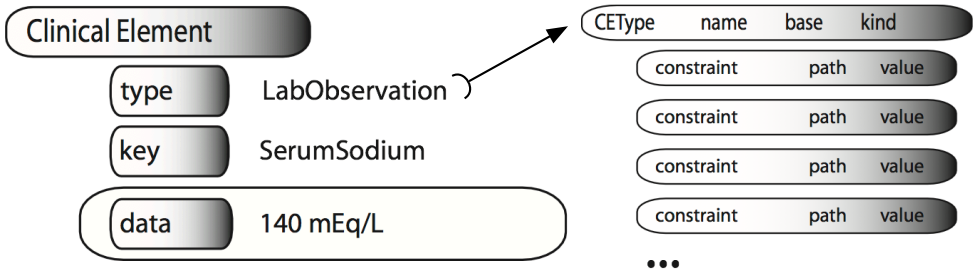


Figure 6.12. A Clinical Element Instance is constrained by constraint rules contained in the CEType named LabObservation.

noted to have a value of "mmHg." This means that any instance that conforms to CEType SystolicBloodPressure is required to have a "PQ" datatype, which in turn, must have a coded unit of measure with the value "mmHg." The specifics of each constraint path are beyond the scope of this article, but we hope these examples convey their purpose.

6.4 Results

In Chapter 5, we described the requirements for DCMs, and here we examine how well the Clinical Element Model meets those requirements, which we enumerate below. Table 6.2 lists an overview of our findings.

Table 6.1. Possible Path Constraints for the CE Instance shown in Figure 6.7

Path	Value
key.code	SystolicBloodPressure_CODE
data.pq.unit.code	mmHg_CODE
qual.bodyPosition.type	BodyPosition
qual.bodyPosition.card	0-1

6.4.1 Comprehensive Model

The language should be comprehensive so as to allow models to be defined for all clinical data. The Clinical Element Model is generic to the point that can it define not only all clinical data, but also nonclinical data, such as population data, research data, and even hospital departments and rooms. The core datatypes derived from a subset of HL7 datatypes are broad enough to describe anything we foresee, but in the event that more data types are needed, they can be added to the set without any effect on current models.

6.4.2 User Input Validation and Guide

Models should be able to guide and validate user input. In our model, every datatype in any defined model can be individually constrained to guide user input. But we fall short because we have not defined a mechanism for co-occurrence constraints. The Clinical Element Model does not exclude the possibility of co-occurrence constraints, but they have not yet been included as a part of the Abstract Constraint Model.

6.4.3 Datatypes

A concrete set of datatypes should be formalized, and modelers should be constrained to use only those datatypes. We accomplished this by defining datatypes derived from the HL7 version 3 datatypes. Their use is mandated in the Abstract Constraint Model, and it is impossible to successfully compile a CEML model with an undefined datatype.

Table 6.2. The CE Model compared to Desiderata for Detailed Clinical Models

Desideratum	Clinical Element Model
Comprehensive Model	
– Patient Clinical Data	yes
– Ancillary Non-Clinical Data	yes
– Population Data	yes, but no models created
User Input Validation and Guide	yes
Datatypes	
– Formalized	yes
– HL7 v3dt-like	yes
Terminology Separation	
– Code Separation	yes
– Value Set Separation	yes
– Handles Exceptions	yes
Negation	yes
Subject	yes
Meaning of Absence	yes
Constraints	
– Value	yes
– Domain	yes
– Cardinality	yes
– Co-occurrence	NO
– Reference	yes
– Semantic Link	yes
Builds New Models from Existing	
– Restriction	yes
– Extension	abstract models only
– Collections or Panels	yes
– Can Reuse Model in new model	yes
– Allows unlimited depth	yes
Instance Identity	
– Instance Identity Datatype	yes
– Handles IIs for all instances	NO
Temporal Requirements	
– Point in Time	yes
– Time Interval	yes
– Conceptual Times	per model basis
Isosemantic Forms	yes
Documentation	
– Overall model	yes
– Individual parts	NO
Authoring System	
– Compiles and Validates	yes
– Model Search	yes
– Tied to standard terminology	NO
Runtime System	
– Query syntax defined	in progress
– Provides default values	yes
– Records how user saw data	yes
– Accesses and sorts items within collection	yes
– Implements semantic links	yes

6.4.4 Terminology Separation

The modeling language and coded terminology should reside in an independent terminology server. CEML was created specifically to function in this manner. During the modeling process, a modeler uses identifiers to represent required concepts for coded values and value sets. Later, these identifiers are either mapped to existing terminology or new concepts are created which are then mapped. The concepts referenced in the model actually reside in a terminology server.

6.4.5 Negation

There should exist a mechanism to express Negation of a given vocabulary concept. We handle this on a model-by-model basis depending on whether or not we want to include negation for the given model. If we want negation functionality, we modify the value by including one of two predefined negation mods called CertaintyOfExistence and CertaintyOfOccurrence. CertaintyOfExistence defines the degree to which a concept describing the patient is thought to be true. CertaintyOfOccurrence is the degree of certainty that an action or procedure was believed to have occurred or been performed.

6.4.6 Subject

There should be a mechanism to indicate who is the subject of a finding or observation. The Clinical Element Model handles this by predefining a mod, in this case called Subject, which the modeler adds to any given model. Furthermore, as part of our authoring style, we have specified that any statements or panels that could have a subject must have this mod added to the model.

6.4.7 Meaning of Absence

A meaning should be specified for absent data. If a modeler has placed the Subject mod into a model and does not require a value for Subject, then it should be possible to assign a meaning for an absent Subject. We have decided that an empty Subject value should indicate a value of "self," which we assume will be the value 99% of the time. This obviates the need for users to fill out the Subject field in the majority of cases. Although

we did not cover this before, the Abstract Constraint Model defines the absence property which contains the value to use when absence is encountered. For the example given, the absence tag would be defined to have the value of "self" directly in the CEML model for Subject.

6.4.8 Constraints

Patient data need constraints. CEML is a constraint language. It provides support for value constraints, domain constraints, cardinality constraints, reference constraints, and semantic link constraints, but it falls short in the area of co-occurrence constraints. We have begun to design a syntax to address this shortcoming.

6.4.9 Build New Models from Existing Models

It should be possible to build new models from existing ones. CEML meets this important requirement, which allows modelers to recycle existing models, in three ways. First, existing models can be subclassed by restriction to create daughter models. For example, a Quantitative Lab model can be restricted to create a new Serum Sodium model. Second, abstract models can be subclassed by extension. Third, existing models can be used as building blocks to create new models. For example, a SystolicBloodPressure and a DiastolicBloodPressure model can be used as components of a larger BloodPressurePanel Model.

6.4.10 Instance Identity

It should be possible to identify instances of specific information. In the Clinical Element Model, we use a derivative of the HL7 Instance Identifier (II) datatype to uniquely identify instances of such things as diseases, people, and places, but where an instance functions as a concept (e.g., as organizations, states, locations, or physicians), we model it with a coded data type. One area we have yet to approach is identifying instances of disease. This is not a limitation of the Clinical Element Model, however, but a reflection of our failure to record any identifier to distinguish separate occurrences of the same disease process in a patient. For example, if a patient has two myocardial infarctions a few weeks

apart, it would be difficult for us to computationally identify the difference between the two events. This difficulty arises whenever events being recorded occur longer than a point in time, but shorter than forever.

6.4.11 Temporal Requirements

Each instance of data should be time-stamped. All of our models for Clinical Statements and Panels include various Attribution properties, such as Observed, that include a StartTime and an EndTime model. StartTime and EndTime both resolve to a datatype derived from the HL7 Time Stamp (TS) datatype. This allows us to specify ranges of time to various degrees of granularity, but we have no way to specify conceptual epics and repeating cycles of time-such as infancy, childhood, adulthood, and old age or spring, summer, fall, and winter.

6.4.12 Isosemantic Forms

Models should permit isosemantic forms of the same clinical information. In the process of developing individual models, we have found different structures to be appropriate for different uses, such as storage and presentation. For our storage models, we allow only one form to exist for any type or class of clinical information, but we allow modelers to build an unlimited number of transient forms of the same clinical information which can be used in data capture or rule engines or in user display. We label these models CEMorphs.

6.4.13 Documentation

Documentation should be provided for the model as a whole as well as for its internal properties. Since we lack a complete formal mechanism for documentation for all of our models, we fall short in this requirement. Currently, each model does contain a definition, but modeler commitment to even this brief description is mediocre at best. Currently, the best documentation we have is in the spreadsheets and text documents that were used in collaborative model development. Unfortunately, each modeler formats information differently and to a different extent, and there is no automatic way to parse and present the information.

6.4.14 Authoring System

The authoring system should be implemented. Our authoring system is not complete; it is a growing collection of tools and human enforced policies and procedures that we believe will eventually fulfill this requirement. Currently, we keep our uncompiled xml-based CEML constraint models in a version control system modelers can check out when they want to make extensions, corrections, or updates. We have a command line compiler that modelers use to validate the models they are building. The compiler will also output various forms of the compiled models, such as an html compiled form for model browsing and searching and a compiled xml form to be used computationally. As previously noted, we have yet to incorporate a consistent approach to model documentation.

The authoring system should aid the modeler when linking terminology to the model. This is another requirement we do not meet. At present, modelers are forced to bind all terminology by hand, and they must use one tool to browse terminology and another (either a code editor or CEDAR) to incorporate terminology into the current model. We are currently working on tools to incorporate all these pieces into one authoring system.

6.4.15 Runtime System

The runtime system should quickly accept new models and model changes from the authoring system. The Clinical Element Model is designed to accommodate this requirement. Its separation into a static Instance Model and a Constraint Model allows the runtime system to be built around the unchanging Instance Model while the modelers build models as collections of constraints. Not having to deal with constant model changes allows runtime engineers to concentrate on building efficient query algorithms and to focus on improvements and fine tuning.

The runtime system should be able to record how the data was originally seen. In the Clinical Element Model, this is handled by the Abstract Instance Model's use of the HL7:CD derived datatype, which contains the field "originalText."

In addition to the above, our runtime engineers have implemented our vision of collections and semantic links, where individual items of a collection can be queried independently, and independently stored items can be linked via a semantic connection.

6.5 Discussion

The Clinical Element Model represents Intermountain Healthcare's most current DCM strategy. Its development evolved from previous efforts and their perceived limitations, and changes were based on developers' career DCM experiences. We never thought of putting together a formal list of requirements for DCMs, and it was enlightening to examine our current model against such a list.

A change to the Clinical Element Model worth considering is making documentation a primary focus within CEML, where each and every constraint is fully documented, as is the relationship between constraints and how they affect the entire model. This documentation should span model changes over time and preserve the reasons for updates. A problem any modeling team faces is the loss of both the original modeler and a full understanding of their models. In fact, the modelers themselves eventually forget why they made certain modeling decisions.

With hindsight, we realize that we should have included co-occurrence constraints in the initial design of the Abstract Constraint Model. Solving that harder problem after the fact may lead to a need to change the syntax for our current constraints or to differing syntaxes that do not work well together.

A problem we anticipate is having a lack of proper instance identity tracking for disease processes. We do not see that as a problem now, but as more and more decision support systems begin to rely on the data, it will become evident. For example, not only will the decision support system need to address complex clinical situations, it will also need to use date, time, and clinical information to try to determine whether two statements regarding a disease process concern the same disease instance.

6.6 Conclusion

Overall, we feel the Clinical Element Model meets the desiderata for DCMs, but as we move forward, we will use the lessons discussed here to improve it. It should be noted that regardless of any given modeling implementation, modeling style choices have an enormous impact on the success or failure of a system. It would be interesting to categorize various modeling styles and examine the pros and cons of each.

In the future, we plan to evaluate how well other DCM strategies meet our modeling requirements, and we are particularly interested in evaluating how the capabilities of the Archetype Definition Language[20] compare with clinical element models.

CHAPTER 7

COMPARISON BETWEEN THE CLINICAL ELEMENT MODEL AND ARCHETYPES

7.1 Introduction

Archetypes are the Detailed Clinical Models developed and used by OpenEHR[51, 20]. Archetypes are described using the Archetype Definition Language (ADL) or its XML derivative. The approach of OpenEHR is very similar to the Clinical Element Model, except for the level of the core reference model. The Abstract Instance Model in the Clinical Element Model is defined at a lower level and is more generic, which allows CEML to be used to build any models imaginable without changes to a runtime system that implements the Abstract Instance Model. OpenEHR, on the other hand, begins with a set of higher level types such as Evaluation, Action, Instruction, Observation, etc., and an Archetype will constrain any one of these types.

In this chapter, Archetypes are critiqued[1] in comparison to the desiderata presented in Chapter 5. Table 7.1 lists the overview of these results, and the following sections will describe the specifics of each issue.

7.2 Comparison

7.2.1 Comprehensive Model

Both Clinical Elements and Archetypes are comprehensive. One difference is that the reference model for Clinical Elements is defined at a lower level, so the modeler is free to model anything from the reference model. On the other hand, the OpenEHR reference model is defined at a higher level with Evaluation, Action, Instruction, Observation, Demographic, etc. and an Archetype will constrain one of these types. If one wants to model

[1] These assertions were made in 2011, so Archetypes may have changed since that time.

Table 7.1. Archetypes compared to Desiderata for Detailed Clinical Models

Desideratum	Archetype
ComprehensiveModel	
– Patient Clinical Data	yes
– Ancillary Non-Clinical Data	yes
– Population Data	yes, but no models created
User Input Validation and Guide	yes
Datatype	
– Formalized	yes
– HL7 v3dt-like	NO
Terminology Separation	
– Code Separation	yes
– Value Set Separation	yes
– Handles Exceptions	yes
Negation	requires extra modeling
Subject	yes
Meaning of Absence	yes
Constraints	
– Value	yes
– Domain	yes
– Cardinality	yes
– Co-occurrence	NO
– Reference	yes
– Semantic Link	NO
Builds new models from existing	
– Restriction	yes
– Extension	NO
– Collections or Panels	yes
– Can Reuse Model in new model	with some limitations
– Allows unlimited depth	yes
Instance Identity	
– Instance Identity Datatype	yes
– Handles IIs for all instances	NO
Temporal Requirements	
– Point in Time	yes
– Time Interval	yes
– Conceptual Times	per model basis
Isosemantic forms	NO
Documentation	
– Overall model	yes
– Individual parts	NO
Authoring System	
– Compiles and Validates	yes
– Model Search	yes
– Tied to standard terminology	yes
Runtime System	
– Query syntax defined	yes
– Provides default values	yes
– Records how user saw data	yes
– Accesses and sorts items within collection	yes
– Implements semantic links	yes

something that is outside of these types, this could not be done without an addition to the underlying reference model. Despite this fact, Archetypes are able to model physical things such as hospital departments and rooms using the Demographic class, although this class could be better named. Archetypes could be used to model population data, but they have not modeled any at this point.

7.2.2 Guide and Validate User Input

Yes, the combination of the reference model and archetype provide the information for validation. One difference between this and the Clinical Element Model is that the validator can be built more generically for CEM, since there is only one object type and all constraints go against this object type. With Archetypes, there are multiple object types that are constrained, so the validator is more complex, and moreover, must be modified if additional base types are ever added.

7.2.3 Datatypes Formalized

Clinical Elements use datatypes which are derived directly from HL7 version 3 datatypes, and thus, there is a very close mapping between the datatypes. OpenEHR uses its own set of datatypes which are not as closely mapped to the HL7 datatypes. Any transformation involving HL7, Clinical Elements, and Archetypes will most likely require a human-derived model-to-model mapping, but the mapping may be more easily derived between HL7 and Clinical Elements because of the datatype similarity.

The HL7 website contains a document that describes datatype mapping between the OpenEHR datatypes [52]. The following is an excerpt from that document:

> When converting from an HL7 model to an archetype, most HL7 data types match an OpenEHR equivalent directly, and it is a simple matter to convert. Some HL7 data types map to specially developed archetypes for demographic type information, and some HL7 data types map to OpenEHR structures. Finally, some HL7 data types express constraints on the data that are properly expressed in the archetype itself in OpenEHR. For this reason, converting from HL7 to OpenEHR is not a simple matter of replacing data types, the entire model must be translated or transformed. This specification provides detailed equivalence notes for each HL7 data type to their matching OpenEHR constructs to help in this process. Users should be aware that though the result

of converting an HL7 Model into an archetype may work, it is unlikely to be an optimally designed archetype.

7.2.4 Terminology Separation

Archetypes have a separation of terminology with tokens used for coded terms within the archetype. This is similar to CEML, but with a few differences. Archetypes define the mapping to a defined coded terminology within the ADL file at the end of the file, where Clinical Elements define the mapping externally. One drawback is that the identifiers archetypes use are abstract and not human understandable, so when reading ADL, the user has to scroll back and forth to the ontology section at the bottom of the file to make sense of the ADL. CEML uses human-readable identifiers for concepts that infer the meaning of the concept, which makes the CEML more understandable. Of course, a user interface could be designed for ADL that does the substitutions, but the current Clinical Knowledge Manager does not seem to have such a view. The Clinical Knowledge Manager is described later in this chapter.

As in CEML, ADL can use external value sets as a coded restriction. Archetypes can handle exceptions using DV_CODED_TEXT.

7.2.5 Negation

Archetypes do not contain a negation operator for coded concepts. Instead, they use separately modeled positive exclusion statements, which are modeled from a generic Exclusion parent.

7.2.6 Subject

Archetypes can specify subject in the Entry Class.

7.2.7 Meaning of Absence

Archetypes do not allow a specification for meaning of absence, and all values must be stored explicitly.

7.2.8 Constraints

Constraints are implemented for values, cardinality, and reference to other Archetypes.

Domains or Value sets can either be constrained within an archetype by specifying an external value set, or by explicitly listing allowed values.

Co-occurrence constraints are not supported by Archetypes.

The OpenEHR system had a mechanism for semantic links, but it was disabled because of misuse by users.

7.2.9 Model Construction from Existing Models

Archetype modeling is by restriction from the reference object types (Evaluation, Action, Instruction, Observation, etc.) and extension of these types is not possible. But similar to how one can add subtypes of qualifiers, modifiers, and attributions to a Clinical Element, Archetypes have a similar mechanism with "elements" and "clusters." A modeler can build "elements" which are a portion of an archetype and use these as building blocks to build archetypes. "Clusters" are used particularly when recursiveness is required.

One difference with CEM is that an Archetype entry cannot be used within another Archetype entry, but you can combine them together with Templates.

7.2.10 Instance Identity

Archetypes have defined a type for identifiers called DV_Identifier. But like the Clinical Element Model, it is up to the modeler to use it appropriately. So in models where instances exist greater than point in time, and less than forever, such as myocardial infarction, it is up to the modeler to devise an instance id mechanism to distinguish one MI from another.

7.2.11 Temporal Requirements

Archetypes allow both a point in time or time range. Conceptual coded times can be modeled, but like the Clinical Element model, this is on a model-by-model basis.

7.2.12 Isosemantic Forms

Archetypes have no specific mechanism to assign a group of models as belonging to an isosemantic group.

7.2.13 Documentation

Archetypes have a detailed documentation section with description, purpose, use, and misuse guidelines, which is more extensive than in CEML. But this documentation is still just an overall description, and there is no way to explicitly document the nodes within an archetype other than inserting random programmatic c-style comments, which is basically the state of CEML.

7.2.14 Authoring System

The Archetype authoring system allows the modeler to compile and validate their models using the ADL Workbench Tool (AWB). Models are edited using the Archetype Editor, and the editor can link to external terminology services to aid the modeler in mapping coded concepts. OpenEHR also has developed the Clinical Knowledge Manager (CKM), which is an online authoring environment to allow collaborative design, editing, and searching of models among the various institutions using Archetypes.

7.2.15 Runtime System

The OpenEHR system defines a runtime system for use with Archetypes. The runtime system defines a query language called the Archetype Query Language (AQL). The system can provide default values where specified in an Archetype. AQL allows the querying and return of statements contained within collections as individual statements. Moreover, the individual items within a collection, such as a Serum Sodium within a Chemistry 20 Panel, all end up as entries when stored, so not only can they be queried individually, this is also the way they are stored, which would naturally improve the performance of this type of query.

The DV_CODED_TEXT datatype within archetypes can be used if the system needs the functionality of the originalText field within the HL7:CD datatype to store what textual representation of a code the user saw.

As previously stated, the OpenEHR system does implement semantic linking between stored data, but it is not being used.

7.3 Conclusion

Archetypes do meet the vast majority of desiderata for Detailed Clinical Models. One interesting observation is that Archetypes and the Clinical Element Model satisfy and fail to satisfy virtually the same desiderata. In particular, both fail to provide co-occurrence constraints and provide detailed documentation to subparts within a model. Also, both do not provide instance identifiers for all models, and conceptual times are handled on a model-by-model basis as well.

Archetypes do not handle a definition for isosemantic forms of models as Clinical Elements do, and the current datatypes used by Archetypes do not derive from HL7 datatypes. And finally, negation in Archetypes requires a parallel set of negation models, where in the Clinical Element Model, the same model can specify assertion and negation.

CHAPTER 8

CONCLUSION

The aim of this research is to develop a detailed clinical model architecture that meets the needs of Intermountain Healthcare and other healthcare organizations.

In summary, our approach to this problem was as follows:

1. An updated version of the Clinical Event model was created using XML Schema as a formalism to describe models.
2. In response to problems with XML Schema, The Clinical Element Model was designed and created using Clinical Element Modeling Language as a formalism to describe models.
3. To verify that our model met the needs of Intermountain Healthcare and others, a desiderata for Detailed Clinical Models was developed.
4. The Clinical Element Model is then critiqued using the desiderata as a guide, and suggestions for further refinements to the Clinical Element Model are described.

8.1 Lessons Learned

The process of designing the Clinical Element Model architecture began 10 years ago, and it still continues today. I would like to emphasize this time, because I never imagined such a long development duration. But the question could be asked in the case of detailed clinical models, does the development window ever close? Is DCM design a constant evolution, continually fixing problems as they arise? Unfortunately, over the years, we were very undisciplined at documenting these problems and the solutions to these problems. The desiderata (Desiderata for Detailed Clinical Models, 2011) was not created until recently, and it was during their creation that I realized we wasted a wealth of intellectual insight into DCM design by not starting the desiderata discovery as part of the design process from the very beginning.

I believe an important finding of this research is that every architectural DCM change should be documented with the problem being addressed, possible ways to solve the problem, the chosen solution, and why that solution was selected over other alternatives. We did do this instinctually of course, in meetings and verbally, but after a month or two, most or all of the information for the change was lost. I even remember a few times over the years that a change was made, a few years later the change was undone because we had forgotten the significance of the feature, and then later was redone because we finally remembered the problem that it had originally solved. I realize this type of design documentation is very time intensive, but I believe this information is as important as the model itself, as this information will be used to make the next-generation system. How many desiderata were lost for lack of record keeping?

