

Archetypes

Constraint-based Domain Models for Future-proof Information Systems

Keywords: archetypes, domain modelling, software engineering, health

Authors: Thomas Beale

Revision: 2.2.1

Pages: 69

Copyright © 2000, 2001

Thomas Beale

email: thomas@deephought.com.au **web:** www.deephought.com.au

Amendment Record

Issue	Details	Who	Date
1.1 draft A	Initial Writing	T Beale	28 Nov 2000
1.1 draft B	Added introductory material; more examples based on data structures; GEHR links.	T Beale	13 Dec 2000
1.1 draft C	Further improvements in methodology description.	T Beale	20 Dec 2000
1.1 draft D	Minor changes.	T Beale	26 Dec 2000
1.2 draft A	Significant updates due to conversations at the Orlando working group meeting of HL7, Jan 2001.	T Beale	20 Jan 2001
1.2 draft B	Pre-review modifications.	T Beale	2 Feb 2001
1.2 draft C	Changes from Europe trip Feb 2001. Refined validation ideas, added archetype model diagram. Changed use of term "KR" model to "Reference Object Model" to avoid confusion with AI uses.	T Beale	20 Feb 2001
2.1 draft A	Major additions to model description, technical detail.	T Beale	22 Mar 2001
2.1 draft B	Post Brazil trip additions	T Beale	20 May 2001
2.2	Major additions and rework of "Formal Model" section. Document split into theory and methodology documents.	T Beale	14 Aug 2001
2.2.1	Corrections to "Formal.Model" section.	T Beale	20 Aug 2001

Acknowledgements

Thanks to...

Dr Sam Heard, with whom I first worked on the original GEHR (Good *European* Health Record) in 1994, and the clinical lead of the current GEHR (Good *Electronic* Health Record; Australia) project, has provided the vision for GEHR, and a wealth of domain knowledge and technical inspiration for the current formulation of the archetype concept (in fact, we thought of the concept as we walked to lunch one day along a shady street near my house, and he described how he thought degrees of freedom could be added to clinical concept models....)

Dr Peter Schloeffel and Dr David Rowed, both also involved in the Australian GEHR project, as well as various medical standards developments including ISO and HL7 have also provided numerous insights into the medical informatics domain for GEHR.

Members of the “Titanium” group at the DSTC (Distributed Systems Technology Centre; University of Qld, Australia): Linda Bird, Andrew Goodchild and Zar Zar Tun, for numerous technical discussions on the use of XML, as well as ongoing development work based on GEHR and XML.

Discussions with many people involved in the CEN EHCRA and HL7v3 clinical messaging standards have provided better insights into the medical domain, and the use of formal knowledge models such as archetypes. In particular: David Markwell, M.D. (UK), Angelo Rossi-Mori (Istituto Tecnologie Biomediche, CNR, Italy), Ken Rubin (EDS), Martin Kernberg, M.D. (US), Gunther Schadow, M.D. (Regenstrief Institute, US), Gunnar Klein (Sweden; chairman, CEN TC 251), Gerard Freriks (TNO, The Netherlands).

Insight into using archetypes for legacy systems was gained in discussions with Dr Dipak Kalra and Tony Austin of CHIME, UCL, where an archetype-based legacy federation system has been implemented as part of the EU-funded SynEx project.

Discussions with Dr. Bernd Blobel (Uni. Magdeburg, Germany) resulted in replacing the term “KR model” with “Reference Object Model”, a term closer to the existing names of existing health information models.

Improvements have been made due to online discussions with various people including Wayne Wilson (U. of Michigan), Dr. Andrew po-jung Ho (Harbor-UCLA Medical Center), and Tim Cook (president Free Practice Management).

Some important changes to the ontology section resulted from discussions with Dr. Beatriz de Faria Leao (Brazilian Health Informatics Association) during a trip to Brazil.

I would like to express my gratitude to all those others who have contributed comments, suggestions and corrections.

Much as I would like to claim credit for using the term “archetype” for archetype models in information systems, the usage here was inspired some years ago by Derek Renouf, a well-known object-oriented expert based in Sydney who used “archetypes” to configure Smalltalk systems.

LEGO® is a trademark of The Lego Group.

Contents

Introduction	7
System Development Methodologies	11
The Single-model Methodology	11
Shortcomings of the Classical Approach	13
“Standard” Models	14
When Is The Classical Approach Appropriate?.....	14
An Improvement: The Semi-structured Model.....	14
Generic Knowledge Representation	16
An Interoperable Knowledge Methodology	17
Domain Ontologies	19
Ontologies: The Gross Structure of Domains	19
Level 0 - Principles	19
Level 1 - Content	20
Level 2 - Organisation	21
Level 3 - Storage	21
Level 4 - Communications.....	22
Concepts: the Fine Structure of Ontologies	23
What is a “Concept”?	24
A Simple Solution: Templates	24
The Problem of Variability	26
A Better Solution: Constraint-Based Concept Definitions (Archetypes) ..	28
Composition.....	29
Specialisation	30
A Formal Language for Archetypes	33
Introduction.....	33
Structured Model versus Formal Constraint Language	33
Example Reference Object Model.....	34
Types	35
Basic Types	35
1:1 and 0:1 Relationships.....	36
Special Basic Types - Dynamic Variable Types.....	36
Container Types	36
1:N Relationships	37
Constructed Types & Automatic Class Generation	38
The General Archetype Model	39
Ontological Levels in the ROM and Archetype Models	39
Archetype Fragments	40
Naming and Archetype Paths	40
Relationships Between Archetypes	42
Composition	42
Runtime Archetype Validation of Data.....	43
An Example	43
Class-level Archetype (Local) Rules	45
The Big Picture	46

Alternative Approaches	47
Archetype Identification	49
The Archetype Space.....	49
Archetype Specialisation	51
The Specialisation Relationship	51
The New-version Relationship	53
Automatic Data Conversion	54
Forward Compatibility	55
Backward Compatibility.....	55
Archetype-based Querying.....	57
Archetype Query profile.....	57
Archetype Query Maps.....	58
Population Queries	59
Constructing Query Statements From Archetype Paths	60
Other Query Requirements.....	60
Advanced Features.....	61
More Sophisticated Validation	61
External Rules	61
An Advanced Archetype Model.....	62
Conclusions	63
Domain Empowerment.....	63
Future-proof Systems <i>and</i> Information	63
Interoperable Systems	63
Intelligent Querying	64
Automatic processing	64
References	65
General	65
Health	65

Introduction

“Classical” information systems are developed in such a way that the domain concepts which the system is to process are hard-coded directly into its software and database models. While this “single-model” approach may allow for relatively quick development, the usual legacy is systems which have a limited lifespan and are expensive to modify and extend in order to accommodate changing needs, primarily because it is already deployed software and databases which must be modified. In many such systems, particularly those developed in relational and/or SQL- and C-language technologies, a particular shortcoming is the non-explicit (i.e. implied) nature of domain level concepts, because the programming and database formalisms can express models only in the simplest data attribute terms.