Of the desiderata we addressed in this research, there are two areas where we failed that stand out and should have been addressed early in development of the Clinical Element Model.

First is documentation of the individual models themselves. Just as the core architecture evolved over time, so do individual detailed clinical models, such as SystolicBloodPressure, BreathSounds, or Orders. As was previously described, we only documented the overall purpose or description of the model, but not the components of each model and how those components interact with each other. The interaction and meaning of the parts of the model are crucial for users to properly understand and use these models. A formal syntax for this type of documentation was never attempted, but if we could start again, I would design this as a core part of the modeling syntax. Hopefully we can address this in future iterations of the Clinical Element Model.

The second area where we failed is in the implementation of co-occurrence constraints. I believe we avoided co-occurrence constraints because we were unfamiliar with this type of modeling paradigm at Intermountain. It is difficult to design a mechanism where certain values or groups of values change the constraints on other values. And once a mechanism and syntax are designed, it is also hard for modelers to envision all the possible interactions. If we do undertake co-occurrence constraints in the future, it will again be important to fully document all the interactions and reasons for any co-occurrence constraint, so modelers and

users can fully understand the work of others in order for models to properly evolve over time.

In summary, the most important findings of this research are as follows:

1. The problems, options, solutions, and reasons for change in the DCM architecture should be recorded over time.
2. The individual detailed clinical models and their parts should be fully documented over time.
3. Co-occurrence constraints should be implemented from the beginning

8.2 Contribution

The design of the Clinical Element Model is unique in that it is completely generic with no clinical content modeled at the level of the the Abstract Instance Model. There is a complete separation of the Abstract Instance Model and the Abstract Constraint Model, with all clinical content represented within the Abstract Constraint Model. This design is now being implemented by General Electric (GE) and Intermountain Healthcare.

Using the Clinical Element Model as a base, a transformation language for Clinical Element instance data was developed. As part of the transformation process, solutions were developed to handle the complex problems of terminology composition and decomposition. And finally, the Desiderata for Detailed Clinical Models, which lists the requirements needed in a DCM implementation, can be used by future designers to build next-generation DCM designs.

8.3 Limitations

The limitation of this research is that I was trained and worked with a single methodology at one institution. My solutions to design problems were influenced by Intermountain Healthcare's rich history in DCM design. My search for desiderata in the literature was obviously biased to some extent by everything I had learned during 10 years of a particular DCM design. Another designer of a separate DCM implementation may view something as a desiderata that I simply overlooked because I did not understand its true significance.

And finally, the field of DCM design is very young and constantly evolving, and the entire set of desiderata does not exist in the literature.

I hope this research and the lessons learned here have contributed to the evolution of detailed clinical models, and the collection of desiderata will evolve in parallel.

APPENDIX A

CLINICAL ELEMENT DATATYPES

Each detailed clinical model in the Clinical Element Model defines a value choice between a datatype or a list of child Clinical Elements. This can be thought of as the payload of the model. The following section describes the datatypes used when a Clinical Element is modeled to contain a datatype. These datatypes are based on the design of the HL7 version 3 datatypes.[1] We have chosen to use only a subset of the HL7 datatypes, and the datatypes used have been modified by the addition and deletion of various attributes we found necessary for use in a permanent datastore.[2] In addition, it should be understood that the HL7's datatype UML description is a hierarchy of types. The Clinical Element implementation of the datatypes is not a hierarchy, but a flat declaration of the types with the appropriate attributes. To understand the following datatype discussion, it is best to have some familiarity with the HL7 version 3 datatypes.

A.1 Datatypes

The Value Choice of a Clinical Element may use any of the datatypes listed in Table A.1. In addition to these datatypes, we have also defined the following datatypes, which are either used to define the previous datatypes, or are used directly by parts of the Clinical Element Model and these are shown in Table A.2.

[1] As part of HL7's RIM, a set of datatypes called the HL7 version 3 datatypes have been defined. These datatypes are used to represent actual information such as numbers, strings, and codes. A UML and textual description of the HL7 datatypes is available from HL7. HL7 has implemented the datatypes in XML schema, and is now working with Sun Microsystems to develop them in Java. HL7's UML description describes methods and properties, which are both implemented in the Java types, yet the XML Schema only implements the properties.

[2] It should be remembered that HL7 messages were not designed to be used in a permanent datastore; they were designed to be a temporary exchange format between disparate systems.

Table A.1. Datatypes for use in Clinical Elements.

Value
CWE
CO
PQ
ED
TS
ST
REAL
INT
II
RTOPQ

Table A.2. Datatypes for use in other datatypes.

Value
ANY
CS
COT
CET
PQR
EDNT
CWENT
TEL
OID
CNE

A.2 HL7 Attributes and Methods

The Hl7 version 3 datatypes that we have chosen to implement are composite data types, meaning they have substructure with internal fields. These internal fields of the datatypes define both data properties which can house pieces of data, and derived properties which return a value based on the data properties. For example, the *is-zero* property which returns a boolean true or false is derived from the *value* property by checking whether the

value is zero or not. For now, our CEM datatypes ignore the derived properties of the HL7 datatypes. We will leave the derived properties for a later implementation stage of development.

Our datatypes are more similar to the HL7 XML ITS than the abstract specification. Unfortunately, in R2 of the HL7 specification, these two specifications are not yet in sync.

APPENDIX B

ANY

ANY, shown in Figure B.1, is the base datatype in the HL7 hierarchy and all other datatypes derive from this type. ANY includes the attribute nullFlavor, shown in Table B.1, which is used to indicate the coded reason for the absence of data. Examples of these coded reasons include unknown (UNK) and not applicable (NA).

Our implementation should support nullFlavor for our datatypes, either by inheritance of nullFlavor or the direct declaration of nullFlavor in each datatype. [1]

B.1 Properties

B.1.1 nullFlavor

The property *nullflavor* is a code providing the reason a datatype has a null value.

Figure B.1. ANY

[1] In a previous implementation of the Clinical Element model, nullFlavor was not part of each datatype, but was itself its own datatype. Any Clinical Element instance could then instantiate this nullFlavor datatype as needed, rather than its own defined datatype. This was also an easy way to implement the co-occurrence constraints inherent in the use of nullFlavor.

Table B.1. ANY Properties

Property	Type	Description
nullFlavor	ST	A code describing the reason a datatype has a null value

B.2 HL7 Comparison

The ANY datatype is the base type of the HL7 version 3 datatypes and is shown in Figure B.2. The property *nullFlavor* is the only nonderived data property, and is the property we use for comparison in Figure B.3. The property *dataType*, while not exactly derived, is implied by the implementation, which means the implementation will implicitly know the datatype of the instantiated datatype. The properties *nonNull*, *isNull*, *notApplicable*, *unknown*, and *other* are all derived from the value in *nullFlavor*. The properties *equal* and *identical* are comparison methods used to evaluate the given datatype instance as compared to another. Thus, only the property *nullFlavor* is of importance to us.

```
abstract type DataValue alias ANY {
   TYPE  dataType;
   CS    nullFlavor;
   BN    nonNull;
   BN    isNull;
   BL    notApplicable;
   BL    unknown;
   BL    other;
   BL    equal(ANY x);
   protected  BN    identical(ANY x);
};
```

Figure B.2. HL7:ANY declaration.

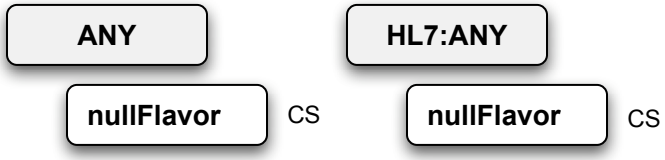


Figure B.3. ANY to HL7:ANY Comparison

B.2.1 dataType

We do not include the property *dataType*, because this can be resolved by the implementation. For example, the implementation will know if it is dealing with a **PQ** or a **CWE**.

B.2.2 nullFlavor

We include the property *nullFlavor*.

APPENDIX C

CWE

Coded With Exceptions (**CWE**), shown in Figure C.1, is used to store coded values. The properties are described in Table C.1 and constraint properties are described in Table C.2. In our Clinical Element models, we will always use **CWE** for coded values. The reason for this decision is that **CWE** permits translations of the code, and for a storage model, translations will occur very often. Also, **CWE** permits the use of a textual description in lieu of a code from the primary coding system. It is important for us to store such data, because although these data are not coded and cannot be operated on by decision support, a physician can still make timely use of these data.

Due to the performance requirements of a permanent storage-based system, we have chosen to only allow codes from the primary coding system in *code*, and alternate codes from other coding systems will be placed in translation.

For our storage form, we have decided to only allow a single **CET** within *translation*. We have removed the property *codingRationale*, and instead rely on the following rule to determine if the *code* is original. If a *translation* exists, then the *translation* is the original code, and if it does not exist, then the *code* within **CWE** is the original code.[1]

C.1 Properties

C.1.1 code

The property *code* contains a code for a concept defined in the primary coding system.

[1]For use cases, see Appendix V.

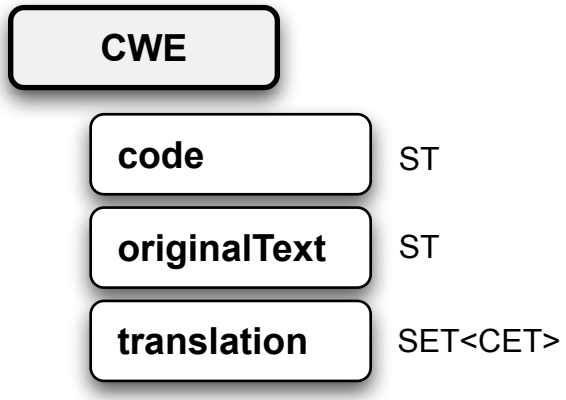


Figure C.1. Coded With Exceptions

Table C.1. CWE Properties

Property	Type	Description
code	ST.SIMPLE	A code for a concept defined in the primary code system.
originalText	ST.SIMPLE	Textual representation of the code
translation	SET<CET>	Zero to many translations of the code

C.1.2 originalText

The property *originalText* is used for textual representation of the code that was presented to the user, or the representation that came over the interface.

C.1.3 translation

The property *translation* is used to represent zero to many translations of the code from any code system. We have decided that in our storage system, we will only retain one translation.[2]

[2]Further research is required to determine which translations get discarded and which remain. Keep the source or the original translation?

Table C.2. CWE Constraint Properties

Property	Type	Description
domain	ST	A code for a domain of concepts

C.2 Constraint Properties

C.2.1 domain

The property *domain* is a code for a domain of concepts defined in the primary code system.

C.3 HL7 Comparison

The Clinical Element **CWE** datatype is much like a **HL7:CD** shown in Figure C.2 with the coding strength qualifier *CWE*, except that a code outside the specified value set/code system may not be used. It is very often the case that we might receive text in lieu of a code, and we still need to store such data, so **CWE** makes sense. Coding Strength Qualifiers are described in Table C.3. The comparison of **CWE** to **HL7:CD** is shown in Figure C.3.

For performance requirements, we have decided that only codes from the primary coding system will be allowed in *code*. The *codeSystem*, *codeSystemName*, and *codeSystemVersion* components are dropped because they are defaulted to the primary coding system. The *displayName* component is dropped because it is an optional HL7 component that supplies default text, and we saw no practical reason for designating default text that would be usable across the entire enterprise. [3]

HL7:CD *translation* is defined as a **<SET>HL7:CD**, but we created **<SET>CET** for our *translation*. By defining **CET**, we were able to remove recursive translations, as well as remove other properties that we had stripped out of **HL7:CD**.

HL7:CD defines *originalText* as an **HL7:ED** datatype. We decided that the complexity of the **HL7:ED** datatype was not worth the cost, so we defined *originalText* to be an **ST**.

[3] In a previous implementation, we had included the attribute *displayName* to be used as a best guess surface form for performance denormalization. Engineering decided they had easier ways to solve any performance problems with realtime data dictionary lookups, so *displayName* was removed.

```
type ConceptDescriptor alias CD specializes ANY {
    ST.SIMPLE         code;
    ST                displayName;
    UID               codeSystem;
    ST.NT             codeSystemName;
    ST.SIMPLE         codeSystemVersion;
    UID               valueSet;
    TS.DATETIME.FULL  valueSetVersion;
    ED.TEXT           originalText;
    SET<CS>           codingRationale;
    DSET<CD>          translation;
    CD                source;
    BL                isCompositional;
    BL                equal(ANY x);
    BL                implies(CD x);
};
```

Figure C.2. HL7:CD declaration.

If we need to store a thumbnail or sound byte in *originalText*, then one proposed solution would be to to store a textual pointer, such as a URI, to the file.

C.3.1 code

We include the property *code*.

C.3.2 displayName

We do not include *displayName* due to the fact we only allow codes from our primary coding system in **CWE**, and a displayName can be generated at any time from the vocabulary server.

C.3.3 codeSystem

We do not include *codeSystem* because in **CWE**, the *codeSystem* is defaulted to the primary coding system.

Table C.3. HL7 CE and Coding Strength Qualifiers.

Value	Description
CWE - Coded With Exceptions	This is a coding strength qualifier that is applied to a binding of a vocabulary domain with a **CD**. It signifies that a code outside the specified value set/code system may be used, or that free text may be used in lieu of a code.
CNE - Coded, No Exceptions	This is another coding strength qualifier that is applied to a binding of a vocabulary domain with a **CD**. It signifies that a code outside the specified value set/code system is not permitted, nor is free text permitted in lieu of a code.

C.3.4 codeSystemName

We do not include *codeSystemName* because in **CWE**, the *codeSystemName* is defaulted to the primary coding system.

C.3.5 codeSystemVersion

We do not include *codeSystemVersion* because in our primary coding system, we do not use versioning.[4]

C.3.6 valueSet

We do not currently include the *valueSet*, which specifies the value set that applied when this CD was created.[5]

C.3.7 valueSetVersion

We do not currently include the *valueSetVersion*, which specifies the value set version that applied when this CD was created. Even if we decided to add the *valueSet* property, I believe the version could be handled by the vocabulary server.

---

[4]Requires more research.

[5]Requires more research.

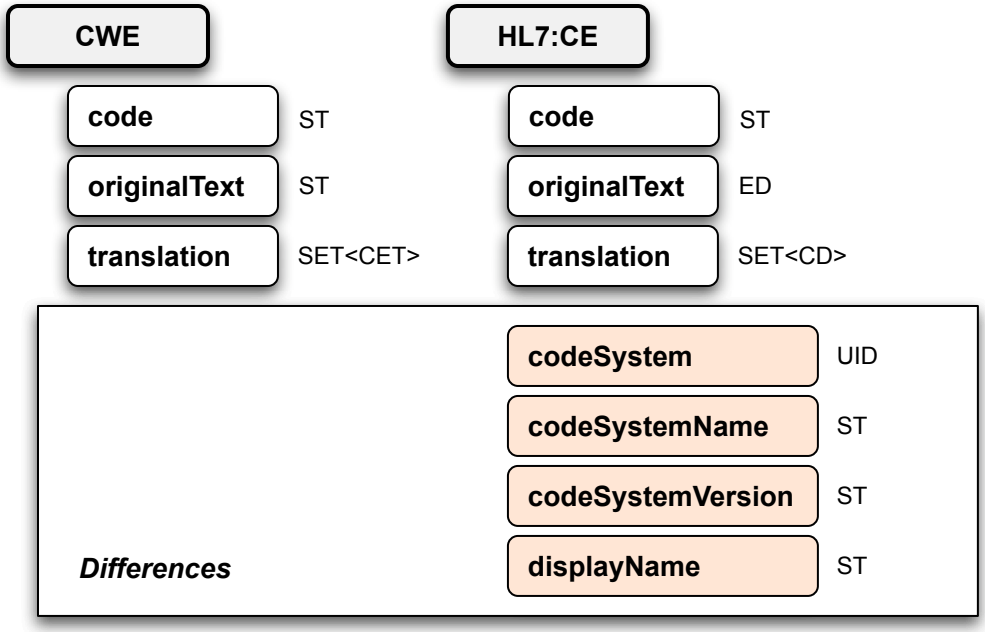


Figure C.3. CWE to HL7:CE Comparison

C.3.8 originalText

We include the property *originalText*.

C.3.9 codingRationale

We do not include the property *codingRationale*. For better or worse, we assume it is original if there is not a translation.[6]

C.3.10 translation

We included the property *translation*.

C.3.11 source

We do not include the property *source*. This property identifies the translation from which this was translated. We have not discussed this issue, but it seems this is implicit due

[6]This implementation only uses part of the functionality that *codingRationale* provides, and we have not researched the other aspects.

to the fact we are only allowing one translation in the storage form. The question remains whether or not this is important transactionally when we allow more than one translation.

C.3.12 isCompositional

We do not include the property *isCompositional*. This can be derived from the vocabulary server.

C.3.13 equal

We do not include the property *equal*. This is a comparison operation that can be handled by the vocabulary server.

C.3.14 implies

We do not include the property *implies*. This is a comparison operation that can be handled by the vocabulary server.

C.3.15 Properties inherited from HL7:ANY

Please see Appendix B.

APPENDIX D

CO

Coded Ordinal (**CO**), shown in Figure D.1, is used to store coded ordinal data such as the code for "2+," which comes from an ordered domain of codes. The properties of **CO** are described in Table D.1 and D.2.

Due to the performance requirements of a permanent storage-based system, we have chosen to only allow codes from the primary coding system in *code*, and alternate codes from other coding systems will be placed in translation.

Previously, we had the property *operator* in **CO**, but this has been removed, and there does not seem to be any documentation explaining this.

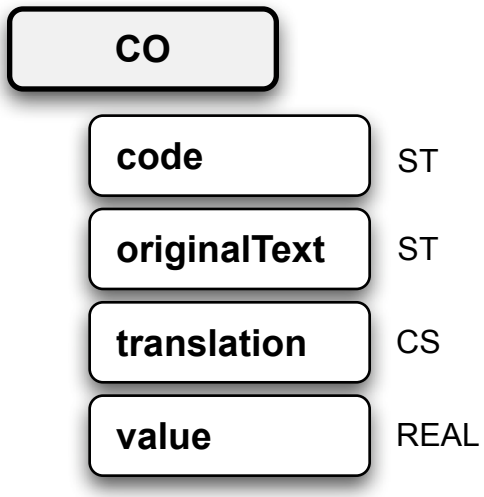


Figure D.1. Coded Ordinal

Table D.1. CO Properties

Property	Type	Description
code	ST	A code for a concept defined in the primary code system.
originalText	ST	Textual representation of the code
translation	SET<COT>	Zero to many translations of the code
value	REAL	A numeric representation of the code's meaning

Table D.2. CO Constraint Properties

Property	Type	Description
domain	ST	A code for a domain of concepts

D.1 Properties

D.1.1 code

The property *code* is a code for a concept defined in the primary code system.

D.1.2 originalText

The property *originalText* is a textual representation of the code that was presented to the user, or the representation that came over the interface.

D.1.3 translation

The property *translation* is used to represent a zero to many translations of the code from any code system.

D.1.4 value

The property *value* provides for a numeric representation of the code's meaning.

D.2 Constraint Properties

D.2.1 domain

The property *domain* represents a code for a domain of concepts defined in the primary code system.

D.3 HL7 Comparison

HL7:CO which specializes **HL7:CD** is shown in Figure D.2. It adds one additional data property which is *value*. [1] We include the other properties inherited from **HL7:CD** as we did in **CWE** and **CNE** so please see either of these chapters for the reason for inclusion or exclusion of any data properties. The comparison of **CO** to **HL7:CO** is shown in Figure D.3.

For performance requirements, we have decided that only codes from the primary coding system will be allowed in *code*. The *codeSystem*, *codeSystemName*, and *codeSystemVersion* components are dropped because they are defaulted to the primary coding system. The *displayName* component is dropped because its an optional HL7 component that supplies default text, and we saw no practical reason for designating default text that would be usable across the entire enterprise. [2]

[1] Previously, HL7 did not have *value* in **HL7:CO**, so we added it to our **CO** and called it *numericOperator*. Now that Hl7 has added this, we have changed the name of *numericOperator* to *value*.

[2] In a previous implementation, we had included the attribute *displayName* to be used as a best guess surface form for performance denormalization. Engineering decided they had easier ways to solve any performance problems with realtime data dictionary lookups, so *displayName* was removed.

```
type CodedOrdinal alias CO specializes CD {
   REAL value;
   BL  lessOrEqual(CO o);
   BL  lessThan(CO o);
   BL  greaterThan(CO o);
   BL  greaterOrEqual(CO o);
};
```

Figure D.2. HL7:CO declaration.

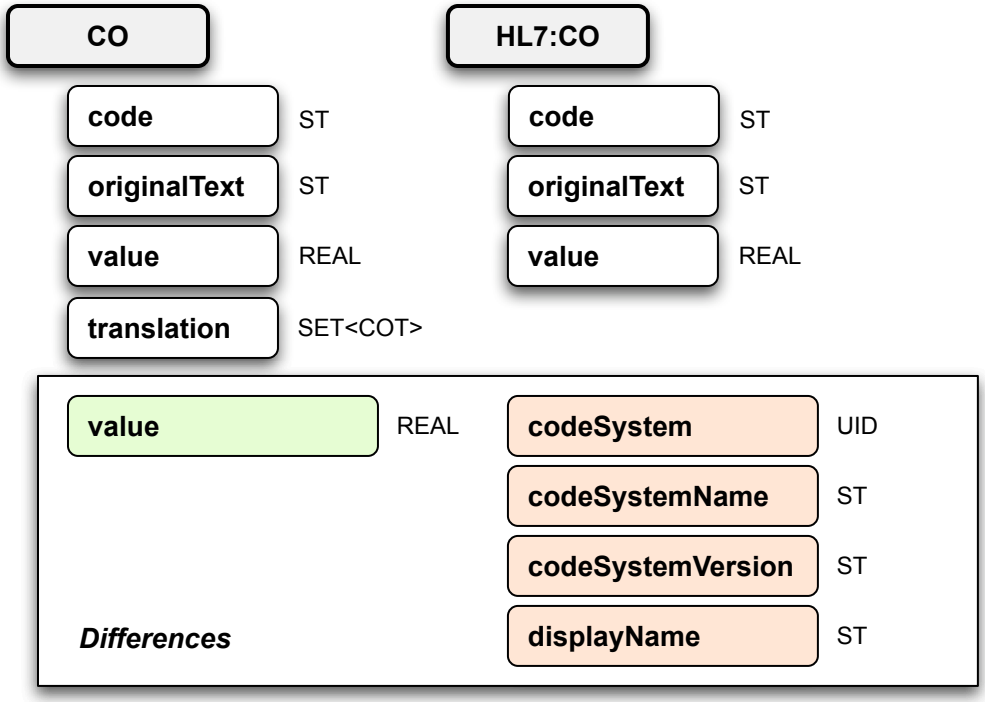


Figure D.3. CO to HL7:CO Comparison

HL7 does not allow translations for **HL7:CO** but we have relaxed this and allow translations within our **CO** and have defined this a **SET<COT>**.

HL7:CO defines *originalText* as an **HL7:ED** datatype. We decided that the complexity of the **HL7:ED** datatype was not worth the cost, so we defined *originalText* to be an **ST**. If we need to store a thumbnail or sound byte in *originalText*, then one proposed solution would be to store a textual pointer, such as a URI, to the file.

D.3.1 value

We include the property *value*.[3]

D.3.2 lessOrEqual

We do not include the property *lessOrEqual* because it is a comparison operator.

[3]We previously called this *numericScore*.

D.3.3 lessThan

We do not include the property *lessThan* because it is a comparison operator.

D.3.4 greaterThan

We do not include the property *greaterThan* because it is a comparison operator.

D.3.5 greaterOrEqual

We do not include the property *greaterOrEqual* because it is a comparison operator.

D.3.6 Properties inherited from HL7:CD

Please see Appendix C.

APPENDIX E

II

Instance Identifier (**II**), shown in Figure E.1, is used to uniquely identify an instance, thing, or object. Examples of this include social security numbers or medical record numbers. The properties of **II** are described in Table E.1.

E.1 Properties

E.1.1 root

The property *root* provides a unique identifier that guarantees the global uniqueness of the instance identifier. The root alone may be the entire instance identifier. An example of this would be an identifier that represents the concept of Social Security Number.

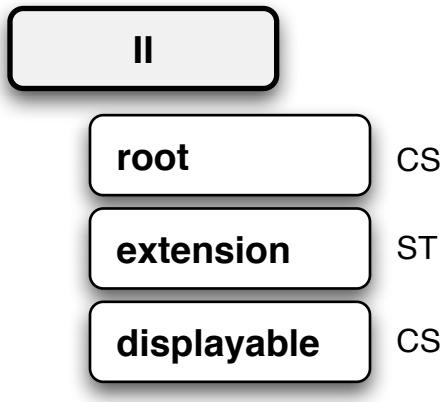


Figure E.1. Instance Identifier

Table E.1. II Properties

Property	Type	Description
root	CS	A unique identifier that guarantees the global uniqueness
extension	ST	A unique identifier within the scope of the identifier root
displayable	CS	For human display or pure machine interoperation

E.1.2 extension

The property *extension* is a character string that is unique in the namespace designated by the root. The extension property may be null, in which case the root UID is the complete unique identifier. The root and extension scheme effectively means that the concatenation of root and extension must be a globally unique identifier for the item that this II value identifies.

E.1.3 displayable

The property *displayable* specifies if the identifier is intended for human display and data entry (displayable = true) as opposed to pure machine interoperation (displayable = false).

One of the use cases of displayable is that when a DICOM image is received, the OID represents the DICOM imaging and the extension is empty. The displayable is useful here to identify that the extension is not intended to be human readable.

E.2 HL7 Comparison

The **HL7:II** datatype is shown in Figure E.2. We use the **CS** data type to represent the *root* property where HL7 uses **HL7:OID**. We do this to keep a consistent coding system in our repository. All registered OIDs will be incorporated into our vocabulary server as concepts and the OID will exist as one possible surface form for the concept.

```
type InstanceIdentifier alias II specializes ANY {
      ST.SIMPLE  extension;
      UID        root;
      ST.NT      identifierName;
      CS         scope;
      CS         reliability;
      BL         displayable;
      BL         equal(ANY x);
    literal ST.NT
};
```

Figure E.2. HL7:II declaration.

In our version of **II**, we have removed the property *assigningAuthority*, but a code for the Assigning Authority is maintained in our coding system and mapped to the root so we can always determine the Assigning Authority from the root. The comparison of **II** to **HL7:II** is shown in Figure E.3.

E.2.1 extension

We include the property *extension*.

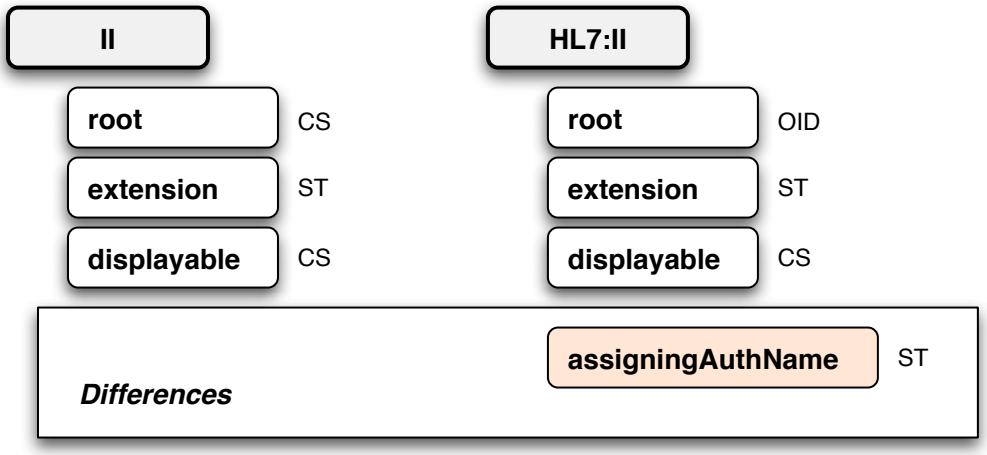


Figure E.3. II to HL7:II Comparison

E.2.2 root

We include the property *root*, but we have changed the type from an **HL7:UID** to a **CS**. We will be able to retrieve the **HL7:UID** representation from the vocabulary server.

E.2.3 identifierName

We do not include the property *identifierName* which is a human readable form of the **HL7:UID**, since this can just be retrieved as just another representation in the vocabulary server.

E.2.4 scope

We do not include the property *scope*. We feel the information provided by scope will be known by the context the **II** is used within a Clinical Element model.[1]

E.2.5 reliability

We do not include the property *reliability*. The issue of reliability of the value seems to be a global problem to any datatype, and we would move this out of the datatype and into a qualifier in the clinical element model.[2]

E.2.6 displayable

We do include the property *displayable*. But I am wondering if this is correct? The answer to this should be the same for every given root. Thus, the knowledge repository should be able to answer this question based on the root.[3]

E.2.7 equal

We do not include the property *equal*. This is just a comparison operator between two instance identifiers.

---

[1] Requires further research.

[2] Requires further research.

[3] Requires further research.

E.2.8 Properties inherited from HL7:ANY

Please see Appendix B.

APPENDIX F

INT

Integer Number (**INT**), shown in Figure F.1, is used to represent an integer number ranging from negative to positive infinity. Operator was added by Intermountain to represent codes such as "greater than" or "less than." Examples include: 0, 1, 2, 100, 3398129, -12. The properties of **INT** are described in Table F.1 and constraint properties are described in Table F.2.

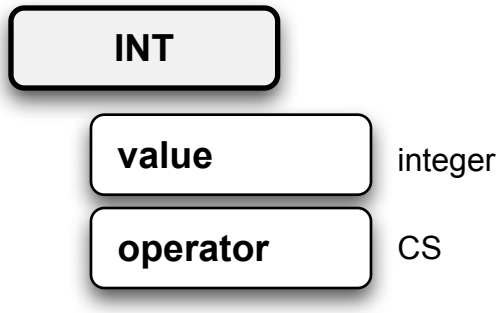


Figure F.1. Integer

Table F.1. INT Properties

Property	Type	Description
value	int	Value of the integer
operator	CS	>, <, >=, or <=

Table F.2. INT Constraint Properties

Property	Type	Description
minInclusive	INT	*value* must be greater than or equal
maxInclusive	INT	*value* must be less than or equal
minExclusive	INT	*value* must be greater than
maxExclusive	INT	*value* must be less than

F.1 Properties

F.1.1 value

The property *value* represents the value of the integer number, e.g., 5, 26. Precision must be maintained by the implementation.

F.1.2 operator

The property *operator* provides a coded value that will represent either >, <, >=, or <=.

F.2 Constraint Properties

F.2.1 minInclusive

The property *minInclusive* is an integer that *value* must be greater than or equal to.

F.2.2 maxInclusive

The property *maxInclusive* is an integer that *value* must be less than or equal to.

F.2.3 minExclusive

The property *minExclusive* is an integer that *value* must be greater than.

F.2.4 maxExclusive

The property *maxExclusive* is an integer that *value* must be less than.

F.3 HL7 Comparison

The **HL7:INT** datatype is shown in Figure F.2. Where HL7 would use an **IVL<INT>** to represent "greater than" or "less than" an integer, we use a coded operator within **INT**. For simplicity, we will not implement HL7's **IVL<INT>** in our system. The comparison of **INT** to **HL7:INT** is shown in Figure F.3.

F.3.1 value

The property *value* is not part of the HL7 abstract definition. The HL7 specification leaves it up to implementation where to store the numeric value of an integer. We have chosen to use *value* which is the same choice HL7 made in their version 3 XML ITS.

F.3.2 operator

The property *operator* is not part of the HL7 abstract definition. While HL7 would use an **IVL<INT>** to represent "greater than" or "less than" an integer, we use a coded operator within **INT**. For simplicity, we will not implement HL7's **IVL<INT>** in our system.

F.3.3 successor

We do not include the property *successor* because it can be derived. This successor is just the next integer value above, so easily can be calculated.

```
type IntegerNumber alias INT specializes QTZ {
       INT  successor;
       INT  times(INT x);
       INT  predecessor;
       INT  negated;
       BL   isNegative;
       BL   nonNegative;
       REAL  dividedBy(REAL x);
       INT  dividedBy(INT x);
       INT  remainder(INT x);
       BL   isOne;
  literal  ST.SIMPLE;
};
```

Figure F.2. HL7:INT declaration.

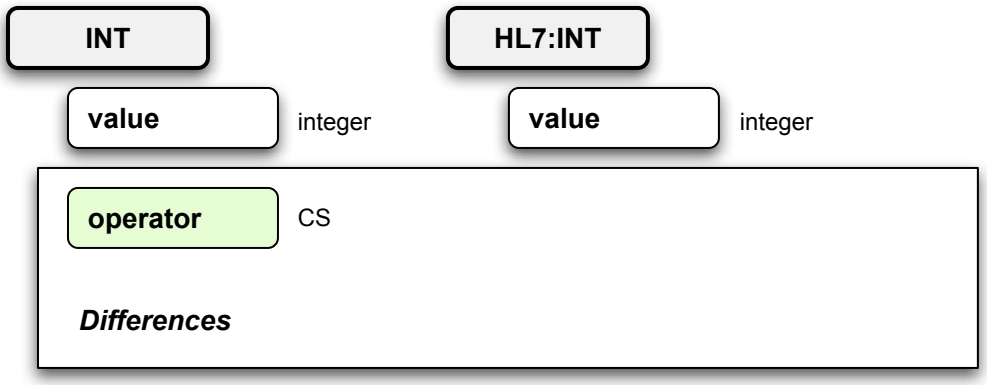


Figure F.3. INT to HL7:INT Comparison

F.3.4 times

We do not include the property *times*, which is just a calculation operator.

F.3.5 predecessor

We do not include the property *predecessor*. This predecessor is just the previous integer value below, so easily can be calculated.

F.3.6 negated

We do not include the property *negated*. The negated just returns the opposite sign of the current integer value.

F.3.7 isNegative

We do not include the property *isNegative*. This boolean result is derived from the current integer value.

F.3.8 nonNegative

We do not include the property *nonNegative*. This boolean result can just be derived by checking if the current integer value is zero or greater.

F.3.9 dividedBy

We do not include the property *dividedBy*, which is just a calculation operator.

F.3.10 remainder

We do not include the property *remainder*, which is just a calculation operator.

F.3.11 isOne

We do not include *isOne* because it can be derived by checking if the current integer value is one.

APPENDIX G

PQ

Physical Quantity (**PQ**), shown in Figure G.1, is used to store a real number value with a real physical unit to represent quantities such as 2.3 milligrams, 24 days, or 2 drops. It also allows the representation of greater than and less than some physical quantity. Moreover, **PQ** has the capability to represent equivalent translations of the Physical Quantity, with a different physical unit and an appropriately converted real number value. The value in **PQ** always represents the normalized value and unit for use in the ECIS system. If a different original unit exists, it is represented in the translation section.[1] The properties of **PQ** are described in Table G.1 and constraint properties are described in Table G.2.

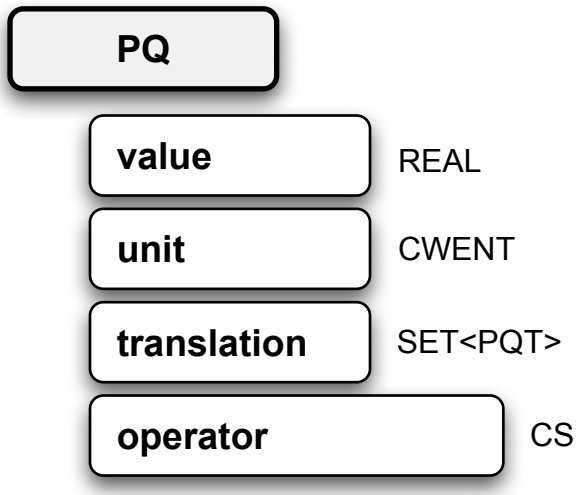


Figure G.1. Physical Quantity

[1]For use cases, see Appendix W.

Table G.1. PQ Properties

Property	Type	Description
value	REAL	The magnitude of the quantity measured in terms of the *unit*
unit	CWENT	Unit of measure, e.g., mg, seconds
operator	CS	>, <, >=, or <=
translation	SET<PQT>	One or more translations of this **PQ**

Table G.2. PQ Constraint Properties

Property	Type	Description
normal	ST	A code for the normalized unit of measure
minInclusive	REAL	*value* must be greater than or equal
maxInclusive	REAL	*value* must be less than or equal
minExclusive	REAL	*value* must be greater than
maxExclusive	REAL	*value* must be less than

G.1 Properties

G.1.1 value

The property *value* is the magnitude of the quantity measured in terms of the *unit*

G.1.2 unit

The property *unit* is a unit of measure, e.g., mg, seconds. Note that *unit* is of type **CWE**NT which has no translations. The reason for this is that the **PQ** datatype itself has translations in the form of **PQT**, so any translated *unit* would just result in another translation within a **PQT**.

G.1.3 operator

The property *operator* is a coded value that will represent either >, <, >=, or <=.

G.1.4 translation

The property *translation* represents one or more translations of this **PQ**. Translations are of type **PQT**. For storage, we only allow one **PQT** within the *translation* section of **PQ**. If this translation exists, it is always the original physical quantity, and the representation in the upper section of the **PQ** is a normalized value taken from this original physical quantity.

G.2 Constraint Properties

G.2.1 normal

A code from the primary coding system, for a preferred unit of measure, which will be the required normalized value for this **PQ**.

G.2.2 minInclusive

The property *minInclusive* represents a real number that *value* must be greater than or equal to.

G.2.3 maxInclusive

The property *maxInclusive* is a real number that *value* must be less than or equal to.

G.2.4 minExclusive

The property *minExclusive* is a real number that *value* must be greater than.

G.2.5 maxExclusive

The property *maxExclusive* is a real number that *value* must be less than.

G.3 HL7 Comparison

The **HL7:PQ** datatype is shown in Figure G.2 and the comparison of **PQ** to **HL7:PQ** is shown in Figure G.3. One major difference between **PQ** and **HL7:PQ** is that HL7 requires that *unit* be a UCUM code, but we have decided to code *unit* using the primary

```
type PhysicalQuantity alias PQ specializes QTZ {
            REAL      value;
            CS        unit;
            BL        equal(ANY x)
            BL        lessOrEqual(PQ x);
            BL        compares(PQ x);
            PQ        canonical;
            SET<CS>   codingRationale;
            DSET<PQR> translation;
            PQ        negated;
            PQ        times(REAL x);
            PQ        dividedBy(REAL x);
            PQ        times(PQ x);
            PQ        dividedBy(PQ x);
            PQ        inverted;
            PQ        power(INT x);
            BL        isOne;
  literal   ST.SIMPLE;
  demotion  REAL;
};
```

Figure G.2. HL7:PQ declaration.

coding system.[2] The UCUM system enables one to create physical quantities that are not constrained to any particular unit, but by using UCUM codes, you could never create inexact physical quantities such as 2 drops. Even though our *unit* is not represented by a UCUM code, the code used should, when applicable, be mapped to the appropriate UCUM code in the vocabulary server.

Another difference is that HL7 uses **IVL<PQ>** to represent greater than and less than. Instead, we handle this directly in **PQ** with the *operator* property. The HL7 solution allows the representation of concrete ranges as well as the opened ended range of greater than and less than, so we use **IVL<PQ>** when that is appropriate.

G.3.1 value

We include the property *value*.

[2]The primary coding system is the single coding system that has been selected for the storage of Clinical Elements.

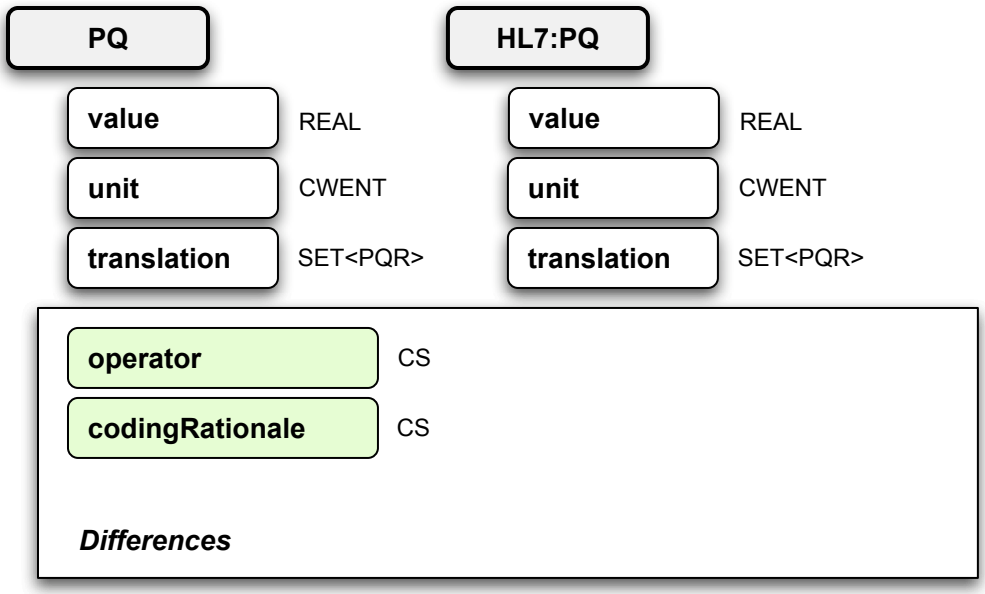


Figure G.3. PQ to HL7:PQ Comparison

G.3.2 operator

The property *operator* is not part of the HL7 abstract definition. While HL7 would use an **IVL<PQ>** to represent "greater than" or "less than" a real number, we will also use a coded value within **PQ** to represent this.

G.3.3 unit

We do include the property *unit* but we have changed the type to our **CWENT** because we want to be able to store *originalText* along with the code. In addition, HL7 uses UCUM codes and we are using codes from our primary coding system. We do not want to internally use multiple coding systems, and in addition, UCUM does not support concepts like "drops" since they are not truly quantitative.

G.3.4 equal

We do not include the property *equal* because it is just a comparison operator.

G.3.5 lessOrEqual

We do not include the property *lessOrEqual* because it is just a comparison operator.

G.3.6 compares

We do not include the property *compares* because it is just a comparison operator.

G.3.7 canonical

We do not include the property *canonical*, the reason being that we always put the canonical form in our *PQ*. Any value we receive that is not in the canonical or what we call "normal" form is normalized, and the original is stored in *translation*.

G.3.8 codingRationale

We do not include the property *codingRational*.[3]

G.3.9 translation

We do include the property *translation* but have changed the type from **HL7:DSET<PQR>** to our **SET<PQT>**.

G.3.10 negated

We do not include the property *negated* because it is derived as a negation of the current *value* in the **PQ**.

G.3.11 times

We do not include the property *times* because it is just a calculation operator.

G.3.12 dividedBy

We do not include the property *dividedBy* because it is just a calculation operator.

---

[3] Requires further research.

G.3.13 inverted

We do not include the property *inverted* because it is just a calculation operator.

G.3.14 power

We do not include the property *power* because it is just a calculation operator.

G.3.15 isOne

We do not include the property *isOne* because it is just a boolean check to see if the current *value* in the **PQ** is one or not.

APPENDIX H

REAL

Real Number (**REAL**), shown in Figure H.1, is used to store real number values such as 1.25678, 3, or 0. **REAL** contains the property *operator* to represent concepts such as "greater than" or "less than." The properties of **REAL** are described in Table H.1 and constraint properties are described in Table H.2.

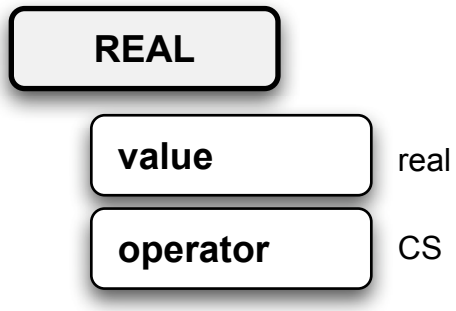


Figure H.1. Real Number

Table H.1. REAL Properties

Property	Type	Description
value	real	Value of the real number
operator	CS	>, <, >=, or <=

Table H.2. REAL Constraint Properties

Property	Type	Description
minInclusive	REAL	*value* must be greater than or equal
maxInclusive	REAL	*value* must be less than or equal
minExclusive	REAL	*value* must be greater than
maxExclusive	REAL	*value* must be less than

H.1 Properties

H.1.1 value

The property *value* represents the value of the real number, such as 1.25678, 3.00, or 0. Precision must be maintained by the implementation.

H.1.2 operator

The property *operator* provides a coded value that will represent either >, <, >=, or <=.

H.2 Constraint Properties

H.2.1 minInclusive

The property *minInclusive* is a real number that *value* must be greater than or equal to.

H.2.2 maxInclusive

The property *maxInclusive* is a real number that *value* must be less than or equal to.