With no explicit formal model available which formal developers can reason about or extend as needs change, the ability of software and databases to keep up with their requirements is limited.

Even in more object-oriented systems, where the model is clear, a basic problem remains: the software can never be “finished”, since new and changed domain concepts will always be appearing, forcing continual rebuilding, testing and re-deployment of systems. If changes are not made, the system suffer creeping obsolescence, and as a result, diminishing utility over time.

One common way of avoiding rebuilding software and databases is for developers to include a “black box” part of the model, in which data whose semantics are not described by the main model can be freely included. Over time, more and more data may be treated in this way, as the formal part of the model becomes harder to maintain.

As a consequence of these shortcomings, not only do most information systems today not serve their local users well in the long term, they also exhibit limited interoperability. Typically, they are only interoperable if they (i.e. their relevant vendors or development organisations) subscribe to the same formal model of information or services, i.e., they are standardised or productised. In the long term, the time-dependent “black box” effect severely limits the ability of systems to know what they can do with received data and also limits the ability of engineers to maintain them.

A different approach is needed, predicated on an idea of the world as a changing place, not a static one in which changes to requirements can somehow be regarded as exceptional. In many domains, both the total number of concepts and the rate of change is high; in health for example, there are thousands of concepts. For example, the SNOMED [23.] medical termset codes some 350,000 atomic concepts. The rate of change is hard to estimate, but at least five known effects contribute to change. These can be characterised as follows (taking the clinical health domain as an example):

- The rate of formalising currently understood concepts for the first time. Many well-known concepts such as “ECG results” in medicine have never been modelled in an agreed way because no appropriate language or infrastructure exists.
- Modified and new concepts due to improving technology. New kinds of devices enable new or more complex measurements to be taken; for example 3-d medical imaging machines are always being improved, requiring that the models for their results be updated as well.
- New research opens up whole new areas of the domain concept space. In medicine, genomics and virology are just two areas in which entirely new concepts are being defined.
- Changes in operational workflows and management. In health, changes in hospital procedures leading to modifications of clinical information models
- Changes in business process, information movement and privacy/security due to legislation leads to modifications of information models.

The approach proposed here is a rigorous knowledge-modelling one, and is founded on a basic tenet: *the separation of domain and technical concerns in information systems*. Practically, this translates to:

- The removal of domain concepts from concrete software and database models, into independently managed, standardised *vocabularies* and libraries of *domain concept models*.
- Re-engineering software and databases using a generic *reference object model* (ROM) system architecture, designed to process information by using externally supplied domain definitions.

The new approach to building software is one in which the analysis and design models express not domain concepts, but informational concepts which are capable of expressing *any* domain concept of a certain category. For example, rather than directly including the classes `DISTRIBUTOR`, `IGNITION`, `STARTER_MOTOR` in an automotive database system, a generic concept such as `ELECTRICAL_PART` might be used, or even just `CAR_PART`.

Standardised vocabularies, or structured dictionaries of standard term definitions, are a key element of a knowledge-based approach. Vocabulary development has existed in clinical medicine for years, and is finally beginning in other domains, as part of the general drive for business-to-business (B2B) and business-to-consumer (B2C) internet communications. Some advanced vocabularies encode meaningful semantic relationships, such as classification and associations between concepts, and it is preferable that such semantics are *not* replicated in domain models, but *used* by them.

The term *archetype* is used here for definitions of domain concepts, since it connotes “an original model, prototype or typical specimen” (Concise Oxford Dictionary). Archetypes serve various purposes:

- To enable users in a domain to formally express their concepts. Archetypes can be developed and change-managed by user groups, conferences, standards working groups, and any other forum available to users, without reference either to the systems that will process them, or to the developers or vendors of those systems.
- To enable information systems to guide and validate user input during the creation and modification of information at runtime, guaranteeing that all information “instances” conform to domain requirements.
- To guarantee interoperability at the knowledge level, not just the data structure level.
- To provide a well-defined basis for efficient querying of complex data.

A re-conception of information system engineering based on archetypes is the key to achieving widespread, knowledge-level interoperability. Under a methodology oriented toward *interoperable knowledge*, system developers need only agree on:

- Much smaller software models which describe *generic domain* concepts, rather than actual knowledge.
- A technical means of interoperability, such as CORBA, COM, HTTP etc.

Compared to the classical methodology, the size of models agreed to by software developers in this approach is actually much smaller.

The main difference is that it is now system *users* who define and agree on domain knowledge models, consisting of archetypes built from standardised vocabularies and rules.

The result is systems built from small reference models, which *process* archetypes in a way semantically equivalent to hard-modelled systems. This guarantees that all information created not only con-

forms to the concrete models (enabling database storage and intersystem transmission), but also to the domain models described in archetypes, thus ensuring that all co-operating systems can understand the information at the highest possible level. In particular, this allows effective automated processing to occur, such as by decision support systems (DSS), since these systems can now make many assumptions about the structure and content of the information in underlying systems, by inspecting the archetypes used to create it.

Archetypes are also the key to data validation can be used for batch data purification and interactive applications.

This paper describes a methodology consisting of a technical and managerial approach to archetypes designed to enable the advances described above.

It has been implemented and tested in the health domain, by the Good Electronic Health Record (GEHR; see <http://www.gehr.org>) project, using object-oriented technologies in underlying systems and XML-schema documents and tools for archetypes, and also by the SynEx project at UCL (<http://www.chime.ucl.ac.uk/HealthI/SynEx/>).

System Development Methodologies

We will start by investigating classical and alternative development methodologies. To make sense of either, we need to remember the purpose of any information system (as distinct from real-time or purely computational systems):

Purpose: the creation and processing of instances of concepts

Here, “concepts” means the entity types understood by the system, such as PERSON, ORDER, or PACKET; “instances” means actual occurrences of such types, usually in the form of the structured data representing a particular PERSON, ORDER or PACKET. (This is not to preclude the encapsulation of behaviour and data in the object-oriented sense; here we simply want to concentrate on the informational aspect).