H.2.3 minExclusive

The property *minExclusive* is a real number that *value* must be greater than.

H.2.4 maxExclusive

The property *maxExclusive* is a real number that *value* must be less than.

H.3 HL7 Comparison

The **HL7:REAL** datatype is shown in Figure H.2 and the comparison of **REAL** to **HL7:REAL** is shown in Figure H.3. While HL7 would use an **IVL<REAL>** to represent "greater than" or "less than" some real number, we use a coded operator within **REAL**. For simplicity, we will not implement HL7's **IVL<REAL>** in our system.

H.3.1 value

The property *value* is not part of the HL7 abstract definition. The HL7 specification leaves it up to implementation where to store the numeric value of a real number. We have chosen to use *value*, which is the same choice HL7 made in their version 3 XML ITS.

H.3.2 operator

The property *operator* is not part of the HL7 abstract definition. While HL7 would use an **IVL<REAL>** to represent "greater than" or "less than" a real number, we use a coded operator within **REAL**. For simplicity, we will not implement HL7's **IVL<REAL>** in our system.

```
type RealNumber alias REAL specializes QTZ {
             REAL  negated;
             REAL  times(REAL x);
             REAL  dividedBy(REAL x);
             REAL  inverted;
             BL    isOne;
             REAL  power(REAL x);
  literal    ST.SIMPLE;
             INT   precision;
  demotion   INT;
  promotion  REAL  (INT x);
  promotion  PQ;
};
```

Figure H.2. HL7:REAL declaration.

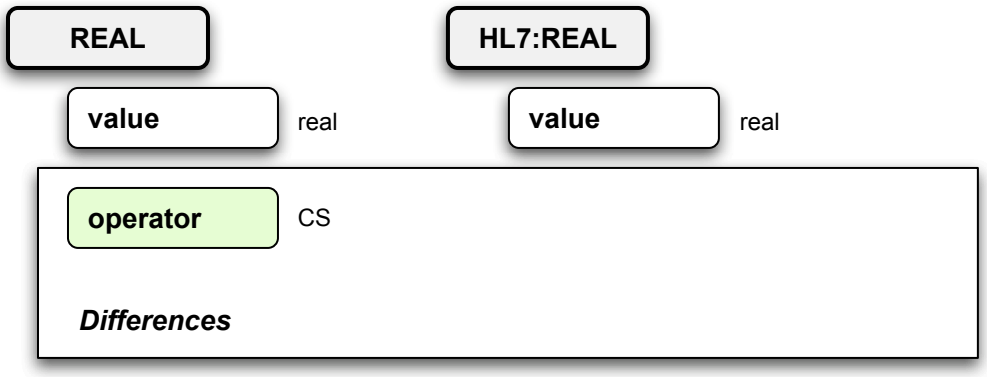


Figure H.3. REAL to HL7:REAL Comparison

H.3.3 negated

We do not include the property *negated*. The negated just returns the opposite sign of the current real number value.

H.3.4 times

We do not include the property *times*, which is just a calculation operator.

H.3.5 dividedBy

We do not include the property *dividedBy*, which is just a calculation operator.

H.3.6 inverted

We do not include the property *inverted*, which is just a calculation operator.

H.3.7 isOne

We do not include *isOne* because it can be derived by checking if the current real value is one.

H.3.8 power

We do not include the property *power*, which is just a calculation operator.

H.3.9 precision

We do not include the property *precision*, which is the number of significant digits of the decimal representation. This can be derived from our *value* property.

APPENDIX I

ST

Character String (**ST**), shown in Figure I.1, is used to store text/plain data. The properties of **ST** are described in Table I.1 and constraint properties are described in Table I.2.

I.1 Properties

I.1.1 value

The property *value* is used to hold the string of characters. When **ST** is used in some implementation, the explicit representation of *value* is not required, but only need be implied. **ST** is frequently used in the internal representation of other datatypes, and it would be a hinderance to expect an implementation to have a *value* field internal to each usage of **ST**. For example, **CWE** (Figure C.1) has the property *code*, which is declared to be of type **ST**, and an implementation would probably access the value of *code* through something like *cwe.code* rather than *cwe.code.value*.

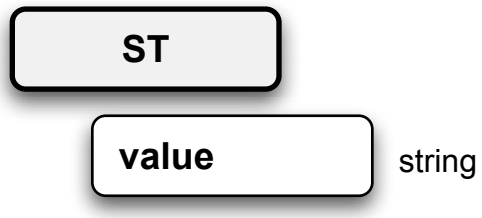


Figure I.1. Character String

Table I.1. ST Properties

Property	Type	Description
value	string	String of Characters

Table I.2. ST Constraint Properties

Property	Type	Description
max	INT	An integer value for the maximum string length
min	INT	An integer value for the minimum string length

I.2 Constraint Properties

I.2.1 max

The property *max* is an integer value for the maximum string length.

I.2.2 min

The property *min* is an integer value for the minimum string length.

I.3 HL7 Comparison

The **HL7:ST** datatype is shown in Figure I.2 and the comparison of **ST** to **HL7:ST** is shown in Figure I.3. In the HL7 hierarchy, **HL7:ST** is a subtype of **HL7:ED** with *mediaType* fixed to text/plain data. We have not physically implemented this as a subtype, but it is implied.

```
type CharacterString alias ST specializes ED {
   INT       length;
   ST.NT     headCharacter;
   ST.NT     tailString;
   DSET<ST.NT> translation;
};
```

Figure I.2. HL7:ST declaration.

Figure I.3. ST to HL7:ST Comparison

I.3.1 value

We have added the property *value* as a direct replacement for the property *data* which is inherited in HL7 from **HL7:ED**.[1]

I.3.2 data

We do not include the property *data*, and instead put the string characters in the property *value*.

I.3.3 mediaType

We do not include the property *mediaType*, as this is fixed to "text/plain."

[1]Requires further research. This property name change was done to avoid the awkward path notation "data.st.data," yet we have not corrected **HL7:ED** and still have the awkward "data.ed.data." These should be harmonized to operate in the same manner.

I.3.4 charset

We do not include the property *charset*.[2].

I.3.5 language

We do not include the property *language*.[3].

I.3.6 length

We do not include the property *length*. This can be calculated from the value.

I.3.7 headCharacter

We do not include the property *headCharacter*. This can be calculated from the value.

I.3.8 tailString

We do not include the property *tailString*. I am not sure if this can be derived from the value.[4]

I.3.9 translation

We do not include the property *translation*.[5].

I.3.10 Other Properties Inherited from HL7:ED

Please see Appendix K for additional properties not stated here. Also note that since the string characters are fixed to "text/plain" and cannot be compressed, that the properties *compression*, *reference*, *integrityCheck*, *algorithm*, and *thumbnail* are not needed in this context.

---

[2]Never considered, needs further research.

[3]Never considered, needs further research.

[4]Requires further research.

[5]Never considered, needs further research.

APPENDIX J

TS

Point in Time (**TS**), shown in Figure J.1, is used to store a point in time. Operator was added by Intermountain to represent concepts such as "greater than" or "less than." The format recommended for value is that specified by constrained ISO 8601 (a simpler ISO 8601 variant), which is defined in ISO 8824 (ASN.1) under clause 32 (generalized time). The syntax is YYYYMMDDHHMMSS.UUUU[+|-ZZzz]. For example, the literal form for April 1, 2000, 3:15 and 20.34 Eastern Standard Time is "20000401031520.34-0500." Note the dash is a minus sign and begins the time zone.

It should be noted that the XML ITS of the HL7 version 3 specification has a different format and follows the literal form defined by the XML Schema Part 2: Datatypes specification available from the World Wide Web Consortium.

To represent "greater than" or "less than" a particular point in time, ECIS uses **TS** with a coded operator, whereas HL7 would use another datatype – IVL<TS>. For simplicity, we are not implementing IVL<TS>.

If value can support specification of time zone, we do not need an additional timeZone component. If value cannot specify time zone, we need the timeZone component. The properties of **TS** are described in Table J.1.

J.1 Properties

J.1.1 value

The property *value* is often represented as a calendar expression, which is a text string. The constrained ISO 8601 (simper ISO 8601 variant), which is defined in ISO 8824 (ASN.1) under clause 32 (generalized time), is well established in use in HL7. The simpler ISO 8601 variant has no decorating dashes, colons, and no "T" between the date and time. The syntax is YYYYMMDDHHMMSS.UUUU[+|-ZZzz].

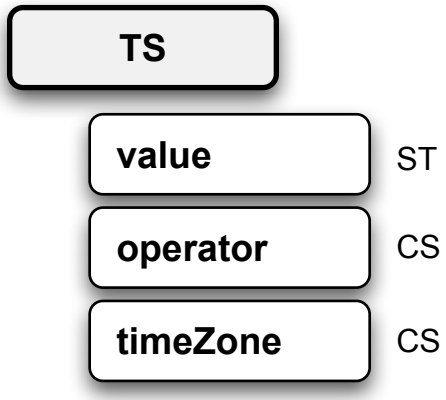


Figure J.1. Point in Time

Table J.1. TS Properties

Property	Type	Description
value	ST	UTC representation of the point in time
operator	CS	>, <, >=, or <=
timeZone	CS	Time zone of the point in time

J.1.2 operator

The property *operator* is a coded value that will represent either >, <, >=, or <=.

J.1.3 timeZone

The property *timezone* is used to store time zone information. The value for timeZone is a code from the time zone code domain in our own code system. Because we would convert all the point in time data to a standard time zone like GMT before storing them in our clinical data repository, and we cannot guarantee that we will always have geographic information stored. We need local time zone information to allow us to convert the time back to the local time the data were collected. One of the clinical examples where being able to convert back to local time is very useful is, when a patient has seizure, we not

only want to know the time the seizure happened, but also would like to know this patient's seizures usually happen in the morning. Without time zone and geographic location information, we loose the ability to know the seizure happens in the morning.

J.2 HL7 Comparison

The **HL7:TS** datatype is shown in Figure J.2 and the comparison of **TS** to **HL7:TS** is shown in Figure J.3. We have added the coded *operator* property to represent "greater than" or "less than" a Point in Time. For concrete intervals of time, we will define two separate Clinical Element models, one for each end point.

We have added a coded *timeZone* property because a UTC requires a comparison with a geographic location to retrieve the true local time. This has to do with the fact that not all locations follow daylight savings time.

J.2.1 value

Like the numeric datatypes, HL7 does not have a field for the value of **HL7:TS** and leaves this to the implementation. As in our other datatypes, we have chosen to use the property *value*.

```
type PointInTime alias TS specializes QTY {
           PQ   offset;
           CS   calendar;
           INT  precision;
           PQ   timezone;
           BL   equal(ANY x);
           TS   plus(PQ x);
           TS   minus(PQ x);
           PQ   minus(TS x);
  literal  ST.SIMPLE;
};
```

Figure J.2. HL7:TS declaration.

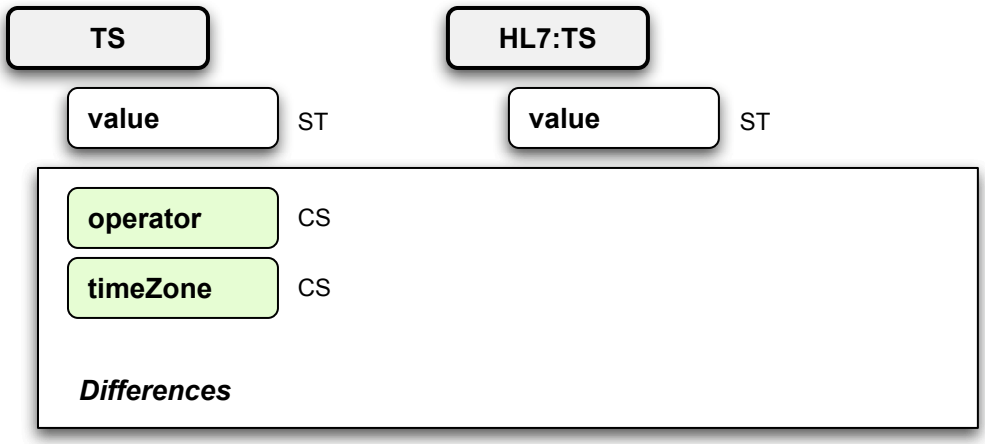


Figure J.3. TS to HL7:TS Comparison

J.2.2 operator

As in our numeric datatypes, we have added the property *operator* to signify "greater than" and "less than."

J.2.3 offset

We do not include the property *offset* which is defined as the elapsed time since any constant epoch, measured as a physical quantity in the dimension of time. We do not yet have a use case where the normal representation does not suffice.

J.2.4 calendar

We do not include the property *calendar* and default this value to the Gregorian calendar. This is the default calendar used by HL7.

J.2.5 precision

We do not include the property *precision* which is the number of significant digits of the calendar expression representation. The precision can be calculated from the current *value*.

J.2.6 timzone

We do include the property *timezone*, but change the type to a coded **CS** rather than use an **HL7:PQ**.

J.2.7 equal

We do not include the property *equal* because it is a comparison operator.

J.2.8 plus

We do not include the property *equal* because it is a calculation operator.

J.2.9 minus

We do not include the property *equal* because it is a calculation operator.

APPENDIX K

ED

Encapsulated Data (**ED**), shown in Figure K.1, is used to convey any data from a plain character string, formatted text, to multimedia binary data. It may also contain formatted data from another standard such as XML. The properties of **ED** are described in Table K.1.

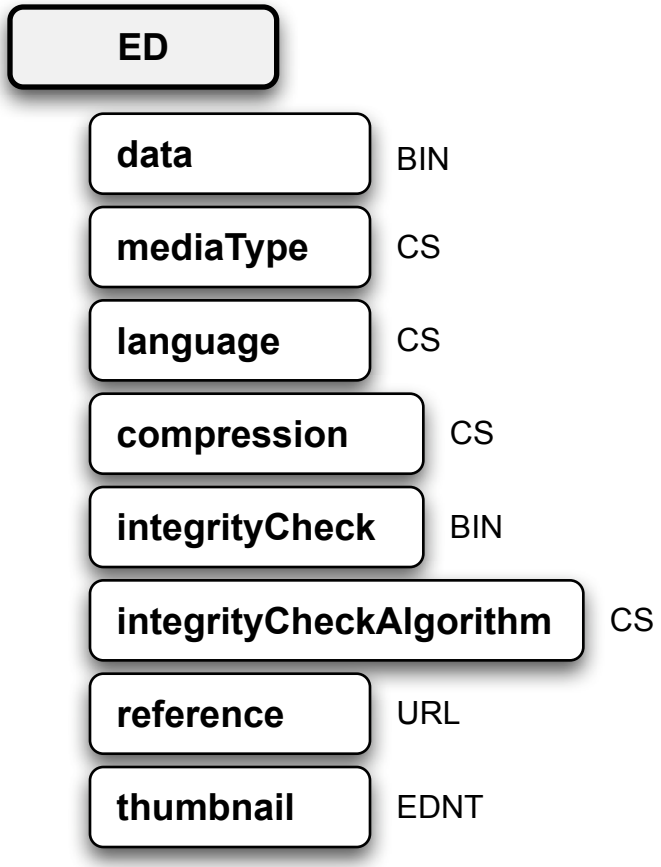


Figure K.1. Encapsulated Data

Table K.1. ED Properties

Property	Type	Description
data	BIN	Binary data
mediaType	CS	Type of binary data
language	CS	Human Language
compression	CS	Compression Algorithm of binary data
integrityCheck	CS	Value generated by Integrity Check Algorithm
integrityCheckAlgorithm	CS	Algorithm used to generate value
reference	URL	Pointer to externally stored binary data
thumbnail	EDNT	Abbreviated rendition of binary data

K.1 Properties

K.1.1 data

The property *data* represents raw binary data. We have decided to store large binary data, such as X-Rays, in an external data store, but these data will still be present in the *data* property over the wire. The property *reference* will contain the pointer to the data in the external data store. For smaller binary data, they will actually be stored in the *data* property, and *reference* will be null. The services should function in a seamless manner, so that users of the service need not be aware whether the data were stored inline or as part of the external data store.

K.1.2 mediaType

The property *mediaType* represents the type of the encapsulated data and identifies a method to interpret or render the data.

K.1.3 language

The property *language* represents character-based information specifying the human language of the text.

K.1.4 compression

The property *compression* indicates whether the raw byte data is compressed, and what compression algorithm was used.

K.1.5 integrityCheck

The property *integrityCheck* is a short binary value representing a cryptographically strong checksum that is calculated over the binary data. The purpose of this property, when communicated with a reference, is for anyone to validate later whether the reference still resolves to the same data that the reference resolved to when the encapsulated data value with reference was created.

The integrity check is calculated over the raw binary data that are contained in the data component, or that is accessible through the reference. No transformations are made before the integrity check is calculated. If the data are compressed, the integrity check is calculated over the compressed data.

K.1.6 integrityCheckAlgorithm

The property *integrityCheckAlgorithm* specifies the algorithm used to compute the integrityCheck value. The cryptographically strong checksum algorithm Secure Hash Algorithm-1 (SHA-1) is currently the industry standard. It superseded the MD5 algorithm several years ago, when certain flaws in the security of MD5 were discovered. Currently, the SHA-1 hash algorithm is the default choice for the integrity check algorithm. Note that SHA-256 is also entering widespread usage.

K.1.7 reference

The property *reference* is a URL which will resolve to precisely the same binary data that could as well have been provided as inline data. This serves as the pointer to the external data source where the data of **ED** is actually stored.

An IHE Profile will be used to define the format of URL. The IHE profile basically specifies a common URL format to use and calls for the use of a UID or OID to reference

the thing on the other end. The consistent URL format makes it possible to rewrite a stored URL to hit a different server using the same query at a later time.

K.1.8 thumbnail

The property *thumbnail* is an abbreviated rendition of the full data. A thumbnail requires significantly fewer resources than the full data, while still maintaining some distinctive similarity with the full data. A thumbnail is typically used with by-reference encapsulated data. It allows a user to select data more efficiently before actually downloading through the reference.

K.2 HL7 Comparison

The **HL7:ED** datatype is shown in Figure K.2 and the comparison of **ED** to **HL7:ED** is shown in Figure K.3. HL7 uses **HL7:TEL** for the *reference* property, which we have substituted with **URL**. Another small change we make to avoid recursion in the *thumbnail* property is to assign **EDNT** rather than **ED**.

```
type EncapsulatedData alias ED specializes ANY {
   BIN     data;
   CS      mediaType;
   CS      charset;
   CS      language;
   CS      compression;
   TEL.URL reference;
   BIN     integrityCheck;
   CS      integrityCheckAlgorithm;
   ST      description;
   ED      thumbnail;
   DSET<ED> translation;
   INT     length;
   ED      subPart(INT start, INT end);
   BL      equal(ANY x);
};
```

Figure K.2. HL7:ED declaration.

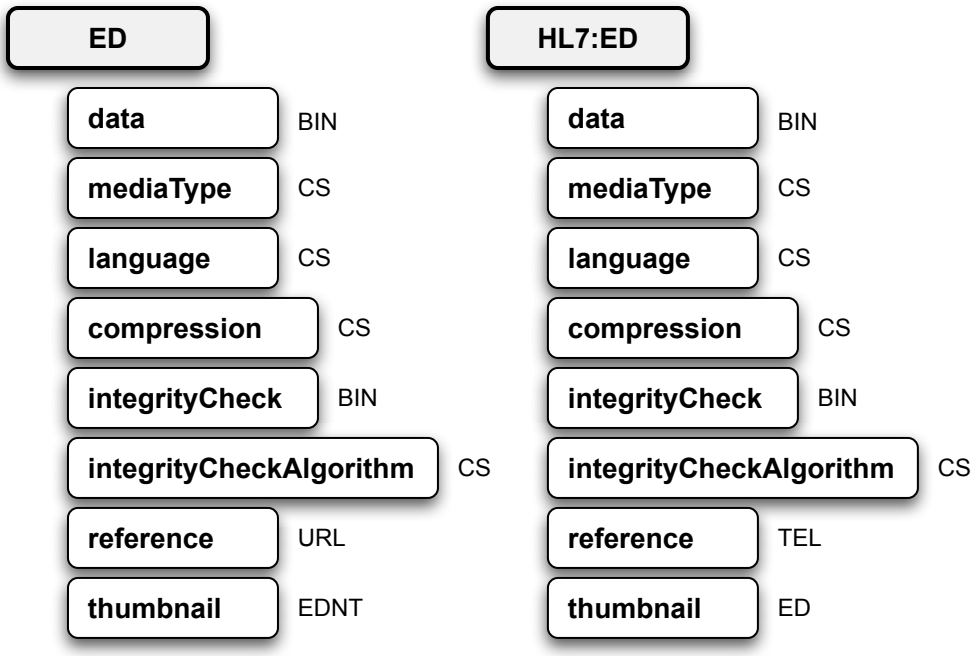


Figure K.3. ED to HL7:ED Comparison

K.2.1 data

We include the property *data*.[1]

K.2.2 mediaType

We include the property *mediaType*.

K.2.3 charset

We do not include the property *charset* for character-based encoding.[2]

K.2.4 language

We include the property *language* which is used for character-based data.

---

[1]Requires further research. The awkward path statement "data.ed.data" exists, and this property needs to be harmonized with the corresponding property in **ST**.

[2]Requires further research.

K.2.5 compression

We include the property *compression* which is used to identify the compression algorithm used.

K.2.6 reference

We include the property *reference* to identify external references.

K.2.7 integrityCheck

We include the property *integrityCheck* but currently have this defined as a type **CS** instead of a **BIN**. I think this may be an error.[3]

K.2.8 integrityCheckAlgorithm

We include the property *integrityCheckAlgorithm*.

K.2.9 description

We do not include the property *description*. This property is intended to be a short description of the media contained in case the media cannot be presented.[4]

K.2.10 thumbnail

We include the property *thumbnail*, but change the type from **HL7:ED** to our **EDNT** which does not allow a thumbnail. This prevents a recursive nesting of thumbnails.

K.2.11 translation

We do not include the property *translation*, which allows for alternate renditions of the same content translated into a different language or a different mediaType.[5]

---

[3] Requires further research.

[4] Requires further research.

[5] Requires further research.

K.2.12 length

We do not include the property *length*, as the length of the binary data can be calculated by the implementation.

K.2.13 subpart

We do not include the property *subpart*, as this is an operator on the stored data.

K.2.14 equal

We do not include the property *equal*, as this is a comparison operator on the stored data.

APPENDIX L

IVLPQ

Interval Physical Quantity (**IVLPQ**), shown in Figure L.1, is used to represent an interval of Physical Quantities. We will only use **IVLPQ** for closed intervals, because open intervals can be handled by our **PQ** using the *operator* property. The properties of **IVLPQ** are described in Table L.1.

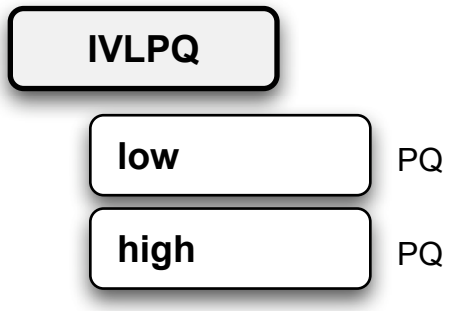


Figure L.1. Interval Physical Quantity

Table L.1. IVLPQ Properties

Property	Type	Description
low	PQ	The low end of the interval
high	PQ	The high end of the interval

L.1 Properties

L.1.1 low

The property *low* represents the low value of the interval.

L.1.2 high

The property *high* represents the high of the interval.

L.2 HL7 Comparison

The **HL7:IVL<PQ>** datatype is shown in Figure L.2 and the comparison of **IVLPQ** to **HL7:IVL<PQ>** is shown in Figure L.3. The HL7 datatype **IVL<PQ>** has been restructured to no longer include a high and low **HL7:PQ**, which resulted in redundant units of measure. Now the unit of measure has been promoted, and the interval is instead defined by real numbers.[1]

We have removed the properties *width* and *center* from **HL7:IVL<PQ>** because both are calculated values. If it is considered important for us to index on these properties, we could add them back at a later time.

We have also removed the properties *lowClosed* and *highClosed* because they will always be true in our case.

L.2.1 low

We include the property *low*, but it is currently a **PQ** instead of a **REAL**.

L.2.2 lowClosed

We do not include the property *lowClosed*, and always assume lowClosed is true, which means the lower point is included in the interval.[2]

L.2.3 high

We include the property *high*, but it is currently a **PQ** instead of a **REAL**.

---

[1] Requires further research, with a possible remodeling.

[2] Requires further research.

```
type Interval<T> alias IVL<T> specializes QSET<T> {
              T       low;
              BL      lowClosed;
              T       high;
              BL      highClosed;
              QTY     width;
              T       center;
              T       any;
              IVL<T>  hull;
              IVL<T>  hull(IVL<T> x);
   literal    ST.SIMPLE;
   promotion  IVL<T>  (T x);
   demotion   T;
};

type Interval<PQ> alias IVL<PQ> {
   IVL<REAL> value;
   CS  unit;
};
```

Figure L.2. HL7:IVL<PQ> declaration.

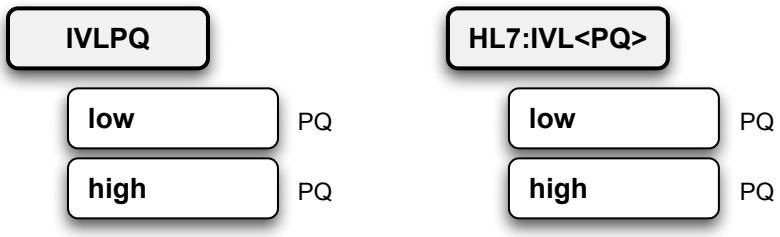


Figure L.3. IVLPQ to HL7:IVL<PQ> Comparison

L.2.4 highClosed

We do not include the property *highClosed*, and always assume highClosed is true, which means the higher point is included in the interval.[3]

[3]Requires further research.

L.2.5 width

We do not include the property *width*, as this is a calculation based on the high and low values.

L.2.6 center

We do not include the property *center*, as this is a calculation based on the high and low values.

L.2.7 any

We do not include the property *any*, which specifies that nothing is known about the interval except that some particular value lies within the interval.[4]

L.2.8 hull

We do not include the property *hull*, because this a calculation.

---

[4]The same can be accomplished with a qualifier or this can be known from the context in a model. Requires further research.

APPENDIX M

RTOPQ

Ratio Physical Quantity (**RTOPQ**), shown in Figure M.1, is used to represent a quantity constructed through the division of a numerator quantity with a denominator quantity. The RTO data type supports titers (e.g., "1:128") and other quantities produced by laboratories that truly represent ratios. Blood pressure measurements (e.g., "120/60") are not ratios. The properties of **IVLPQ** are described in Table M.1.

M.1 Properties

M.1.1 numerator

The property *numerator* stands for the numerator of the ratio.

M.1.2 denominator

The property *denominator* is the denominator of the ratio.

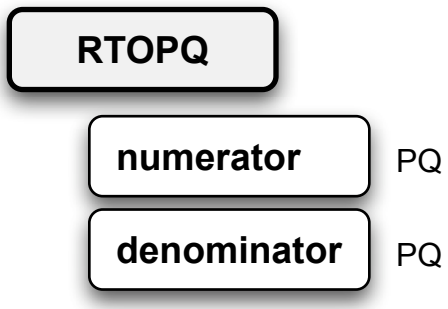


Figure M.1. Ratio Physical Quantity

Table M.1. RTOPQ Properties

Property	Type	Description
numerator	PQ	The numerator of the ratio
denominator	PQ	The denominator of the ratio

M.2 HL7 Comparison

The **HL7:RTO<PQ>** datatype is shown in Figure M.2 and the comparison of **RTOPQ** to **HL7:RTO<PQ>** is shown in Figure M.3. There are no structural differences between our **RTOPQ** and **HL7:RTO<PQ>** at least at the level these are defined. However, it should be noted that the **PQ** and HL7:PQ which are internally referenced by these two types are different.

M.2.1 numerator

We include the property *numerator*.

M.2.2 denominator

We include the property *denominator*.

```
type Ratio<QTZ N, QTZ D> alias RTO specializes QTY {
   N         numerator;
   D         denominator;
   literal   ST.SIMPLE;
   demotion  REAL;
   demotion  PQ;
};
```

Figure M.2. HL7:RTO<PQ> declaration.

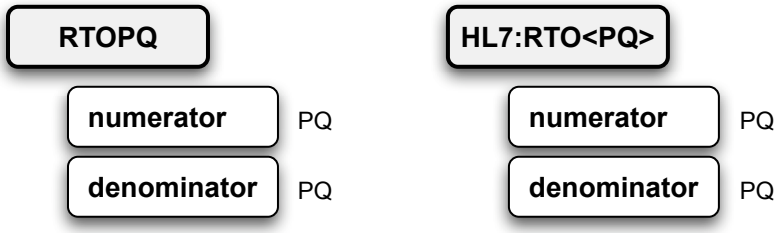


Figure M.3. RTOPQ to HL7:RTO<PQ> Comparison

APPENDIX N

PQT

Physical Quantity Translation (**PQT**), shown in Figure N.1, is used to store a translation of a real number value with a real physical unit to represent quantities such as 2.3 milligrams, 24 days, or 2 drops. The translation could simply be a different representation of the same unit, but from a different coding system, in which case, the real number value would be identical. Or, the unit could be different conceptually, and in this case, the real number value is converted to match this new unit. **PQT** like **PQ** also allows representation of "greater than" and "less than" some physical quantity using the *operator* property.

For storage, we only allow one **PQT** within the *translation* section of **PQ**. If this translation exists, it is always the original physical quantity, and the representation in the upper section of the **PQ** is a normalized value taken from this original physical quantity. The properties of **PQT** are described in Table N.1.

It was decided that *originalText* was not needed in **PQT**. This is because the original form is always stored in the top section of the **PQ** datatype, and any **PQT** is simply a translation. This is in contrast to **CWE** where the original code and representation could be in the translation section.

N.1 Properties

N.1.1 value

The property *value* is the magnitude of the quantity measured in terms of the *unit*.

N.1.2 unit

The property *unit* is a code for unit of measure, e.g., mg, seconds.

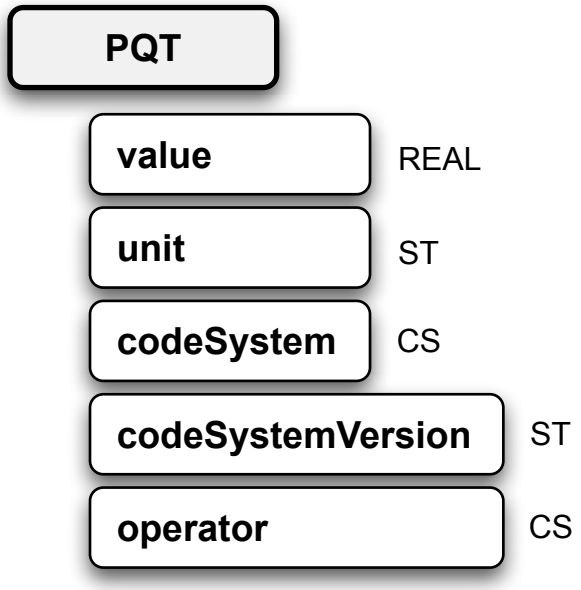


Figure N.1. Physical Quantity Translation

N.1.3 codeSystem

The property *codeSystem* is code for the coding system from which the value in *unit* is defined.

N.1.4 codeSystemVersion

The property *codeSystemVersion* is the version of the coding system.

N.1.5 operator

The property *operator* is a coded value that will represent either >, <, >=, or <=.

N.2 HL7 Comparison

The **HL7:PQR** datatype is shown in Figure N.2 and the comparison of **PQT** to **HL7:PQR** is shown in Figure N.3. HL7 derived their **HL7:PQR** from the **HL7:CD** datatype and then added the property *value*. For this reason, all the data regarding the unit of measure go into the code-related fields from **HL7:CD**. This is a little confusing to users, because the unit of

Table N.1. PQT Properties

Property	Type	Description
value	REAL	The magnitude of the quantity measured in terms of the *unit*
unit	ST	A code for unit of measure, e.g., mg, seconds
codeSystem	CS	The coding system from which *unit* is defined
codeSystemVersion	ST	The version of the coding system
operator	CS	>, <, >=, or <=

```
flavor CodedValue alias CV constrains CD;

invariant (CV x) where x.nonNull {
  x.translation.isEmpty;
  x.source.isNull;
};

protected type PhysicalQuantityRepresentation alias PQR specializes CV {
   REAL  value;
};
```

Figure N.2. HL7:PQR declaration.

measure is now in a property called *code*. Because of this confusion, we renamed the *code* property to *unit*, but we left the confusion of *codeSystem* and *codeSystemVersion* which now are details regarding the code within the *unit* property. Just as in **PQ**, we represent "greater than" and "less than" with the *operator* property.

N.2.1 value

We include the property *value*.

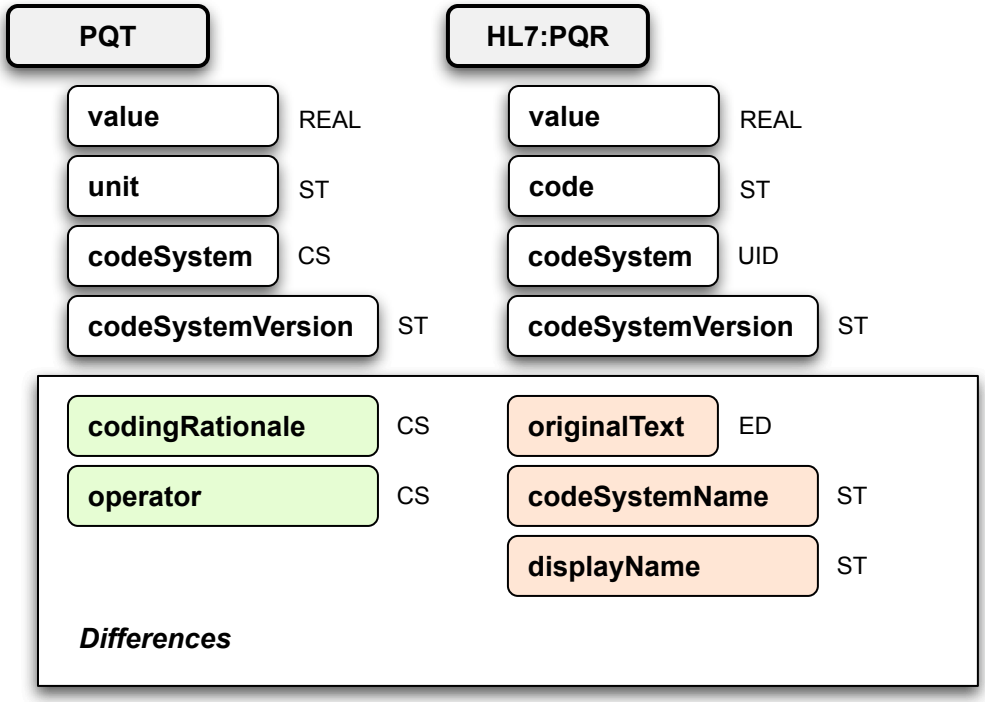


Figure N.3. PQT to HL7:PQR Comparison

N.2.2 operator

The property *operator* is not part of the HL7 abstract definition. While HL7 would use an **IVL<PQ>** to represent "greater than" or "less than" a real number, we will also use a coded value within **PQT** to represent this.

N.2.3 unit

We do include the property *unit* but we have changed the type to our **CWENT** because we want to be able to store *originalText* along with the code. In addition, HL7 uses UCUM codes and we are using codes from our primary coding system. We do not want to internally use multiple coding systems, and in addition, UCUM does not support concepts likes "drops" since they are not truly quantitative.

N.2.4 code

We do not include the property *code*. We have renamed this property *unit*.

N.2.5 displayName

We do not include *displayName*. We do face some danger here if the code came from an external system and the unit code is obtuse. We could put something readable into *originalText*.[1]

N.2.6 codeSystem

We do include the property *codeSystem*.

N.2.7 codeSystemName

We do not include the property *codeSystemName* because this is just a representation of the *codeSystem* and can be provided by the vocabulary server.

N.2.8 codeSystemVersion

We do include *codeSystemVersion*.

N.2.9 valueSet

We do not currently include the *valueSet*, which specifies the value set that applied when this PQR was created.[2]

N.2.10 valueSetVersion

We do not currently include the *valueSetVersion*, which specifies the value set version that applied when this PQR was created. Even if we decided to add the *valueSet* property, I believe the version could be handled by the vocabulary server.

---

[1] Requires further research.

[2] Requires further research. HL7 does not remove this from CD, but they do not discuss it.

N.2.11 originalText

We include the property *originalText*.

N.2.12 codingRationale

We do not include the property *codingRationale*. We need to examine use cases and come to a conclusion on this.[3]

N.2.13 isCompositional

We do not include the property *isCompositional*. This can be derived from the vocabulary server.

N.2.14 equal

We do not include the property *equal*. This is a comparison operation that can be handled by the vocabulary server.

N.2.15 implies

We do not include the property *implies*. This is a comparison operation that can be handled by the vocabulary server.

N.2.16 Properties inherited from HL7:ANY

Please see Appendix B.

---

[3]This implementation only uses part of the functionality that *codingRationale* provides, and we have not researched the other aspects.

APPENDIX O

CET

Coded With Exceptions - Translation (**CET**), shown in Figure O.1, is used internally[1] within the **CWE** datatype, and is used to store translations of the primary code. Multiple **CETs** may be carried by any one **CWE** instance.

One example of its use would be if a code from an external coding system came over an interface, this external code would be placed in a **CET** datatype, which would then be stored in the translation section of a **CWE**, with the code from the primary coding system going in the top section of the **CWE** datatype. The properties of **CET** are described in Table O.1.

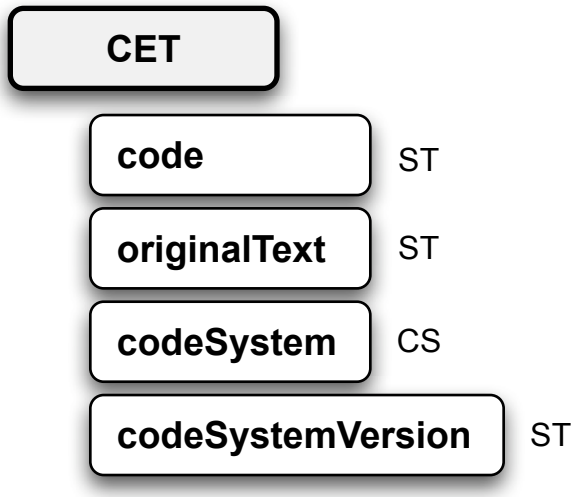


Figure O.1. Coded With Exceptions - Translation

[1]It is important to note that **CET** cannot be directly used within a Clinical Element model, but only within a **CWE** datatype.

Table O.1. CET Properties

Property	Type	Description
code	ST	A code for a concept defined in any code system.
originalText	ST	The textual representation of the code
codeSystem	CS	The coding system from which *code* is defined
codeSystemVersion	ST	The version of the coding system

O.1 Properties

O.1.1 code

The property *code* is the code for a concept defined in any code system; this code system is specified in *codeSystem*.

O.1.2 originalText

The property *originalText* is a textual representation of the code that was presented to the user, or the representation that came over the interface.

O.1.3 codeSystem

The property *codeSystem* represents a coding system from which the value in *code* is defined.

O.1.4 codeSystemVersion

The property *CodeSystemVersion* represents a version of the coding system.

O.2 HL7 Comparison

The **HL7:CD** datatype is shown in Figure O.2 and the comparison of **CET** to **HL7:CD** is shown in Figure O.3. We have evolved **CET** from the **HL7:CD** datatype. In doing so, we stripped out various properties we did not need, as noted in **CWE**. **HL7:CD** translation has recursive translations. By defining **CET**, which does not itself contain a translation

```
type ConceptDescriptor alias CD specializes ANY {
    ST.SIMPLE         code;
    ST                displayName;
    UID               codeSystem;
    ST.NT             codeSystemName;
    ST.SIMPLE         codeSystemVersion;
    UID               valueSet;
    TS.DATETIME.FULL  valueSetVersion;
    ED.TEXT           originalText;
    SET<CS>           codingRationale;
    DSET<CD>          translation;
    CD                source;
    BL                isCompositional;
    BL                equal(ANY x);
    BL                implies(CD x);
};
```

Figure O.2. HL7:CD declaration.

component, we avoid nested translations. **HL7:CD** defines *originalText* as an **HL7:ED** datatype. We decided that the complexity of the **HL7:ED** datatype was not worth the cost, so we defined *originalText* to be an **ST**. If a translation contains a file that requires the complexity of the **HL7:ED** datatype, then that file would be handled at the **CWE** level.

O.2.1 code

We include the property *code*.

O.2.2 displayName

We do not include *displayName*. We do face some danger here if the code came from an external system and the unit code is obtuse. We could put something readable into *originalText*.[2]

O.2.3 codeSystem

We do include the property *codeSystem*.

[2]Requires further research.

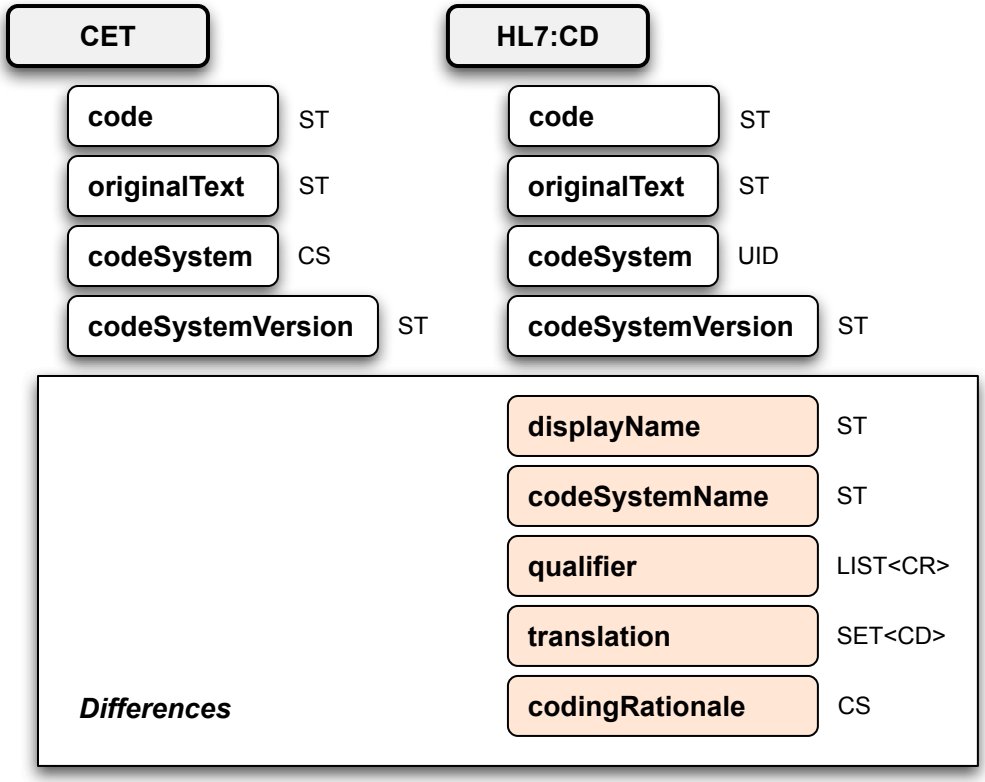


Figure O.3. CET to HL7:CD Comparison

O.2.4 codeSystemName

We do not include the property *codeSystemName* because this is just a representation of the *codeSystem* and can be provided by the vocabulary server.

O.2.5 codeSystemVersion

We do include *codeSystemVersion*.

O.2.6 valueSet

We do not currently include the *valueSet*, which specifies the value set that applied when this CD was created.

O.2.7 valueSetVersion

We do not currently include the *valueSetVersion*, which specifies the value set version that applied when this CD was created. Even if we decided to add the *valueSet* property, I believe the version could be handled by the vocabulary server.

O.2.8 originalText

We include the property *originalText*.

O.2.9 codingRationale

We do not include the property *codingRationale*. We need to examine use cases and come to a conclusion on this.[3]

O.2.10 isCompositional

We do not include the property *isCompositional*. This can be derived from the vocabulary server.

O.2.11 equal

We do not include the property *equal*. This is a comparison operation that can be handled by the vocabulary server.

O.2.12 implies

We do not include the property *implies*. This is a comparison operation that can be handled by the vocabulary server.

O.2.13 Properties inherited from HL7:ANY

Please see Appendix B.

---

[3]This implementation only uses part of the functionality that *codingRationale* provides, and we have not researched the other aspects.

APPENDIX P

CNE

Coded No Exceptions (**CNE**), shown in Figure P.1, requires that a code always be present. The properties are described in Table P.1 and constraint properties are described in Table P.2. Due to the performance requirements of a permanent storage-based system, we have chosen to only allow codes from the primary coding system in *code*, and alternate codes from other coding systems will be placed in translation.