The Single-model Methodology

Classical information systems, including most systems today, are constructed according to the scheme shown in FIGURE 1.

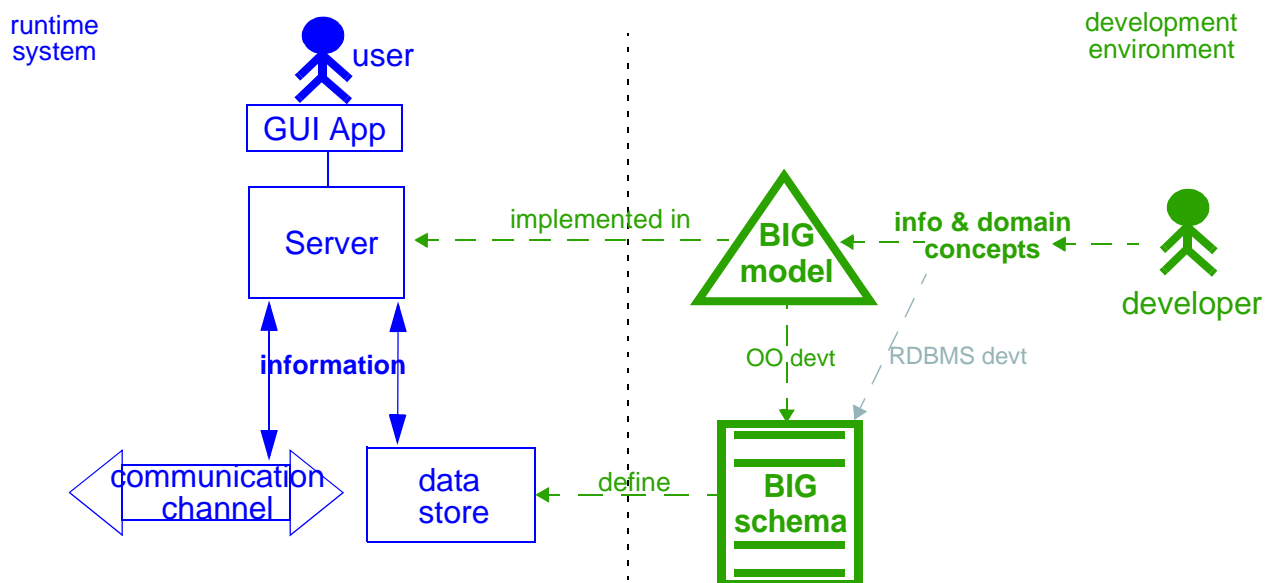


FIGURE 1 The methodology of most information systems today

The system creates information as instances of concepts, stores it, transmits it, and formats it for human use. The database, software and graphical user interface are developed based on an object-oriented (OO) or entity-relationship (ER) model, formally describing the semantics. In typical relational developments, concepts are encoded in the relational schema and informally into program code or stored procedures. In object-oriented systems, they are expressed as an object model in a formalism such as UML (Unified Modelling Language; see [5.]). Many systems use a combination of approaches, with object-oriented models being implemented in software, and also translated into relational schemas, resulting in the well-known “impedance mismatch”.

Even the most recent object system projects executed using (supposedly) advanced object-oriented analysis, design and implementation techniques (e.g. the Booch Method, Rumbaugh, or UML-based methods) end up as single-model systems; indeed most object-oriented methodology books describe system building as an iterative process of writing use cases, finding classes, and building a model which will eventually become software.

As a result, in a majority of object-oriented and relational systems, the semantic concepts are nearly all hard-coded. For example, the concept PERSON in many systems is modelled in a way similar to that shown in FIGURE 2. In this model, the entities contain *attributes* or have *relationships* to other entities (the latter shown by arrows). In true object-oriented systems, behaviours and constraints may be specified in object systems.

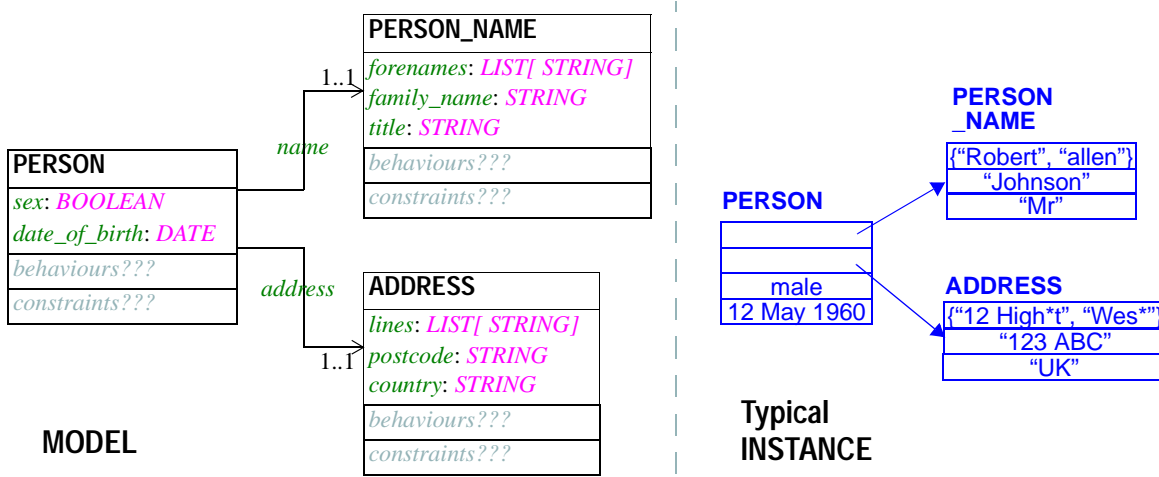


FIGURE 2 Concrete model and example instance of PERSON

The model of PERSON shown above would normally be only a fragment of a larger model, such as the one illustrated in FIGURE 3. The classes shown here are typical of the application of text-book object-oriented analysis and design methodologies, in which domain concepts, found during requirements capture, co-exist in the same inheritance hierarchy with more generalised concepts found during analysis. (This model also includes the classic error of modelling patient, nurse, etc as kinds-of person, which is common in demographic models. In fact “patient”, like many other demographic entities, is a party relationship).

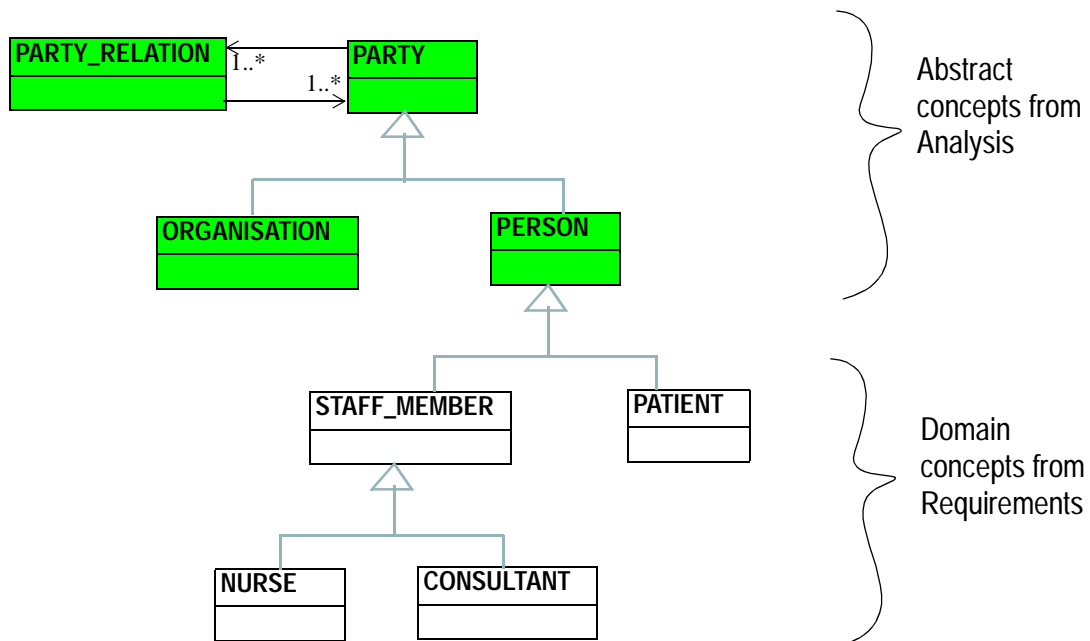


FIGURE 3 Part of a Typical Demographic Model (with typical errors)

Shortcomings of the Classical Approach

Problems with the classical modelling approach include:

- The model encodes *only the requirements found during the current development*, along with best guesses about future ones.
- *Models containing both generic and domain concepts in the same inheritance hierarchy are problematic*: the model can be unclear since very general concepts may be mixed with very specific ones, and later specialisation of generic classes is effectively prevented. The model is also not easily reusable in other domains.
- *Technical problems* such as the “fragile base class” problem (See [9.]) must be understood and avoided both initially, and during maintenance.
- It is often *difficult to complete models satisfactorily*, since the number of domain concepts may be large, and ongoing requirements gathering can lead to an explosion of domain knowledge, all of which has to be incorporated into the final model. In domains where the number of concepts is very large, such as health, this problem can retard software system completion significantly.
- There may be a *problem of semantic fitness*. It is often not possible to clearly model domain concepts directly in the classes, methods and attributes of typical object formalisms. Domain concepts have significant variability, and often require constraints expressed in predicate logic to complete their definition. A more powerful “language” for domain concepts may be needed. See The Problem of Variability below.
- Modelling can be *logistically difficult to manage*, due to the fact that two types of people are involved: domain specialists, and software developers. Domain specialists are forced to express their concepts in a software formalism, such as UML or a programming language (assuming they understand such formalisms), or more usually, make their requirements known through an *ad hoc* interface of discussions and document reviews with developers. Software developers often have difficulty in dealing with numerous concepts they don’t understand. The processes of investigation which would also naturally proceed at different rates are forced to occur in an unhappy synchrony, since domain investigation is typically more involved, whereas software requirements capture and analysis is usually driven by project deadlines. The typical result is a substandard modelling process in which domain concepts are often lost or incorrectly expressed, and software which doesn’t really do what users want.
- *Introduction of new concepts requires software changes, and typically rebuilding, testing and redeployment*, which are expensive and risky. If conversion of legacy data and/or significant downtime is also involved, the costs can become a serious problem. All of these cost factors have routinely made existing systems uneconomic in the past, and mandated complete redevelopment or replacement.
- *Interoperability is difficult to achieve*, since each communicating site must continually either make its models and software compatible with the others, or else continually upgrade software converters or bridges. Single-model systems which achieve interoperability usually satisfy at least one of the following:
 - Conformant to an interface standard (e.g. a an OMG IDL service interface definition).
 - Built from the same software (e.g. same vendor) or otherwise conformant to *exactly* the same information model.

- Use (typically complex and error-prone) converters to massage data into intelligible forms (the “stove-pipe” approach).
- Use only very simple messages.

Other than these approaches, the *ad hoc* approach of writing a converter between each pair of systems as the requirement appears results in an n^2 explosion of unmaintainable, non-standard software components.

Even when some level interoperability is initially achieved, it generally degrades over time, due to systems diverging from agreed common models, as they follow differing local requirements. See [10.] for a discussion of interoperability issues.

- *Standardisation is difficult to achieve.* With large domain models, it is logistically and technically (and often politically) difficult for different vendors and users to agree on a common model. Lack of standardisation not only makes interoperability difficult to achieve, it makes automated processing (such as decision support or data mining) nearly impossible, since there are almost no general assumptions such systems can make about the underlying model.

“Standard” Models

Many of the above shortcomings apply also to “standardised” domain models created by standards bodies, industry groups and governments. The worst feature of these typically very large models is that they embody no single point of view (in fact they are an amalgam of many), and as such cannot be used to build software. Large models do not create any significant improvement in the ability of their target systems to deal with change, although widespread adoption may make for reasonable interoperability, at least initially. However, local requirements need to be catered for, and doing this while remaining faithful to the standard model mandates a level of discipline in change management not found in most organisations.

In short, the single-model methodology produces systems which may work for the present, but whose utility degrades to a point where they become uneconomic.

When Is The Classical Approach Appropriate?