P.1 Properties

P.1.1 code

The property *code* contains a code for a concept defined in the primary coding system.

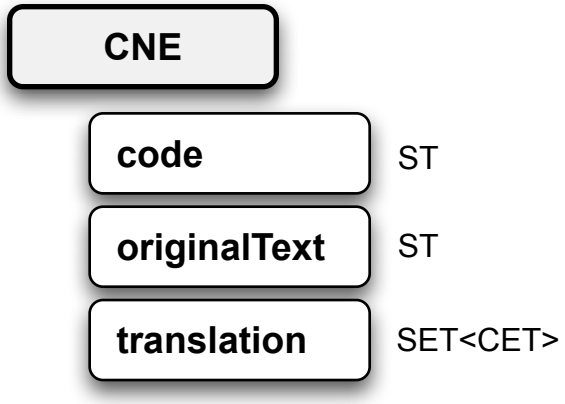


Figure P.1. Coded No Exceptions

Table P.1. CNE Properties

Property	Type	Description
code	ST	A code for a concept defined in the primary code system
originalText	ST	Textual representation of the code
translation	SET<CET>	Zero to many translations of the code

Table P.2. CWE Constraint Properties

Property	Type	Description
domain	ST	A code for a domain of concepts

P.1.2 originalText

The property *originalText* is used for textual representation of the code that was presented to the user, or the representation that came over the interface.

P.1.3 translation

The property *translation* is used to represent Zero to many translations of the code from any code system.

P.2 Constraint Properties

P.2.1 domain

The property *domain* is a code for a domain of concepts defined in the primary code system.

P.3 HL7 Comparison

The Clinical Element **CNE** datatype is derived from the structure of the **HL7:CD** datatype, shown in Figure P.2, but is actually similar in function to **HL7:CD** with the

```
type ConceptDescriptor alias CD specializes ANY {
    ST.SIMPLE         code;
    ST                displayName;
    UID               codeSystem;
    ST.NT             codeSystemName;
    ST.SIMPLE         codeSystemVersion;
    UID               valueSet;
    TS.DATETIME.FULL  valueSetVersion;
    ED.TEXT           originalText;
    SET<CS>           codingRationale;
    DSET<CD>          translation;
    CD                source;
    BL                isCompositional;
    BL                equal(ANY x);
    BL                implies(CD x);
};
```

Figure P.2. HL7:CD declaration.

CNE coding strength qualifier. See Table P.3 for a description of **HL7:CD** with its coding strength qualifiers. The comparison of **CNE** to **HL7:CD** is shown in Figure P.3.

P.3.1 code

We include the property *code*.

P.3.2 displayName

We do not include *displayName* due to the fact that we only allow codes from our primary coding system in **CWE**, and a displayName can be generated at any time from the vocabulary server.

P.3.3 codeSystem

We do not include *codeSystem* because in **CWE**, the *codeSystem* is defaulted to the primary coding system.

Table P.3. HL7 CE and Coding Strength Qualifiers.

Value	Description
CWE - Coded With Exceptions	This is a coding strength qualifier that is applied to a binding of a vocabulary domain with a **CD**. It signifies that a code outside the specified value set/code system may be used, or that free text may be used in lieu of a code.
CNE - Coded, No Exceptions	This is another coding strength qualifier that is applied to a binding of a vocabulary domain with a **CD**. It signifies that a code outside the specified value set/code system is not permitted, nor is free text permitted in lieu of a code.

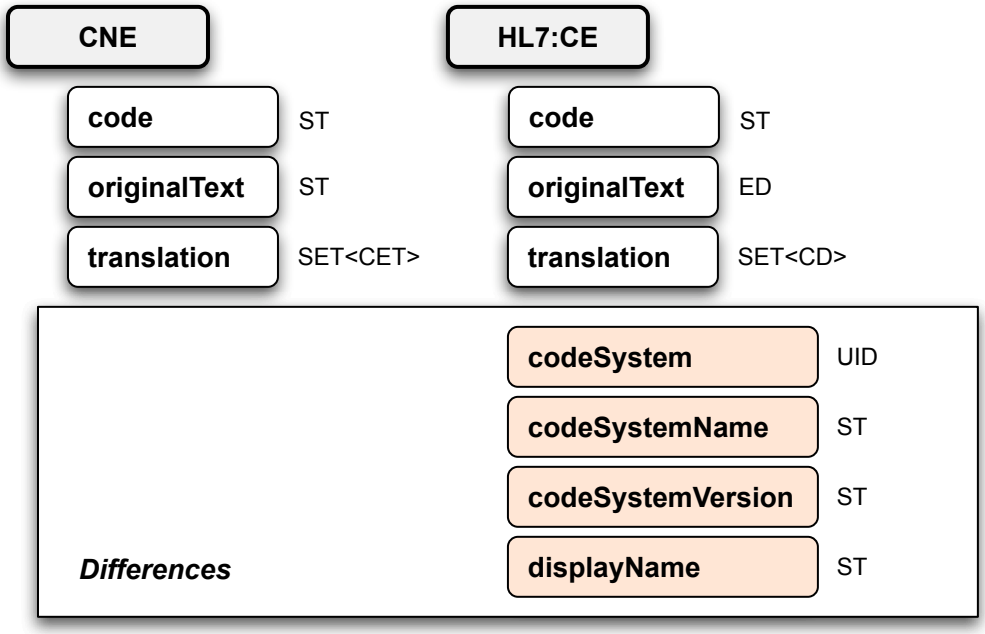


Figure P.3. CNE to HL7:CE Comparison

P.3.4 codeSystemName

We do not include *codeSystemName* because in **CWE**, the *codeSystemName* is defaulted to the primary coding system.

P.3.5 codeSystemVersion

We do not include *codeSystemVersion* because in our primary coding system, we do not use versioning.[1]

P.3.6 valueSet

We do not currently include the *valueSet*, which specifies the value set that applied when this CD was created.[2]

P.3.7 valueSetVersion

We do not currently include the *valueSetVersion*, which specifies the value set version that applied when this CD was created. Even if we decided to add the *valueSet* property, I believe the version could be handled by the vocabulary server.

P.3.8 originalText

We include the property *originalText*.

P.3.9 codingRationale

We do not include the property *codingRationale*. For better or worse, we assume it is original if there is not a translation.[3]

P.3.10 translation

We included the property *translation*.

---

[1] Requires further research.

[2] Requires further research.

[3] This implementation only uses part of the functionality that *codingRationale* provides, and we have not researched the other aspects.

P.3.11 source

We do not include the property *source*. This property identifies the translation from which this was translated. We have not discussed this issue, but it seems this is implicit due to the fact we are only allowing one translation in the storage form. The question remains whether or not this is important transactionally when we allow more than one translation.

P.3.12 isCompositional

We do not include the property *isCompositional*. This can be derived from the vocabulary server.

P.3.13 equal

We do not include the property *equal*. This is a comparison operation that can be handled by the vocabulary server.

P.3.14 implies

We do not include the property *implies*. This is a comparison operation that can be handled by the vocabulary server.

P.3.15 Properties inherited from HL7:ANY

Please see Appendix B.

APPENDIX Q

COT

Coded Ordinal - Translation (**COT**), shown in Figure Q.1, is used internally within the **CO** datatype, and is used to store translations of the primary code. Multiple **COTs** may be carried by any one **CO** instance. One example of its use would be if a code representing an ordered concept from an external coding system came over an interface, this external code would be placed in a **COT** datatype, which would then be stored in the translation section of a **CO**, with the a code from the primary coding system going in the top section of the **CO** datatype. The properties of **COT** are described in Table Q.1.

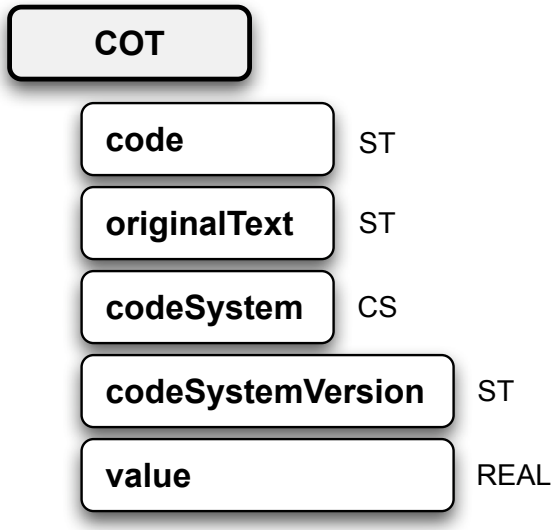


Figure Q.1. Coded Ordinal - Translation.

Table Q.1. CET Properties

Property	Type	Description
code	ST	A code for a concept defined in any code system
originalText	ST	The textual representation of the code
codeSystem	CS	The coding system from which *code* is defined
codeSystemVersion	ST	The version of the coding system
value	REAL	A numeric representation of the code's meaning

Q.1 Properties

Q.1.1 code

The property *code* is the code for a concept defined in any code system; this code system is specified in *codeSystem*.

Q.1.2 originalText

The property *originalText* is a textual representation of the code that was presented to the user, or the representation that came over the interface.

Q.1.3 codeSystem

The property *codeSystem* is a coding system from which *code* is defined.

Q.1.4 codeSystemVersion

The property *CodeSystemVersion* is a version of the coding system.

Q.1.5 value

The property *value* is a numeric representation of the code's meaning.[1]

[1]Previously called numericScore.

Q.2 HL7 Comparison

The **HL7:CO** datatype is shown in Figure Q.2 and the comparison of **COT** to **HL7:CD** is shown in Figure Q.3. We have evolved **COT** from the **HL7:CD** datatype. In doing so, we stripped out various properties we did not need, as noted in **CO**. **HL7:CD** translation has recursive translations. By defining **COT**, which does not itself contain a translation component, we avoid nested translations. **HL7:CD** defines *originalText* as an **HL7:ED** datatype. We decided that the complexity of the **HL7:ED** datatype was not worth the cost, so we defined *originalText* to be an **ST**. If a translation contains a file that requires the complexity of the **HL7:ED** datatype, then that file would be handled at the **CO** level.

Q.2.1 value

We include the property *value*.[2]

Q.2.2 lessOrEqual

We do not include the property *lessOrEqual* because it is a comparison operator.

Q.2.3 lessThan

We do not include the property *lessThan* because it is a comparison operator.

Q.2.4 greaterThan

We do not include the property *greaterThan* because it is a comparison operator.

[2]We previously called this *numericScore*.

```
type CodedOrdinal alias CO specializes CD {
   REAL value;
   BL  lessOrEqual(CO o);
   BL  lessThan(CO o);
   BL  greaterThan(CO o);
   BL  greaterOrEqual(CO o);
};
```

Figure Q.2. HL7:CO declaration.

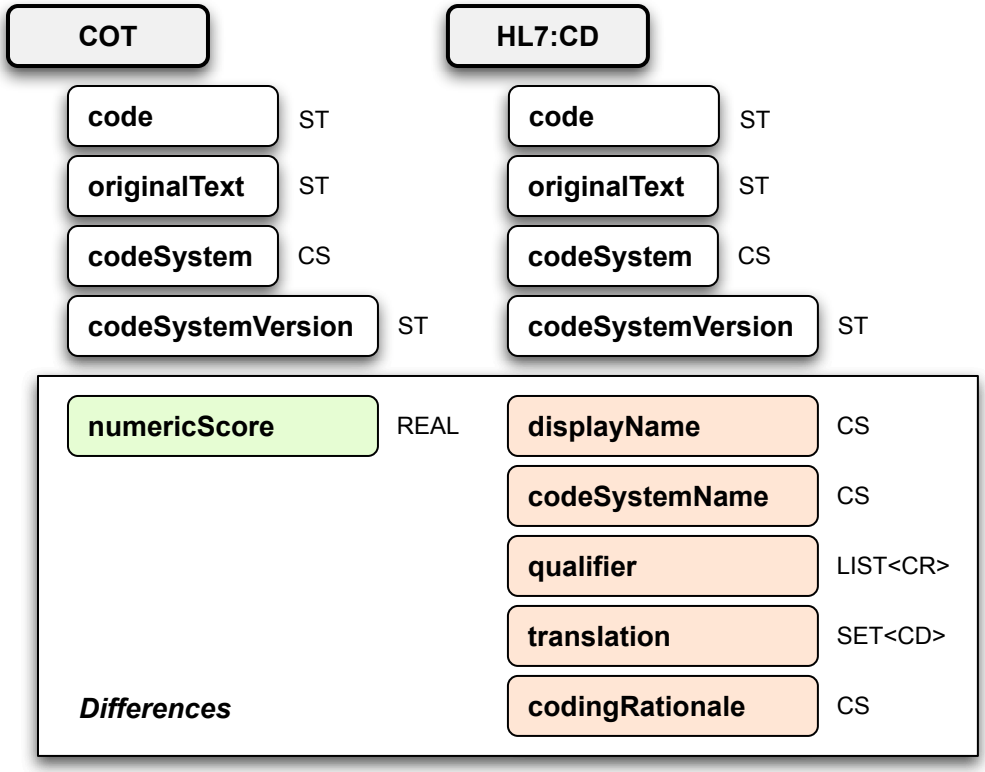


Figure Q.3. COT to HL7:CD Comparison

Q.2.5 greaterOrEqual

We do not include the property *greaterOrEqual* because it is a comparison operator.

Q.2.6 Properties inherited from HL7:CD

Please see Appendix C.

APPENDIX R

CS

Coded Simple (**CS**), shown in Figure R.1), simply contains the property *code* shown in Table R.1. This datatype is not allowed for use in Clinical Element models, but it is used internally in the definitions of datatypes. In each use case, the domain of values is fixed by the implementation to a particular domain.[1]

Because the *code* property within each **CS** is defined as data type **ST**, which is String, some people may question the necessity of defining **CS**. The difference between **CS** and **ST** is that, for **CS**, the string value in the *code* property must come from a code system; you cannot simply put any string into **CS**. In the implementation, this means there will be a need to check whether the string value is a valid code from the specified domain in the primary coding system. The purpose of declaring a **CS** data type is to capture this conceptual meaning. But as stated above, we still need to formally declare these domains.

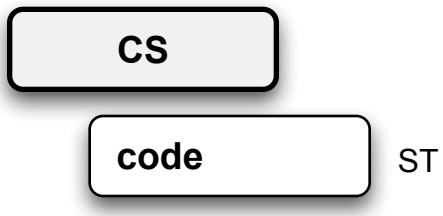


Figure R.1. Coded Simple

[1]These domains are only implied by the meaning of each use of **CS**, and we need to formally define each of these domains.

Table R.1. CS Properties

Property	Type	Description
code	ST	A code for a concept defined in the primary code system.

R.1 Properties

R.1.1 code

The property *code* is a code for a concept defined in the primary code system.

R.2 HL7 Comparison

The **HL7:CS** datatype is shown in Figure R.2 and the comparison of **CS** to **HL7:CS** is shown in Figure R.3. Our use of **CS** is now consistent with the current **HL7:CS**. It should be noted that a previous **HL7:CS** contained the property *originalText*, but they have now removed this as we had.

R.2.1 code

We include the property *code*.

R.2.2 Properties inherited from HL7:ANY

Please see Appendix B.

```
type CodedSimpleValue alias CS specializes CV {
            ST.SIMPLE  code;
   literal  ST.SIMPLE;
};
```

Figure R.2. HL7:CS declaration.

Figure R.3. CS to HL7:CS Comparison

APPENDIX S

CWENT

Coded With Exceptions - No Translation (**CWENT**), shown in Figure S.1, is identical to our datatype **CWE** except it does not allow translations of the primary code. It is only for internal use, and is not used in Clinical Element models. The properties are described in Table S.1 and constraint properties are described in Table S.2.

S.1 Properties

S.1.1 code

The property *code* is a code for a concept defined in the primary code system.

S.1.2 originalText

The property *originalText* is a textual representation of the code that was presented to the user, or the representation that came over the interface.

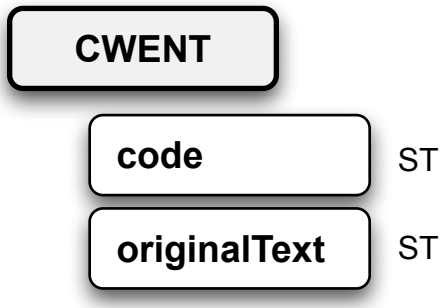


Figure S.1. Coded With Exceptions - No Translation

Table S.1. CWENT Properties

Property	Type	Description
code	ST	A code for a concept defined in the primary code system
originalText	ST	The textual representation of the code

Table S.2. CWE Constraint Properties

Property	Type	Description
domain	ST	A code for a domain of concepts

S.2 Constraint Properties

S.2.1 domain

The property *domain* represents code for a domain of concepts defined in the primary code system.

S.3 HL7 Comparison

Comparison to **HL7:CD** is identical to the comparison of **CWE** with **HL7:CD**, with the additional restriction that we have removed the *translation* property. The **HL7:CD** is shown in Figure S.2 and the comparison of **CWENT** to **HL7:CD** is shown in Figure S.3.

S.3.1 code

We include the property *code*.

S.3.2 displayName

We do not include *displayName* due to the fact that we only allow codes from our primary coding system in **CWE**, and a displayName can be generated at any time from the vocabulary server.

```
type ConceptDescriptor alias CD specializes ANY {
    ST.SIMPLE         code;
    ST                displayName;
    UID               codeSystem;
    ST.NT             codeSystemName;
    ST.SIMPLE         codeSystemVersion;
    UID               valueSet;
    TS.DATETIME.FULL  valueSetVersion;
    ED.TEXT           originalText;
    SET<CS>           codingRationale;
    DSET<CD>          translation;
    CD                source;
    BL                isCompositional;
    BL                equal(ANY x);
    BL                implies(CD x);
};
```

Figure S.2. HL7:CD declaration.

S.3.3 codeSystem

We do not include *codeSystem* because in **CWE**, the *codeSystem* is defaulted to the primary coding system.

S.3.4 codeSystemName

We do not include *codeSystemName* because in **CWE**, the *codeSystemName* is defaulted to the primary coding system.

S.3.5 codeSystemVersion

We do not include *codeSystemVersion* because in our primary coding system, we do not use versioning.[1]

[1]Requires further research.

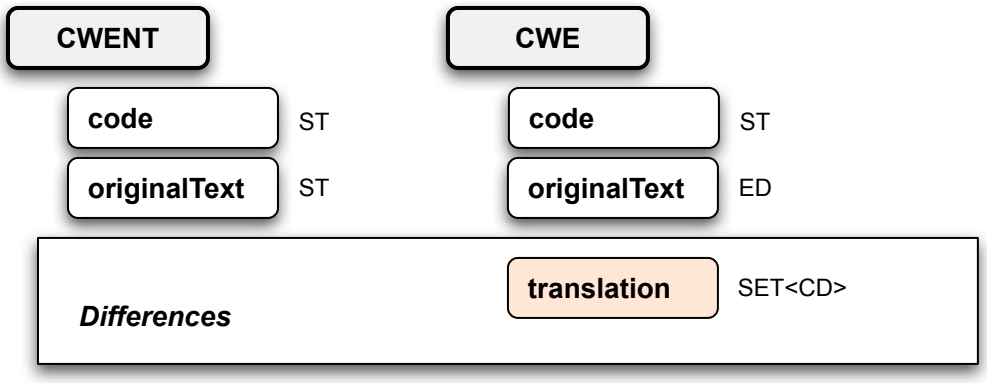


Figure S.3. CWENT to CWE Comparison

S.3.6 valueSet

We do not currently include the *valueSet*, which specifies the value set that applied when this CD was created.[2]

S.3.7 valueSetVersion

We do not currently include the *valueSetVersion*, which specifies the value set version that applied when this CD was created. Even if we decided to add the *valueSet* property, I believe the version could be handled by the vocabulary server.

S.3.8 originalText

We include the property *originalText*.

S.3.9 codingRationale

We do not include the property *codingRationale*. For better or worse, we assume it is original if there is not a translation.[3]

---

[2]We have never considered this. Requires further research.

[3]This implementation only uses part of the functionality that *codingRationale* provides, and we have not researched the other aspects.

S.3.10 translation

We do not include the property *translation*, which is the purpose of *CWENT*.

S.3.11 source

We do not include the property *source*. This property identifies the translation from which this was translated. We have not discussed this issue, but it seems this is implicit due to the fact we are only allowing one translation in the storage form. The question remains whether or not this is important transactionally when we allow more than one translation.

S.3.12 isCompositional

We do not include the property *isCompositional*. This can be derived from the vocabulary server.

S.3.13 equal

We do not include the property *equal*. This is a comparison operation that can be handled by the vocabulary server.

S.3.14 implies

We do not include the property *implies*. This is a comparison operation that can be handled by the vocabulary server.

S.3.15 Properties inherited from HL7:ANY

Please see Appendix B.

APPENDIX T

EDNT

Encapsulated Data No Thumbnail (**EDNT**), shown in Figure T.1, is used to convey any data from a plain character string, formatted text, to multimedia binary data. It may also contain formatted data from another standard such as XML. **EDNT** is not for use in Clinical Element models. It is only used internally in the **ED** data type, to represent the *thumbnail* property, and avoid recursion of the *thumbnail* property. The properties are described in Table T.1.

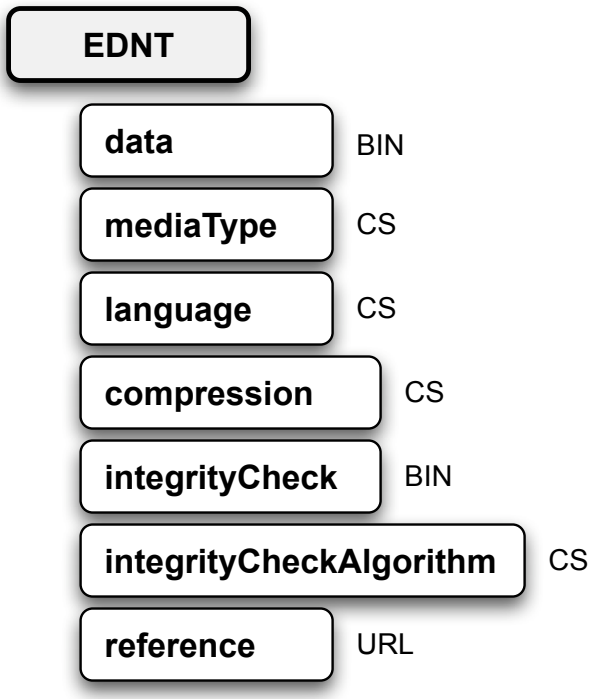


Figure T.1. Encapsulated Data - No Thumbnail.