The list of shortcomings of the classical, single-model approach for application systems gives us a clue to what kinds of work it is actually appropriate for. It can reasonably be used for the development of software which does not contain volatile concepts, typically from a particular application domain. Components built around stable concepts, such as TCP/IP, or libraries consisting of classes implementing data structures, patterns and other re-usable concepts are all appropriate places to use the single model approach.

The mistake in classical methodologies has been to claim that they will work equally well for components or systems containing the ever-changing and growing concepts typical in all application domains.

An Improvement: The Semi-structured Model

The reason why concrete modelling fails outside the short term is because domain concepts are directly coded into the formalism used to build the software and databases. In other words, not only system correctness but informational validity is *directly dependent on the software entities from which the system is constructed*. Change in the requirements and consequently the model at least requires that the system be recompiled and re-started. Sometimes, with languages such as Smalltalk and Java, new classes can be added dynamically, but this does not alter the fact that the system has had to be changed (introducing new bugs), re-tested and re-deployed.

An alternative approach is to write software based on a model of *informational* concepts rather than actual knowledge. Many developers already do this in a simple way. Consider the model of `PERSON` described earlier. Obviously the fixed model of `name`, `address`, `sex` and `date_of_birth` is likely to become inadequate, if any of these data types changes (e.g. maiden or non-western names have to be handled), or if more details are required (for example a photograph or `place_of_birth`). A simple improvement to the earlier model is to make it “semi-structured”, as illustrated in FIGURE 4.

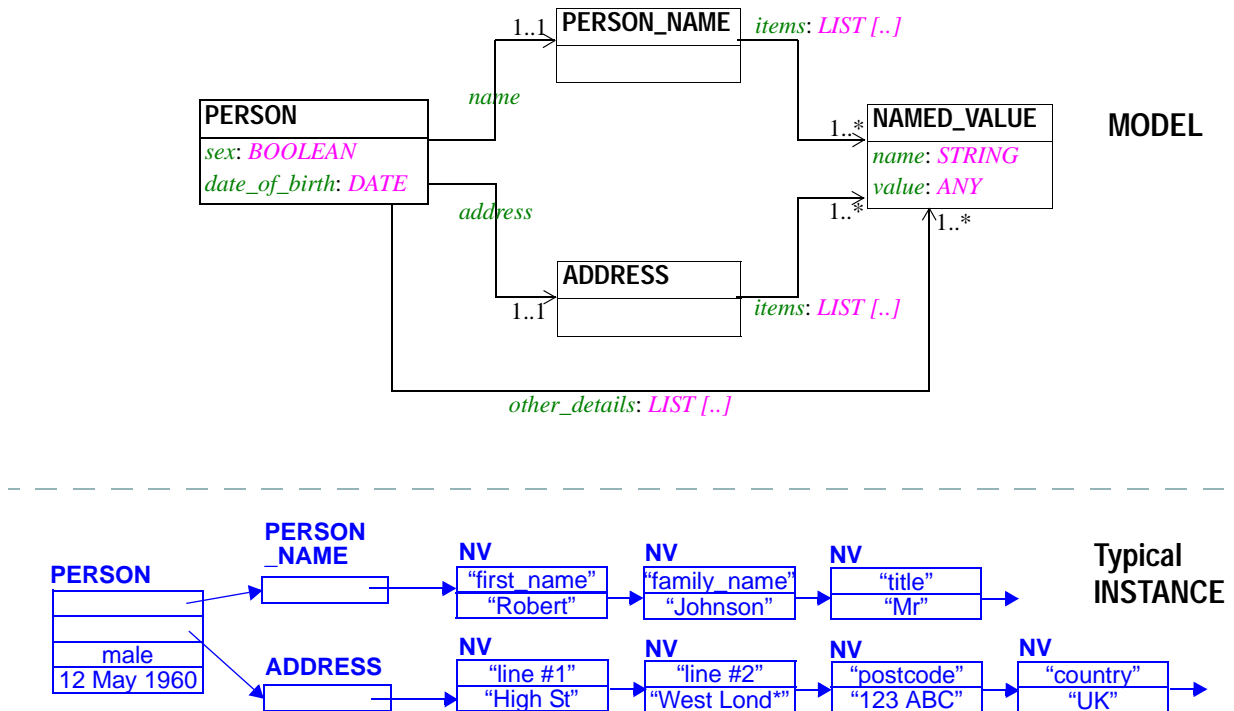


FIGURE 4 Semi-structured version of `PERSON` model

In this model, name and address are now modelled just as lists of `NAMED_VALUE` (sometimes called “tagged values”). The possibility of other data items is allowed for with `other_details`, also a list of `NAMED_VALUE`.

This approach is clearly an improvement, since there is now a possibility that the notoriously deceptive `name` and `address` fields can handle future possibilities not thought of at the time of original development. However, it does not solve all problems, and it introduces some new ones:

- The model is still partially concrete; it assumes that `sex` and `date_of_birth` are of certain types, and that they are all somehow more important than anything found in `other_details`. The `sex` field is still coded as a `BOOLEAN`, allowing only for male/female possibilities; in real systems, values such as “unknown”, “not disclosed”, and “unsure” are sometimes required.
- The `name`, `address`, and `other_details` attributes are just single lists of named values; they do not allow for more complex internal structures. Variability in structure is not generally dealt with.
- How does the system guarantee that at least the previously hard-coded data are provided (or alternatively, say which ones are required)?
- Strong typing is lost wherever the `NAMED_VALUE` class is used; now values are of type `ANY`.
- How much of a complex model should be converted to the unstructured form?

- What about behaviours found in object-oriented models: can they be dealt with in a similar way?

Generic Knowledge Representation

None of these problems is insoluble. Indeed it is fairly easy to see that a generalised approach is available by simply converting an entire model to a general knowledge representation design, such as a hierarchy (or a directed acyclic graph, an even more general structure) of NAMED_VALUE and NAMED_GROUP objects. This is the first step toward a type of system which represents general knowledge rather than particular types. FIGURE 5 illustrates such a model.