Table T.1. EDNT Properties

Property	Type	Description
data	BIN	Binary data
mediaType	CS	Type of binary data
language	CS	Human Language
compression	CS	Compression Algorithm of binary data
integrityCheck	CS	Value generated by Integrity Check Algorithm
integrityCheckAlgorithm	CS	Algorithm used to generate value
reference	URL	Pointer to externally stored binary data

T.1 Properties

T.1.1 data

The property *data* is raw binary data. We have decided to store large binary data, such as X-Rays, in an external data store, but these data will still be present in the *data* property over the wire. The property *reference* will contain the pointer to the data in the external data store. For smaller binary data, they will actually be stored in the *data* property, and *reference* will be null. The services should function in a seamless manner, so that users of the service need not be aware whether the data were stored inline or as part of the external data store.

T.1.2 mediaType

The property *mediaType* identifies the type of the encapsulated data and identifies a method to interpret or render the data.

T.1.3 language

The property *language* is character-based information specifying the human language of the text.

T.1.4 compression

The property *compression* indicates whether the raw byte data is compressed, and what compression algorithm was used.

T.1.5 integrityCheck

The property *integrityCheck* is a short binary value representing a cryptographically strong checksum that is calculated over the binary data. The purpose of this property, when communicated with a reference, is for anyone to validate later whether the reference still resolves to the same data that the reference resolved to when the encapsulated data value with reference was created.

The integrity check is calculated over the raw binary data that are contained in the data component, or that are accessible through the reference. No transformations are made before the integrity check is calculated. If the data are compressed, the integrity check is calculated over the compressed data.

T.1.6 integrityCheckAlgorithm

The property *integrityCheckAlgorithm* specifies the algorithm used to compute the integrityCheck value. The cryptographically strong checksum algorithm Secure Hash Algorithm-1 (SHA-1) is currently the industry standard. It superseded the MD5 algorithm several years ago, when certain flaws in the security of MD5 were discovered. Currently, the SHA-1 hash algorithm is the default choice for the integrity check algorithm. Note that SHA-256 is also entering widespread usage.

T.1.7 reference

The property *reference* is a URL which will resolve to precisely the same binary data that could as well have been provided as online data. This serves as the pointer to the external data source where the data of **ED** are actually stored.

An IHE Profile will be used to define the format of URL. The IHE profile basically specifies a common URL format to use and calls for the use of a UID or OID to reference

the thing on the other end. The consistent URL format makes it possible to rewrite a stored URL to hit a different server using the same query at a later time.

T.2 HL7 Comparison

Comparison to **HL7:ED** is identical to the comparison of **ED** with **HL7:ED**, with the additional restriction that we have removed the *translation* property. The **HL7:ED** is shown in Figure T.2 and the comparison of **EDNT** to **HL7:ED** is shown in Figure T.3.

T.2.1 data

We include the property *data*.[1]

T.2.2 mediaType

We include the property *mediaType*.

[1]Requires further research. One ugly result is the problem with the path statement "data.ed.data." Should we change *data* to *value* which is used throughout most of our datatypes. Note that in **ST**, we are using *value*.

```
type EncapsulatedData alias ED specializes ANY {
   BIN     data;
   CS      mediaType;
   CS      charset;
   CS      language;
   CS      compression;
   TEL.URL reference;
   BIN     integrityCheck;
   CS      integrityCheckAlgorithm;
   ST      description;
   ED      thumbnail;
   DSET<ED> translation;
   INT     length;
   ED      subPart(INT start, INT end);
   BL      equal(ANY x);
};
```

Figure T.2. HL7:ED declaration.

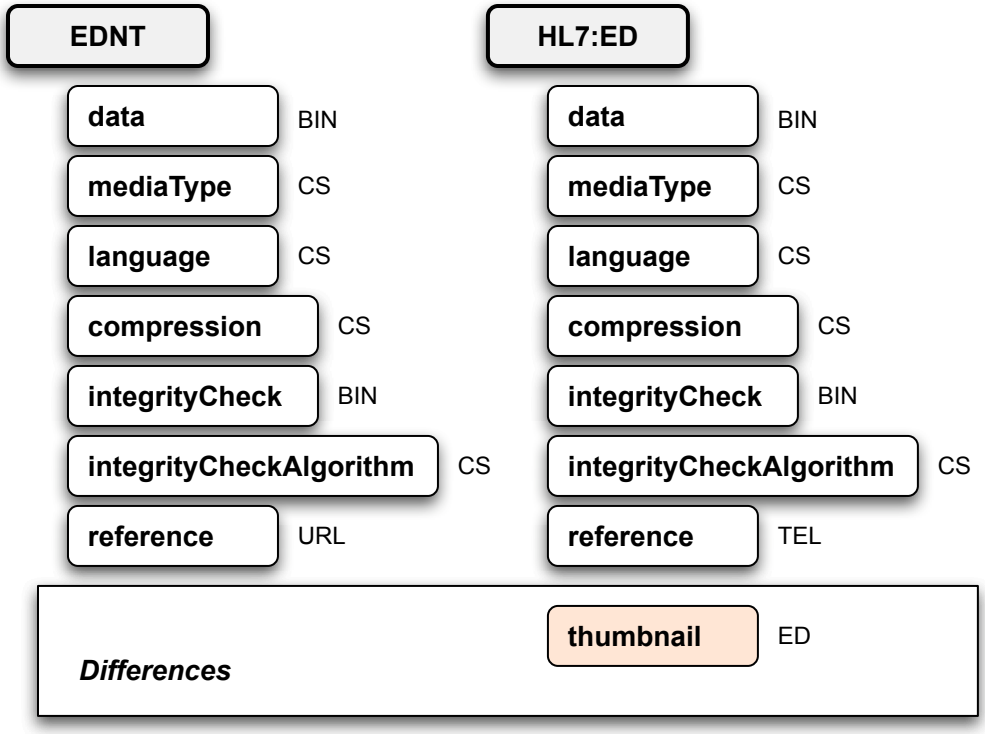


Figure T.3. ED to HL7:ED Comparison

T.2.3 charset

We do not include the property *charset* for character-based encoding.[2]

T.2.4 language

We include the property *language* which is used for character-based data.

T.2.5 compression

We include the property *compression* which is used to identify the compression algorithm used.

[2]Requires further research.

T.2.6 reference

We include the property *reference* to identify external references. HL7 uses **HL7:TEL** for the *reference* property, which we have substituted with **URL**.

T.2.7 integrityCheck

We include the property *integrityCheck* but currently have this defined as a type **CS** instead of a **BIN**. I think this may be an error.[3]

T.2.8 integrityCheckAlgorithm

We include the property *integrityCheckAlgorithm*.

T.2.9 description

We do not include the property *description*. This property is intended to be a short description of the media contained in case the media cannot be presented.[4]

T.2.10 thumbnail

We do not include the property *thumbnail*, which is the purpose of the datatype **EDNT**.

T.2.11 translation

We do not include the property *translation*, which allows for alternate renditions of the same content translated into a different language or a different mediaType.[5]

T.2.12 length

We do not include the property *length*, as the length of the binary data can be calculated by the implementation.

---

[3]Requires further research.

[4]Requires further research.

[5]Requires further research.

T.2.13 subpart

We do not include the property *subpart*, as this is an operator on the stored data.

T.2.14 equal

We do not include the property *equal*, as this is a comparison operator on the stored data.

APPENDIX U

BIN

Binary Data (**BIN**), shown in Figure U.1, is used to represent a raw block of bits. It is not used within Clinical Element models and is only used in the internal definition of the datatypes. The property is described in Table U.1.

U.1 Properties

U.1.1 value

The property *value* is a raw block of bits.

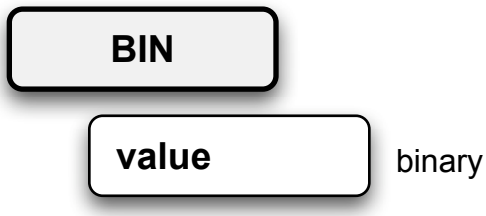


Figure U.1. Binary Data

Table U.1. BIN Properties

Property	Type	Description
value	binary	Raw block of bits

U.2 HL7 Comparison

There are no differences between **HL7:BIN** and our **BIN**. The **HL7:BIN** is shown in Figure U.2 and the comparison of **BIN** to **HL7:BIN** is shown in Figure U.3.

U.2.1 value

The property *value* is not part of the HL7 abstract definition. The HL7 specification leaves it up to implementation where to store the value of binary data. We have chosen to use *value*.

```
protected type BinaryData alias BIN specializes LIST<BN>;

type Sequence<T> alias LIST<T> specializes COLL<T> {
             T        head;
             LIST<T>  tail;
             BL       isEmpty;
             BL       notEmpty;
             T        item(INT index);
             BL       contains(T item);
             INT      length;
             LIST<T>  subList(start INT, end INT);
             LIST<T>  subList(start INT);
  literal    ST.SIMPLE;
  promotion  LIST<T>  (T x);
  demotion   BAG<T>;
};
```

Figure U.2. HL7:BIN declaration.

Figure U.3. BIN to HL7:BIN Comparison

APPENDIX V

CWE CASES

We have removed *codingRationale* from our coded types and physical quantity. We decided to only store one translation rather than a set, and due to this, we think we can remove *codingRationale* and implicitly know the *codingRationale* value.

While this is true for whether the datatype is original, it is not true for whether the datatype was postcoded, but we have decided to store postcoded information in the wrapper structure which contains a clinical element instance.

Prior to removing *codingRationale*, we investigated the possible use cases to analyze whether this made sense. The following sections walk through each of the use cases.

V.1 Application Provides Code

This example, shown in Figure V.1, has a source application sending a single code which is then stored. Upon retrieval, it can be deduced that since there are no translations, that this code is the original code. But if we ever drop translations during storage, the data upon retrieval will incorrectly appear to have been the original. We will assume we will never inappropriately drop a translation; thus, we do not need *codingRationale* for this use case.

V.2 Application Provides Code-OriginalText

This example, shown in Figure V.2, has a source application sending a code and originalText which is then stored. Upon retrieval, it can be deduced that since there are no translations, that this code is the original code. But if we ever drop translations during storage, the data upon retrieval will incorrectly appear to have been the original. We will assume we will never inappropriately drop a translation; thus, we do not need *codingRationale* for this use case.

```
Application provides ...

 CWE
    code            SER_NA_ECID
    originalText

It is stored in the repository as ...

 CWE
    code            SER_NA_ECID
    originalText
```

Figure V.1. Application provides code.

```
Application provides ...

 CWE
    code            SER_NA_ECID
    originalText    SERUM SODIUM

It is stored in the repository as ...

 CWE
    code            SER_NA_ECID
    originalText    SERUM SODIUM
```

Figure V.2. Application provides code-originalText.

V.3 Application Provides OriginalText

In the example shown in Figure V.3, the data came from an application, with no code because the user could not find an appropriate code. The SER_NA_ECID was coded after the fact by a human, based on what was in originalText. However, this looks just like the case above, where a code and an originalText both came in. So how do we indicate that in this case, SER_NA_ECID was coded after the fact – that it was not present in what was originally received?

The possible ways to represent this are listed as follows:

```
Application provides ...

 CWE
    code
    originalText    SERUM SODIUM

It is stored in the repository and then postcoded by a human ...

 CWE
    code            SER_NA_ECID
    originalText    SERUM SODIUM
```

Figure V.3. Application provides code-originalText.

1. Store coding rationale (HL7 has a "Postcoded" value for codingRationale which we could use).
2. Store one version of the data type instance with just original text, then store another version adding a code.
3. Use "translated" attribution in the Clinical Element wrapper to indicate that this instance was translated by a human.
4. A combination of the above.

We have decided to not use *codingRational* for this purpose. Instead, we will use a "translated" attribution in the Clinical Element wrapper to indicate that this instance was translated by a human. We most likely will also store the original message as well.[1]

V.4 Interface Provides Code

In the example shown in Figure V.4, the interface provides a single code, but based on sender-receiver agreement, we know the code system and version are ACME Coding Standard version 2.34.

If *originalText* is present in the message, we will use it in our **CWE** *originalText*. If there is no *originalText* in the message, then we will use the messages's *code* in the **CWE**

[1]This is a good idea in general. Even instances coming over the interface as HL7 messages could be stored as the original form.

```
Interface provides...
 CD
    code            SERUM_NA
    originalText

It is stored in the repository as...

 CWE
    code            SER_NA_ECID
    originalText    SERUM_NA
    translation
        CET
            code            SERUM_NA
            originalText
            cdSys           ACME_Code_STD
            cdSysVersion    2.34
```

Figure V.4. Interface provides code.

originalText. We will also be representing the messages's *code* in the **CET** translation, so this is redundant.

One interesting thing to note is that in this example, we implicitly know the coding system and version of the original message, yet we do not represent this. That is because we have decided that the translation is a copy of the message, and this implicit information was not in the message. Our requirement is that we store what came over the wire.

V.5 Interface Provides Code-OriginalText

In the example shown in Figure V.5, the interface provides a single code, but based on sender-receiver agreement, we know the code system and version are ACME Coding Standard version 2.34. According to the rule stated in Section V.4, the message's *originalText* is copied into *originalText* of the **CWE**. Again, we do not represent the implicit information we know about the coding system and version, because this did not come over the wire.

V.6 Interface Provides Code-Code System Info

In the example shown in Figure V.6, the interface provides a code plus code system information. The important thing to note in this example is in the translation where we store

```
Interface provides...

 CD
    code            SERUM_NA
    originalText    Serum Sodium

It is stored in the repository as...

 CWE
    code            SER_NA_ECID
    originalText    Serum Sodium
    translation
        CET
            code            SERUM_NA
            cdSys           ACME_Code_STD
            cdSysVersion        2.34
            originalText    Serum Sodium
```

Figure V.5. Interface provides code-originalText.

```
Interface provides...

 CD
    code            SERUM_NA
    cdSys           ACME_Coding
    cdSysName       ACME Coding Standard
    cdSysVersion    2.34
    originalText

It is stored in the repository as...

 CWE
    code            SER_NA_ECID
    originalText    SERUM_NA
    translation
        CET
            code            SERUM_NA
            cdSys           ACME_Coding
            cdSysName       ACME Coding Standard
            cdSysVersion    2.34
            originalText
```

Figure V.6. Interface provides code-code system info.

what came over the wire. For the coding system, we do not translate the code "ACME_ Coding" into a code such as "ACME_Coding_ECID". Instead, we store exactly what we received.

V.7 Interface Provides Code-Translation

In the example shown in Figure V.7, the interface provides a code plus a translation. Upon storage, we are losing one of the translations in the message. The interface team would decide on a case-by-case basis which translation to keep. However, in this example, the true original is being thrown away. Yet our implicit rules state it is original.[2]

V.8 Unknown Coding System

In the example shown in Figure V.8, the interface provides a code from an unknown coding system. According to the rule stated in Section V.4, the message's *orignalText* are copied directly into the **CWE**. The *code* is unable to be translated, so it is ignored in the **CWE**. The *translation* will store all of the message's parts, including the code that could not be translated.[3]

---

[2]We have never discussed this fact, and what is meant by "original". Also, this is another case for always saving the message and associating it with the resulting translation.

[3]In an HL7 v3 message, we know that either original text or code or both will be present. In non-HL7 v3 messages, we assume at least a code will be present.

```
Interface provides...

 CD
    code            SERUM_NA
    cdSys           ACME_Coding
    cdSysName       ACME Coding Standard
    cdSysVersion    2.34
    originalText
    translation
        CD
            code            SERUM_SOD
            cdSys           3M_Coding
            cdSysName       3M_Coding Standard
            cdSysVersion    1.56
            originalText
            codingRation    Original

It is stored in the repository as...

 CWE
    code            SER_NA_ECID
    originalText    SERUM_NA
    translation
        CET
            code            SERUM_NA
            cdSys           ACME_Coding
            cdSysName       ACME Coding Standard
            cdSysVersion    2.34
            originalText
```

Figure V.7. Interface provides code-translation.

```
Interface provides...

 CD
    code            123456
    code system     Bob's CodeSystem
    originalText    SERUM SODIUM

It is stored in the repository as...

 CWE
    code
    originalText    SERUM SODIUM
    translation
        CET
            code            123456
            displayName
            cdSys           Bob's CodeSystem
            cdSysName
            cdSysVersion
            originalText    SERUM SODIUM
            codingRation
            source
```

Figure V.8. Interface provides code in unknown coding system.

APPENDIX W

PQ CASES

We have removed *codingRationale* out of both our coded types and physical quantity. We decided to only store one translation rather than a set, and due to this, we think we can remove *codingRationale* and implicitly know the *codingRationale* value.

While this is true for whether the datatype is original, it is not true for whether the datatype was postcoded, but we have decided to store postcoded information in the wrapper structure which contains a clinical element instance.

Prior to removing *codingRationale*, we investigated the possible use cases to analyze whether this made sense. The following sections walk through each of the use cases.

W.1 Application Provides Unit Code

In the example shown in Figure W.1, an application provides a unit code which happens to be the normal value so it is stored in the **PQ** section. Since there is no translation, we also know that this is the original.

W.2 Application Provides Unit Code-OriginalText

In the example shown in Figure W.2, an application has provided the normal value so it is stored in the **PQ** section. Since there is no translation, we also know that this is the original.

W.3 Application Provides Unit OriginalText

In the example shown in Figure W.3, an application has provided the unit as originalText. The mmHg_ECID was coded after the fact by a human, based on what was in *originalText*. However, this looks just like what was stored in section W.2, with a *code* and

```
Application provides...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText

It is stored in the repository as...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText
```

Figure W.1. Application provides unit code.

```
Application provides...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText    mmHg

It is stored in the repository as...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText    mmHg
```

Figure W.2. Application provides unit code-originalText.

```
Application provides...

 PQ
    value       120
    unit
        CWENT
            code
            originalText    mmHg

It is stored in the repository and then postcoded by a human ...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText    mmHg
```

Figure W.3. Application provides unit originalText.

an *originalText*. So how do we indicate that in this case, mmHg_ECID was coded after the fact – that it was not present in what was originally received?

The possible ways to represent this are listed as follows:

1. Store coding rationale (HL7 has a "Postcode" value for codingRationale which we could use).
2. Store one version of the data type instance with just original text, then store another version adding a code.
3. Use "translated" attribution in the Clinical Element wrapper to indicate that this instance was translated by a human.
4. A combination of the above.

We have decided to not use *codingRational* for this purpose. Instead, we will use a "translated" attribution in the Clinical Element wrapper to indicate that this instance was translated by a human. We most likely will also store the original message as well.[1]

[1]This is a good idea in general. Even instances coming over the interface as HL7 messages could be stored as the original form.

W.4 Interface Provides Unit Code

In the example shown in Figure W.4, an interface has provided the unit as a code. The original **PQ** that came over the interface is actually the normal but the unit is from a different coding system. We still create a translation and put the original **PQ** into the **PQT**. There is one big difference here compared to how we handled **CWE**, and that is we add the code system as an ECID code. Thus, the translation is not exactly what came over the wire. We do this because we need to quickly know the coding system so that we can do more translations from **PQT** to other possible units.[2]

[2]This is another case which shows we should save what came over the wire.

```
Interface provides...

 PQ
    value       120
    unit
        CS
            code            mmHg_UCUM

It is stored in the repository as...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText    mmHg_UCUM
    translation
         PQT
            value           120
            unit            mmHg_UCUM
            codeSys         UCUM_ECID
```

Figure W.4. Interface provides unit code.

W.5 Interface Provides Unit Code - Needs Normalization

In the example shown in Figure W.5, an interface has provided the unit as a code that is not the designated normal unit. We know the upper portion is normal, and we know it is not original because of translation. The only difference from this and the previous example is that we do not put the code that came over the wire in *originalText*. This is because that unit is not a surface form of the "mmHg_ECID" code. So, the rule is that we only copy the unit surface form when we are not normalizing to a new unit.

W.6 Interface Provides Unit Code from Unknown Coding System

In the example shown in Figure W.6, an interface has provided a unit code from an unknown coding system. This slightly breaks the rule we have stated that the normal unit is in the **PQ** section, so we must expand that rule to state that if there is a *code* in the **PQ** section, then it is the normal. We also add an additional translation which is identical to the

```
Interface provides...
 PQ
    value       120
    unit
        CS
            code            torr_UCUM

It is stored in the repository as...

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText
    translation
         PQT
            value           120
            unit            torr_UCUM
            codeSys         UCUM_ECID
```

Figure W.5. Interface provides unit code - needs normalization.

```
Interface provides...

 PQ
    value       120
    unit
        CS
            code           mmHg_BobsUnits

It is stored in the repository as...

 PQ
    value       120
    unit
        CWENT
            code
            originalText    mmHg_BobsUnits
    translation
         PQT
            value          120
            unit           mmHg_BobsUnits
            codeSys
```

Figure W.6. Interface provides unit code from unknown coding system.

PQ section. This is because we may eventually postcode the **PQ** and then we could not tell it was original.[3]

W.7 Interface Provides Unit Code and Normalized Translation

In the example shown in Figure W.7, an interface has provided a unit code and a normalized translation. Our system will only keep one of the units that came over the interface. In the **CWE**, case we left this up to the interface team, but in the **PQ** case, it is always the original that is retained. This is due to rounding errors in the value that occur with every calculated translation. So basically, we treat this use case just like we received the translation, which makes it a case we have seen previously. We are losing data that came

[3]I think we may not need the translation until we postcode.

```
Interface provides...

 PQ
    value       .157
    unit
        CS
            code            atm_UCUM
    translation
         PQR
            value           120
            unit            mmHg_UCUM
            codeSys         UCUM
            codingRat       original

It is stored in the repository as:

 PQ
    value       120
    unit
        CWENT
            code            mmHg_ECID
            originalText    mmHg_UCUM
    translation
         PQT
            value           120
            unit            mmHg_UCUM
            codeSys         UCUM_ECID
```

Figure W.7. Interface provides unit code and normalized translation

over the wire, so this is another case which shows we should save the original message in some manner.