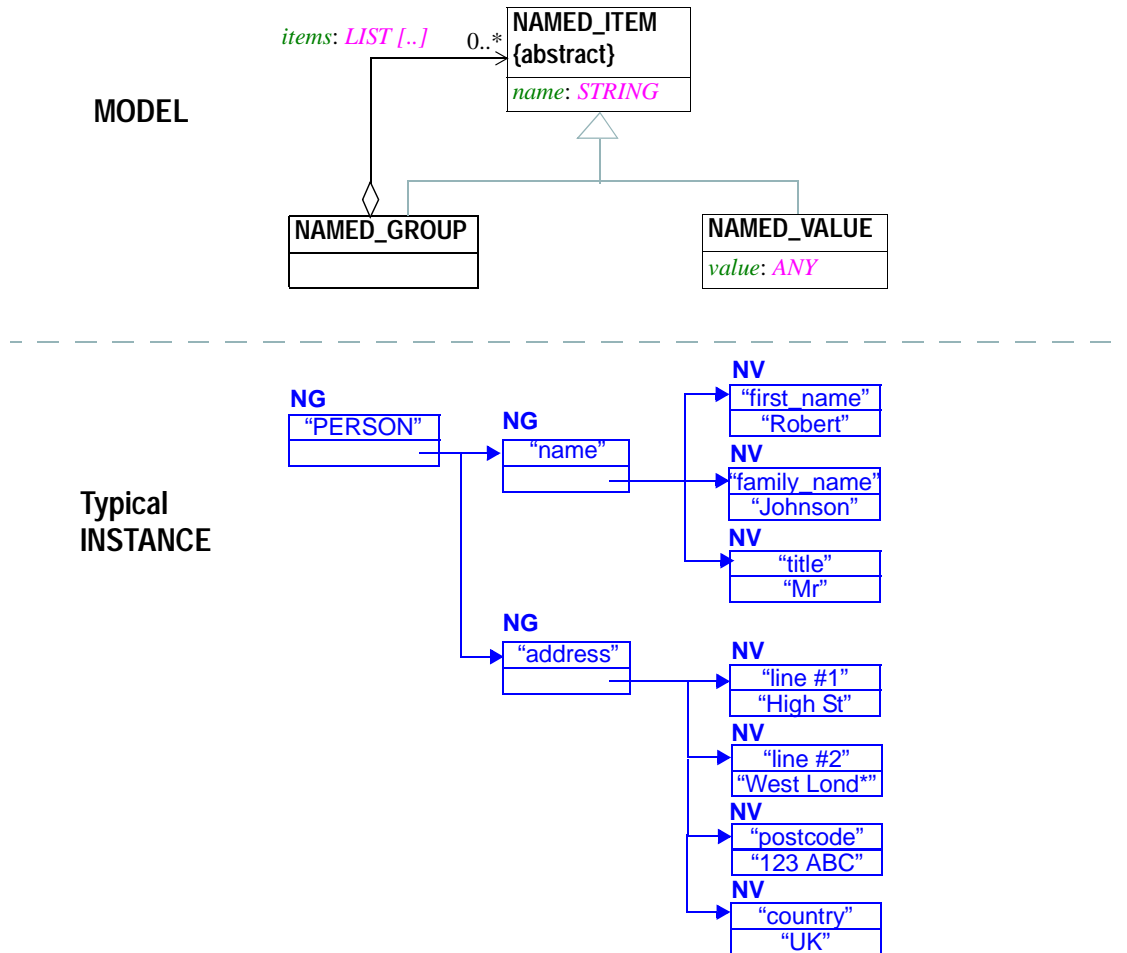


FIGURE 5 A simple generic model

However there is a now a real problem: if the model no longer describes domain concepts, where are they described? Some way of configuring generic knowledge structures is needed.

Systems which are heavily knowledge-oriented rather than concretely coded need more sophisticated knowledge models, since almost none of the original semantics remain in the concrete model. In the limit, the mechanism must be powerful enough to define for domain concepts, the valid:

- Types
- Values
- Structure

- Validity constraints
- Business rules

Depending on how much of the model is hard-coded, and how much is generic, various solutions are available, including configuration text files and new entities (classes) in the model whose job it is to describe valid combinations of knowledge objects (see [2.] for the latter).

In the model shown in FIGURE 5, it is clear that some way of standardising the values used in the name fields will be required for any kind of interoperability to work - the names are no longer part of the software code, but part of the data that instances carry.

A prerequisite therefore, for making any sense of models agreed across a domain is *common terminology*. While the names of classes, attributes and methods which appear in the class models of software systems may not be particularly relevant outside the immediate development environment, domain-wide models clearly need to use the same terms for the same things.

Designing a knowledge modelling approach is a hard enough problem in itself, but we also have to contend with the problem of interoperability: unless builders and users of different systems adhere to common terminology and generic reference models, in addition to a common technical solution, there is no hope for systems to share information. The question is exactly how this works.

An Interoperable Knowledge Methodology

FIGURE 6 illustrates an *interoperable knowledge methodology*, or “dual-model” methodology, in which standard domain terminology and concept models are used to drive the runtime functioning of software in information systems.

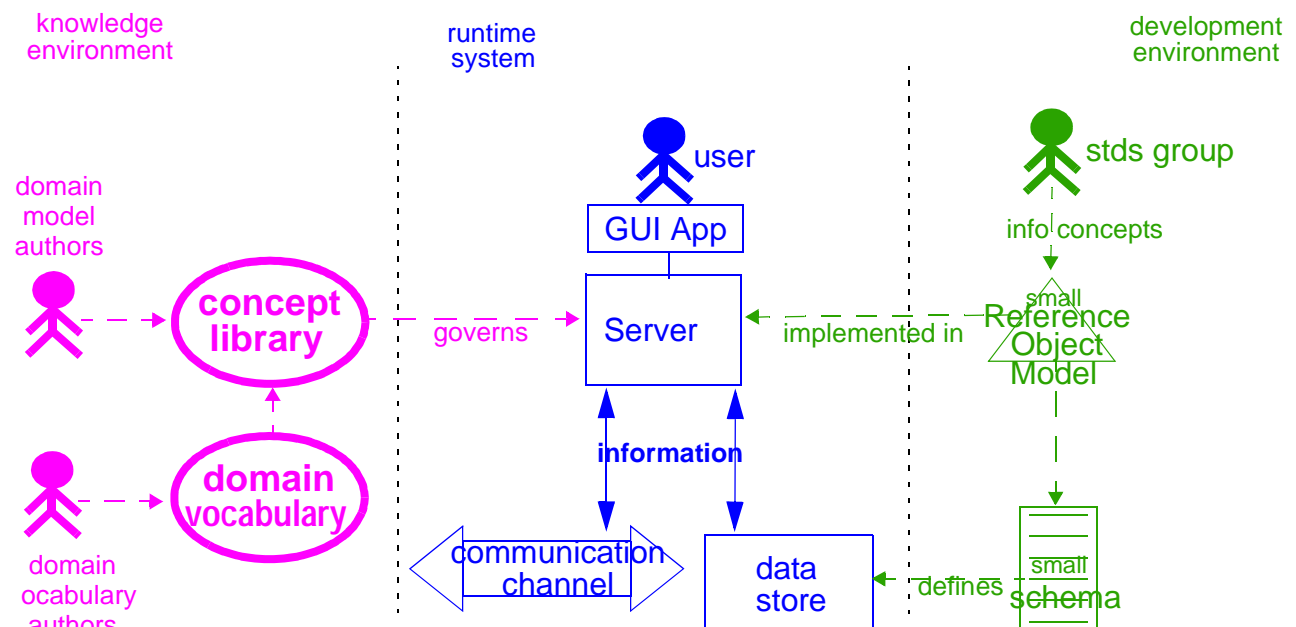


FIGURE 6 An Interoperable Knowledge Methodology

In this approach, the single, hard-coded software model has become a small *reference object model* (ROM), while domain concepts are expressed in a separate formalism and maintained in a concept library. Accordingly, *software development can proceed separately from domain modelling*.

Since the software is an implementation of the ROM, it has no direct dependency on domain concepts; i.e., *if new concept models are introduced, the system does not need to be altered*.

In the following sections we flesh out the nature of an interoperable knowledge methodology, based on the formal definition of domain concepts, which can vastly improve the quality of software as well as the level of interoperability.

Domain Ontologies

Let us revisit the purpose of information systems stated earlier: to create and process *instances of concepts*. Under a knowledge-oriented methodology, we are proposing that concept definitions be moved out of the technical model. Doing so raises a number of issues:

1. In what form are domain concepts now expressed?
2. In what language are they expressed?
3. How do we redesign the concrete model as an appropriate reference object model for externally supplied concepts?

The problem here is to find a way to conceptualise the mass of complexity that forms a real-world domain, and to be able to formalise it for both human and computer use, while escaping the naive, single model approach.

Ontologies: The Gross Structure of Domains

In the artificial intelligence (AI) arena, considerable energy has been focussed on knowledge modelling. The term *ontology* is used to refer to "an explicit specification of a conceptualisation [of a domain]" [7.]; in other words, a formalisation of (some of) the knowledge in the domain.

Although there appears to be no standard knowledge classification, a two-level separation of ontologies is often described, as follows (from [3.]):

- At the first level, one identifies the basic conceptualizations needed to talk about all instances of the of P [P stands for some kind of process, entity etc]. For example, the first level ontology of "causal process" would include terms such as "time instants," "system," "system properties," "system states," "causes that change states," and "effects (also states)," and "causal relations." All these terms and the corresponding conceptualizations would constitute a first-level ontology of "causal processes."
- At the second level, one would identify and name different types of P, and relate the typology to additional constraints on or types of the concepts in the first-level ontology. For the causal process example, we may identify two types of causal processes, "discrete causal processes," and "continuous causal processes," and define them as the types of process when the time instants are discrete or continuous respectively. These terms, and the corresponding conceptualizations, are also parts of the ontology of the phenomenon being analyzed. Second-level ontology is essentially open-ended: that is, new types may be identified any time.

This suggests that domain knowledge has a gross structure which is formalised into (at least) two ontological levels.

Level 0 - Principles

The first level, which we will call level 0, can be thought of as an ontology of the *language and principles* of a domain. In clinical medicine, principles relate to subjects like anatomy, parasitology, pharmacology, measurement and so on; the knowledge of processes and entities constitute the generally accepted facts of the domain - things which are true about all instances of entities (such as the human heart) or processes (such as foetal development). As such, level 0 knowledge is independent of particular users of information or processes such as health care or education; we might say it has *no point of view*.

It is also very stable, i.e. non-volatile. Concepts which are specific to particular use contexts, uncertain, or supposition should not appear in level 0 ontologies due to the dependency of all other ontologies in a domain on this level, and also the widespread use they would normally have.

Level 0 ontologies are what we find expressed in textbooks, academic papers and training courses. Medicine is one of the few domains to also have basic domain knowledge in a highly computable form: it exists in structured vocabularies, such as SNOMED [23.] and READ [ref]. Although such vocabularies do not express all the semantics of basic concepts - they are generally limited to terms, definitions and some semantic relationships - they offer significant value in the computer processing of information at a knowledge level.

Level 1 - Content

In the second ontological level, knowledge becomes more specific to *particular uses and users*. We can divide the second ontology into a number of sub-levels, according to the various use contexts, which we will number 1 to N. As a consequence, we can assume that all concepts in levels 1 to N will be *separately identified*, since they represent particular compositions of vocabulary items and other constraints into structures, similar to the way atoms are composed into molecules. In other words, level 0 may be a large semantic network, whereas levels 1 to N will consist of separate concept definitions.

The first level, level 1, is simply the composition of level 0 elements into basic *content* structures. In some domains, we can sub-divide this level into *ubiquitous* content and *use-case based* content; the former are concepts which everyone in the domain uses and understands in the same way. Examples in clinical medicine include:

- blood pressure
- body mass index
- body part measurement

Each of these represents a particular *use* of elements of the first level ontology. For example, blood pressure as a clinical measurement is commonly defined to be a composition of systolic and diastolic pressures, i.e. the pressures at the 1st and 5th Korotkoff heart sounds. In a hypothetical underlying vocabulary, this particular association might not be found as such; instead, the pressures for all the Korotkoff sounds, as well as venous and arterial pressures might be found, classified under “blood pressure”, as illustrated in FIGURE 7.

The two-valued blood pressure in common clinical use represents a particular selection of vocabulary items to form a useful clinical information entity representing an observation. In the same way, the other examples above are ubiquitously used compositions of underlying semantic elements. There is no guarantee, of course, that all instances of such models are identical - small variations may occur on the theme, such as removal or addition of optional items in a medication order, and of course numerical values will always vary.

The second half of level 1 can be identified by taking into account *specific* processes which occur, according to particular scenarios, or “use cases” (see [5.]). For example, the following concepts commonly occur due to clinical or health-related processes:

- Referral
- Adverse reaction (patient’s description of known reactions to drugs etc)
- Family subject history

Likewise, in pathology, there are numerous laboratory tests corresponding to particular use cases. These types of concept are often understood differently by different domain users.