APPENDIX X

CEML EXAMPLES

The following figures contain actual authoring CEML definitions. Figure X.1 is model of attribution data. Figure X.2 is a model of a vital signs panel. Figure X.3 is a model of a diastolic blood pressure. Figure X.4 is a model of a wound closure procedure. Figure X.5 is a model of a lab order.

```
<ceml>
<cetype name="Attribution" kind="noninstantiable">
        <key code="Action_KEY_ECID"/>
        <data type="cwe" domain="Attribution_DOMAIN_ECID"/>
        <qual name="startTime" type="StartTime" card="0-1"/>
        <qual name="endTime" type="EndTime" card="0-1"/>
        <qual name="participant" type="Participant" card="0-M"/>
        <qual name="patientLocation" type="PatientLocation" card="0-1"/>
        <qual name="providerLocation" type="ProviderLocation" card="0-1"/>
        <qual name="reason" type="Reason" card="0-M"/>
        <qual name="actionMethod" type="ActionMethod" card="0-1"/>
</cetype>
</ceml>
```

Figure X.1. The ceml definition of the type Attribution

```
<ceml>
<cetype name="VitalSignPanel" kind="panel">
        <key code="VitalSignPanel_KEY_ECID"/>
        <item name="bloodPressurePanel" type="BloodPressurePanel" card="0-1"/>
        <item name="bodyTemperatureMeas"
                    type="BodyTemperatureMeas" card="0-1"/>
        <item name="heartRateMeas" type="HeartRateMeas" card="1"/>
        <item name="respiratoryRateMeas" type="RespiratoryRateMeas" card="1"/>
        <item name="oxygenSaturationMeas" type="OxygenSaturationMeas" card="1"/>
        <qual name="relativeTemporalContext"
                    type="RelativeTemporalContext" card="0-M"/>
        <qual name="patientPrecondition"
                    type="PatientPrecondition" card="0-M"/>
        <mod name="subject" type="Subject" card="0-1"/>
        <att name="observed" type="Observed" card="0-1"/>
        <att name="reportedReceived" type="ReportedReceived" card="0-1"/>
        <att name="verified" type="Verified" card="0-1"/>
</cetype>
</ceml>
```

Figure X.2. The ceml definition of the type VitalSignPanel

```
<ceml>
<cetype name="DiastolicBloodPressureMeas" kind="statement">
    <key code="DiastolicBloodPressure_KEY_ECID"/>
    <data type="pq"/>
    <qual name="methodDevice" type="MethodDevice" card="0-1"/>
    <qual name="bodyLocationPrecoord"
                type="BodyLocationPrecoord" card="0-1"/>
    <qual name="bodyPosition" type="BodyPosition" card="0-1"/>
    <qual name="abnormalFlag" type="AbnormalFlag" card="0-1"/>
    <qual name="deltaFlag" type="DeltaFlag" card="0-1"/>
    <qual name="referenceRangeNar" type="ReferenceRangeNar" card="0-1"/>
    <qual name="relativeTemporalContext"
                type="RelativeTemporalContext" card="0-M"/>
    <mod name="subject" type="Subject" card="0-1"/>
    <att name="observed" type="Observed" card="0-1"/>
    <att name="reportedReceived" type="ReportedReceived" card="0-1"/>
    <att name="verified" type="Verified" card="0-1"/>
    <constraint path="qual.abnormalFlag.data.cwe.domain"
                        value="AbnormalFlagNumericNom_DOMAIN_ECID"/>
    <constraint path="qual.deltaFlag.data.cwe.domain"
                        value="DeltaFlagNumericNom_DOMAIN_ECID"/>
    <constraint path="data.pq.unit.domain"
                value="PressureUnits_DOMAIN_ECID"/>
    <constraint path="data.pq.normal" value="MilliMetersOfMercury_ECID"/>
    <constraint path="qual.methodDevice.data.cwe.domain"
                        value="BloodPressureMeasurementDevice_DOMAIN_ECID"/>
    <link name="hasPrecondition" relation="hasPrecondition_ECID" card="0-M">
        <target path="type.domain" value="PreconditionTypes_DOMAIN_ECID"/>
    </link>
</cetype>
</ceml>
```

Figure X.3. The ceml definition of the type DiastolicBloodPressureMeas

```
<ceml>
<cetype name="WoundClosureProc" kind="statement">
        <key code="Procedure_KEY_ECID" />
        <data type="cwe" code="WoundClosure_ECID" />
        <qual name="directSite" type="DirectSite" card="0-1"/>
        <qual name="bodyLocation" type="BodyLocation" card="0-1"/>
        <qual name="routeMethodDevice" type="RouteMethodDevice" card="0-1"/>
        <qual name="patientResponse" type="PatientResponse" card="0-1"/>
        <qual name="relativeTemporalContext"
                    type="RelativeTemporalContext" card="0-M"/>
        <qual name="aggregate" type="Aggregate" card="0-1"/>
    <mod name="subject" type="Subject" card="0-1"/>
        <mod name="negationInd" type="NegationInd" card="0-1"/>
        <att name="observed" type="Observed" card="0-1"/>
        <att name="performed" type="Performed" card="0-M"/>
        <att name="reportedReceived" type="ReportedReceived" card="0-1"/>
        <att name="verified" type="Verified" card="0-1"/>
        <constraint path="qual.routeMethodDevice.data.cwe.domain"
                value="WoundClosureRouteMethodDevice_DOMAIN_ECID" />
</cetype>
</ceml>
```

Figure X.4. The ceml definition of the type WoundClosureProc

```
<ceml>
<cetype name="OrderLab" base="Order" kind="statement">
    <key code="OrderLab_KEY_ECID"/>
    <constraint path="qual.labelInstruction.card" value="0"/>
    <constraint path="qual.administrationInstruction.card" value="0"/>
    <constraint path="item.orderable.qual.routeMethodDevice.card"
                value="0"/>
    <constraint path="item.orderable.qual.totalVolume.card" value="0"/>
    <constraint path="item.orderable.qual.volumeRate.card" value="0"/>
    <constraint path="item.orderable.qual.prnInd.card" value="0"/>
    <constraint path="item.orderable.qual.administrationPeriod.card"
                value="0"/>
    <constraint path="item.orderable.qual.dispenseQuantity.card"
                value="0"/>
    <constraint path="item.orderable.qual.refills.card" value="0"/>
    <constraint path="item.orderable.qual.transportMode.card" value="0"/>
    <constraint path="item.orderable.qual.adminSite.card" value="0"/>
    <constraint path="item.orderable.qual.device.card" value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.formulation.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.specialDoseLowerLimit.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.specialDoseUpperLimit.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.orderedDoseLowerLimit.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.orderedDoseUpperLimit.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.ivComponentType.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.activeIngredient.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.filledSubstance.card"
      value="0"/>
    <constraint
      path="item.orderable.item.orderableItem.qual.substitutionStatus.card"
      value="0"/>
</cetype>
</ceml>
```

Figure X.5. The ceml definition of the type OrderLab

APPENDIX Y

GLOSSARY

Y.1 Definitions

Abstract Syntax Notation One A joint ISO/IEC and ITU-T standard language for platform independent data structure modeling and the encoding rules to create serialized data instances.

Authoring CEML See **CEML**

ASN.1 See **See Abstract Syntax Notation One**

Attribution Attribution is used to define an action along with the details of that action, including the who, what, where, when, and why the action was performed. This is more commonly referred to now as **Provenance**.

Clinical Element Abstract Instance Model The Clinical Element Abstract Instance Model defines a recursive structure that can hold patient instance data. It is defined abstractly, and thus this model must be implemented in an actual language such as XML or Java.

Clincial Element Abstract Constraint Model The Clinical Element Abstract Constraint Model is the model used to constrain and describe allowable instances of patient data. This is the abstract description of our constraint formalism, which is then actually implemented. We have an XML implementation which is called CEML or Clinical Element Modeling Language.

<cetype/> A CEML construct. The element which is used to define a Clinical Element Constraint Type, or CEType.

CE See **Clinical Element**. This should not be confused with **HL7:CE** which is a now deprecated HL7 version 3 datatype Coded With Equivalents.

CEM See **Clinical Element Model**.

CEML See **Clinical Element Modeling Language**.

CEMorph A Clinical Element Morph is a constraint model describing data instances intended for transient use.

CEO See **Clinical Element Object**.

CETL See **Clinical Element Transformation Language**

CEType See **Clinical Element Constraint Type**

Clinical Element The term which unfortunately is used rather loosely and incorrectly. The correct usage is that a Clinical Element is the recursive structure in the Abstract Instance Model, also called a Clinical Element Instance Node. It is NOT a Clinical Element Constraint Type.

Clinical Element Instance Node A Clinical Element Instance Node is the recursive structure in the Abstract Instance Model. It consists of a key, a type, a value choice (data or items), modifiers, and qualifiers.

Clinical Element Constraint Type This is a construct defined by the Abstract Constraint Model which is a collection of constraints used to validate an instance from the Abstract Instance Model. The definition of a particular Clinical Element Constraint Type such as a BloodPressurePanel. Instances of this BloodPressurePanel must conform to the constraints stated within this constraint definition.

Clinical Element Instance An instance of patient data in a format that conforms to the Clinical Element Abstract Instance Model. This instance data can then be constrained by a Clinical Element Constraint Type. For example, an instance of patient data such as a hematocrit with a value of 38.9 stored in the EMR for John Doe, that conforms to the constraint type called LabObservationHematocrit.

Clinical Element Model (CEM) Unfortunately, this term is erroneously used to refer to an individual Clinical Element Constraint Type (CEType).

The Clinical Element Model (The CEM) **The Clinical Element Model** is a term used to denote the global modeling effort as a whole. It is the combination of the Abstract Instance Model and the Abstract Constraint Model.

Clinical Element Modeling Language (CEML) This is our Implementation Technology Specification of the Abstract Constraint Model. It is an XML-based syntax. We have 2 forms: a Strict CEML and Authoring CEML. Strict CEML follows the Abstraint Constraint Model constructs exactly. Authoring CEML includes the syntax of Strict CEML, but also adds shortcut or macro elements to make the definitions more succinct.

Clinical Element Transformation Language An XML syntax which describes the transformation rules needed to transform a source data instance into a destination data instance.

Clinical Element Object (CEO) This is a programatic object that is used to manipulate Clinical Element Instance Data. It is analogous to an XML DOM.

Compound Statement A **Statement** whose meaning is conveyed by multiple clinical values, with associated modifiers and qualifiers. The meaning of the Compound Statement is dependent on a set of elements with values being interpreted together within the context of the collection. In The Clinical Element Model, A Compound Statement has the value choice of items rather than data. For instance, a pharmacy order is a compound statement. See **Statement**.

Component A CE Constraint Type that is only used within another CE Constraint Type (as an item, qualifier, or modifier). A CE Instance that conforms to a component **CEType** can NOT be stored in the patient EMR on its own, but only as an internal part of another instance. Examples of Component **CETypes** include Specimen, MethodAndDevice, BodyPosition, and Length.

Data A Construct of the Abstract Instance Model which contains an HL7-derived data type that serves as the "value" of a Simple Statement.

DCM See **Detailed Clinical Model**

Detailed Clinical Model A structured data model that describes a clinical concept or collection of clinical concepts.

ECID The unique code for a given concept in the ECID terminology server.

ECIS The next-generation patient information system being developed jointly by General Electric (GE) and Intermountain Healthcare. ECIS is an acroynm for "electronic clinical information system."

Explicit Transformation An explicit transformation contains everything needed for the transformation represented in the rules of the transformation. No external information is needed.

<item/> An Authoring CEML construct used to constrain Clinical Element Instance Nodes within the **Items** of a Clinical Element Instance.

Key An property within a Clinical Element Instance that is an **HL7:CWE**. The key's code links the Clinical Element instance to a real-world coding system.

<key/> An Authoring CEML construct used to constrain the **Key** of a Clinical Element Intstance.

Label DEPRECATED : These are now replaced by Semantic Links with a target of a Coded Concept.

<mod/> An Authoring CEML construct used to constrain Clinical Element Instance Nodes within the **Mods** of a Clinical Element Instance.

Modifier A Clinical Element Instance node which modifies the content of the Value Choice in the containing Clinical Element Instance. The extent of this modification is so great that the value choice can never be considered independently without simultaneously considering the effect of the modifier on the value choice.

noninstantiable noninstantiable is a potential value of the property "kind" in a Clinical Element Constraint Type that indicates that no patient data can be instantiated using this constraint type. Instead, this constraint type is used as a starting point to define other constraint types. A Noninstantiable Clinical Element Constraint Type is a type that does not contain enough information (items, modifiers, qualifiers, etc.) to be instantiated.

Panel Represents a common grouping of clinical observations. A chem7 lab result is an example of a common lab panel. A panel is a collection of statements that can exist independently. Synonyms used for panel include battery and collection.

Provenance Provenance of a resource is a record that describes entities and processes involved in producing and delivering or otherwise influencing that resource. Provenance provides a critical foundation for assessing authenticity, enabling trust, and allowing reproducibility. Provenance assertions are a form of contextual metadata and can themselves become important records with their own provenance [25].

Qualifier Clinical Element Instance node which gives more information about the Value Choice in the containing Clinical Element Instance. The degree to which this qualification changes the meaning of the value choice varies, but it is never to the degree of a modifier. In medical informatics circles, some argue you can never even truly ignore a qualifier, so why make the distinction between a qualifier and a modifier.

<qual/> An Authoring CEML construct used to constrain Clinical Element Instance Nodes within the **Quals** of a Clinical Element Instance.

Semantic Link Semantic Link is the construct that is used to establish a relationship between separate independent Clinical Element Instances. The semantic link specifies a coded relationship between CE instances; they are not used for static, a priori relationships.

For example, a semantic link may appropriately be used to express the relationship between a particular medication order for Tom and the reason for the order by ref-

erence to a specific problem in Tom's problem list as only that particular order is related to that particular problem.

An inappropriate use of a semantic link would be to express the relationship between a lab observation and a specimen as all lab observations are related to specimens. This type of static relationship is best defined within the lab observation CE Constraint Type.

Semi-Explicit Transformation Semi-explicit transformation uses external information from the terminology server in addition to the rules in the transformation map.

Simple Statement A **Statement** whose meaning is conveyed by a single clinical value, with associated modifiers and qualifiers. In The Clinical Element Model, A Simple Statement has the value choice of data rather than items. An example of a simple statement is a hematocrit lab result. See **Statement**.

Specializable CEType A Specializable Clinical Element Constraint Type does not specify one particular value to be used as its Key.code. Instead, the Constraint Type specifies a domain of permitted Key.codes.

For example, quantitative lab result is a Specializable CE Constraint Type, in which the Type is quantitative lab result, and the Key is constrained to the domain of specific quantitative labs like hct, serum sodium, etc.

Specific CEType A Specific Clinical Element Constraint Type specifies a single code as the permissible value of its Key.code.

LabObservationHematocrit is an example of a Specific Clinical Element Constraint Type. Its parent type is the Specializable CEType LabObservationQuantitative. The LabObservationHematocrit CE's Type is LabObservationHematocrit, and the only Key value allowed is Hct.

A Specializable Clinical Element Constraint Type may be used to validate patient data where no more specific model is available, but the ultimate goal is to have a Specific Clinical Element Constraint Type for everything stored in a patient record.

Statement A complete assertion about a particular aspect, characteristic, or condition of a patient. A statement contains a value choice (data or items), and may also contain modifiers and qualifiers. The parts of a Statement are not meaningful by themselves out of context of the Statement. There are two types of statements: simple statements and compound statements.

Strict CEML See **CEML**

REFERENCES

[1] Goossen W. T., Goossen-Baremans A., van der Zel M.. Detailed Clinical Models: A Review *Healthc Inform Res*. 2010;16:201-214.

[2] Gardner R. M., Pryor T. A., Warner H. R.. The HELP hospital information system: update 1998 *Int J Med Inform*. 1999;54:169-82.

[3] Huff S. M., Rocha R. A., Bray B. E., Warner H. R., Haug P. J.. An event model of medical information representation *J Am Med Inform Assoc*. 1995;2:116-34.

[4] Huff S. M., Rocha R. A., Solbrig H. R., Barnes M. W., Schrank S. P., Smith M.. Linking a medical vocabulary to a clinical data model using Abstract Syntax Notation 1 *Methods Inf Med*. 1998;37:440-52.

[5] ITU-T . Rec. X.680 | ISO/IEC 8824-1 (Specification of basic notation) 2002.

[6] ITU-T . Rec. X.682 | ISO/IEC 8824-3 (Constraint specification) 2002.

[7] World Wide Web Consortium (W3C) . XML Schema. Part 1: Structures, W3C Recommendation 2001.

[8] World Wide Web Consortium (W3C) . XML Schema. Part 2: Datatypes, W3C Recommendation 2001.

[9] ITU-T . Rec. X.690 | ISO/IEC 8825-1 (BER, CER and DER) 2002.

[10] Huff S. M., Hammond W. E., Williams W. G.. Clinical information interchange with Health Level Seven in *Cancer Informatics: Essential Technologies for Clinical Trials*:176-193New York: Springer 2002.

[11] CDISC . Operational Data Model (ODM) Version 1.1 - Final 2002.

[12] Nadkarni P. M., Marenco L., Chen R., Skoufos E., Shepherd G., Miller P.. Organization of heterogeneous scientific data using the EAV/CR representation *J Am Med Inform Assoc*. 1999;6:478-493.

[13] Health Level Seven . Reference Information Model 2002.

[14] DICOM . Part 1: Introduction and Overview 2001.

[15] Huff S. M., Rocha R. A., McDonald C. J., et al. Development of the Logical Observation Identifier Names and Codes (LOINC) vocabulary *J Am Med Inform Assoc*. 1998;5:276-92.

[16] Health Level Seven . Arden Syntax for Medical Logic Systems 1999.

[17] Pryor T. A., Hripcsak G.. Sharing NLMs: an experiment between Columbia-Presbyterian and LDS hospital in *17th Symposium on Computer Applications in Medical Care,*(Washington, DC) 1993.

[18] CEN/TC251 . prENV 13606-1, Health informatics, Electronic healthcare record communication, Part 1: Extended Architecture 2000.

[19] CEN/TC251 . prENV 13606-2, Health informatics, Electronic healthcare record communication, Part 2: Domain Term List 2000.

[20] Beale T.. Archetypes: Constraint-based Domain Models for Future-proof Information Systems 2002.

[21] Rector A. L., Zanstra P. E., Solomon W. D., et al. Reconciling usersâĂŹ needs and formal requirements: issues in developing a reusable ontology for medicine *IEEE Trans. Inf. Technol. Biomed.* 1998;2:229-242.

[22] Johnson S. B.. Generic data modeling for clinical repositories *J Am Med Inform Assoc.* 1996;3:328-39.

[23] Health Level Seven . Version 3 Data Types 2002.

[24] Health Level Seven . v3 Data Types, Implementable Technology Specification for XML 2002.

[25] World Wide Web Consortium (W3C) . What is Provenance? 2005.

[26] Coyle J. F., Mori A. R., Huff S. M.. Standards for detailed clinical models as the basis for medical data exchange and decision support *Int J Med Inform.* 2003;69:157-74.

[27] Dolin R. H., Alschuler L., Beebe C., et al. The HL7 Clinical Document Architecture *J Am Med Inform Assoc.* 2001;8:552-69.

[28] Health Level Seven . HL7 Version 3 Standard: Core Principles and Properties of Version 3 Models 2010.

[29] Blobel B., Stassinopoulos G., Pharow P.. Model-based design and implementation of secure, interoperable EHR systems *AMIA Annu Symp Proc.* 2003:96-100.

[30] Rocha R. A., Huff S. M., Haug P. J., Warner H. R.. Designing a controlled medical vocabulary server: the VOSER project *Comput Biomed Res.* 1994;27:472-507.

[31] Cimino J. J.. Desiderata for controlled medical vocabularies in the twenty-first century *Methods Inf Med.* 1998;37:394-403.

[32] Friedman C., Hripcsak G., Johnson S. B., Cimino J. J., Clayton P. D.. A Generalized Relational Schema for an Integrated Clinical Patient Database *Proc Annu Symp Comput Appl Med Care.* 1990:335-339.

[33] Blobel B.. Advanced and secure architectural EHR approaches *Int J Med Inform.* 2006;75:185-90.

[34] Rogers J., Rector A.. The GALEN ontology. *In: Medical Informatics Europe (MIE 96). Copenhagen: IOS Press*. 1996:174-178.

[35] Sundvall E., Qamar R., Nystrom M., et al. Integration of tools for binding archetypes to SNOMED CT *BMC Med Inform Decis Mak*. 2008;8 Suppl 1:S7.

[36] Qamar R., Kola J., Rector A. L.. Unambiguous data modeling to ensure higher accuracy term binding to clinical terminologies *AMIA Annu Symp Proc*. 2007:608-13.

[37] Rector A. L.. Clinical terminology: why is it so hard? *Methods Inf Med*. 1999;38:239-52.

[38] Rector A. L., Nowlan W. A., Kay S., Goble C. A., Howkins T. J.. A framework for modelling the electronic medical record *Methods Inf Med*. 1993;32:109-19.

[39] Munoz A., Somolinos R., Pascual M., et al. Proof-of-concept design and development of an EN13606-based electronic health care record service *J Am Med Inform Assoc*. 2007;14:118-29.

[40] Ceusters W., Smith B.. Strategies for referent tracking in electronic health records *J Biomed Inform*. 2006;39:362-78.

[41] Aliferis C. F., Cooper G. F.. Temporal representation design principles: an assessment in the domain of liver transplantation *Proc AMIA Symp*. 1998:170-4.

[42] Campbell K. E., Das A. K., Musen M. A.. A logical foundation for representation of clinical data *J Am Med Inform Assoc*. 1994;1:218-32.

[43] Dolin R. H.. Modeling the temporal complexities of symptoms *J Am Med Inform Assoc*. 1995;2:323-31.

[44] Aliferis C. F., Cooper G. F.. A new formalism for temporal modeling in medical decision-support systems *Proc Annu Symp Comput Appl Med Care*. 1995:213-7.

[45] Parker C. G., Rocha R. A., Campbell J. R., Tu S. W., Huff S. M.. Detailed clinical models for sharable, executable guidelines *Stud Health Technol Inform*. 2004;107:145-8.

[46] Garde S., Knaup P., Hovenga E., Heard S.. Towards semantic interoperability for electronic health records *Methods Inf Med*. 2007;46:332-43.

[47] Huff S. M., Rocha R. A., Coyle J. F., Narus S. P.. Integrating detailed clinical models into application development tools *Stud Health Technol Inform*. 2004;107:1058-62.

[48] Johnson S. B., Hripcsak G., Chen J., Clayton P.. Accessing the Columbia Clinical Repository *Proc Annu Symp Comput Appl Med Care*. 1994:281-5.

[49] Huff S. M., Haug P. J., Stevens L. E., Dupont R. C., Pryor T. A.. HELP the next generation: a new client-server architecture. *Proc Annu Symp Comput Appl Med Care*. 1994:271-275.

[50] Health Level Seven . HL7 Version 3 Standard: Data Types - Abstract Specification, Release 2 Last Ballot: Normative Ballot 4 2009.

[51] OpenEHR . Clinical Content Models 2011.

[52] Health Level Seven . OpenEHR datatypes mapping 2007.