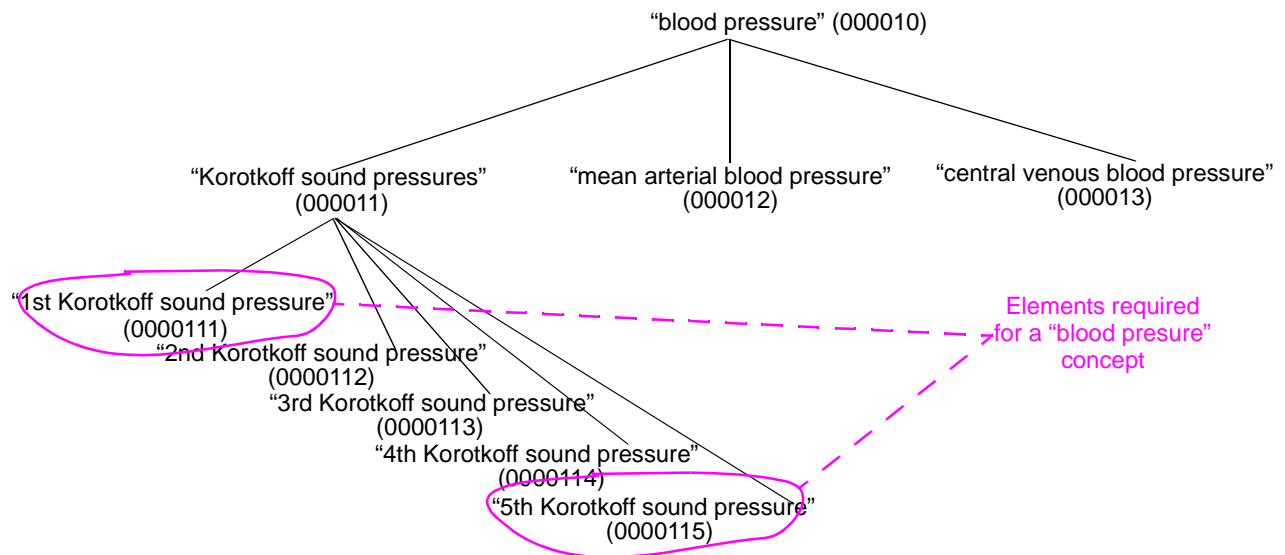


FIGURE 7 Hypothetical Vocabulary Rendering of “blood pressure” concepts

Level 2 - Organisation

Two further levels are also useful: *organisational*, and concepts relating to *storage*. Organisational concepts (level 2) are created by domain users in an attempt to make sense of what might otherwise be a sea of unrelated items: they are a navigational aid to readers of information. They are typically defined according to high-level methodological or process ideas; for example the “problem-oriented health record” gives rise to a very common organisational device known as the “problem/SOAP” headings, a hierarchical heading structure of the form:

```

<problem>
  "subjective"
  "objective"
  "assessment"
  "plan"
  ...
</problem>
etc
  
```

This heading structure is used in general practice to organise various information items a physician obtains during a patient encounter. Like other lower-level concepts, its standardisation is useful, since it permits both human readers and computers (e.g. decision support) to make assumptions about navigating patient encounter information. Other heading structures are used in structuring information relating to:

- Referrals
- Discharge summaries
- Most patient examinations, e.g. cardiovascular exam, pre-natal exam, eye exam

In general, heading structures correspond to typical activities which domain practitioners undertake, and then document, and as such are a vitally important ontological level.

Level 3 - Storage

The levels so far described allow us to create “organised content, expressed in terms of basic elements, including vocabulary entries”. At level 3, we need to consider how such information will be logically packaged with respect to its subject (what it is about). For clinical information about a

patient, this level corresponds to the gross structure in which the information is stored, usually called a “health record”. Items of information at level 3 need to be meaningful in their own right with respect to the subject of the information. That is to say, they must include all the contextual information relating to their collection or creation, such as the identity of the recorder, date/time of recording and so on. Examples of level 3 concepts in clinical medicine include:

- Family history
- Current medications
- Therapeutic precautions
- Problem list
- Vaccination history
- Prescription
- Patient contact

Level 4 - Communications

A final level, level 4, is concerned with concepts relating to the selection and packaging of information for communication with other users. Typical concepts are “document”, “report” and “extract”. The GEHR electronic health record project includes a logical “EHR extract” concept which defines the package of information to be sent to other systems.

In summary, concepts in the second level can be classified into a number of qualitatively different layers, or sub-ontologies, which for health are summarised in Table 1. This particular classification is not claimed to be normative of course; rather it represents one way of partitioning the health domain to make it more tractable for the design of formal ontologies. It is based on the GEHR work and also work described in [XXX beatriz] and [XXXX angelo].

Level	Meaning	Expression	Examples
0 principles	Vocabulary and other stable semantics of domain, facts true for all instances and all use contexts	Semantic networks, controlled vocabularies.	- textbooks - SNOMED, Read, ICPC - statements about quantitative data
1a content (ubiquitous)	Widely used context-dependent concepts with a common understanding by all users in the domain.	Compositions of level 0 concepts	- blood pressure - body part measurement - medication order
1b content (use-case based)	Context-dependent concepts defined according to particular use cases.	Compositions of level 0 concepts	- adverse reaction - family subject history - structures implied in LOINC lab codes
2 organisational	Structural information concepts whose purpose is to organise information, in the same way as headings in a paper document.	Hierarchical structures of level 1 concepts	- problem/SOAP headers - alcohol and tobacco use - family history - referral

Table 1 Knowledge Classification for the Clinical Medicine Domain

Level	Meaning	Expression	Examples
3 storage	Concepts relating to the gross structuring of information for storage.	Compositions of level 2 concepts	- transactions for current medications, problem list etc - EHR
4 communication	Concepts relating to the packaging of information for the purpose of sharing.	Extracts or packages derived from level 3 information	- document - EHR extract

Table 1 Knowledge Classification for the Clinical Medicine Domain

We can visualise the knowledge space in a multi-level form as per FIGURE 8 (three dimensions have been chosen purely for visualisation purposes; in fact the real number of dimensions is higher). Points on the diagram stand for concept descriptions; the sum of concepts at a given level forms the ontology for that level. Concepts at outer levels are generally composed from those at inner levels, with everything ultimately devolving down to elements from the principles level.

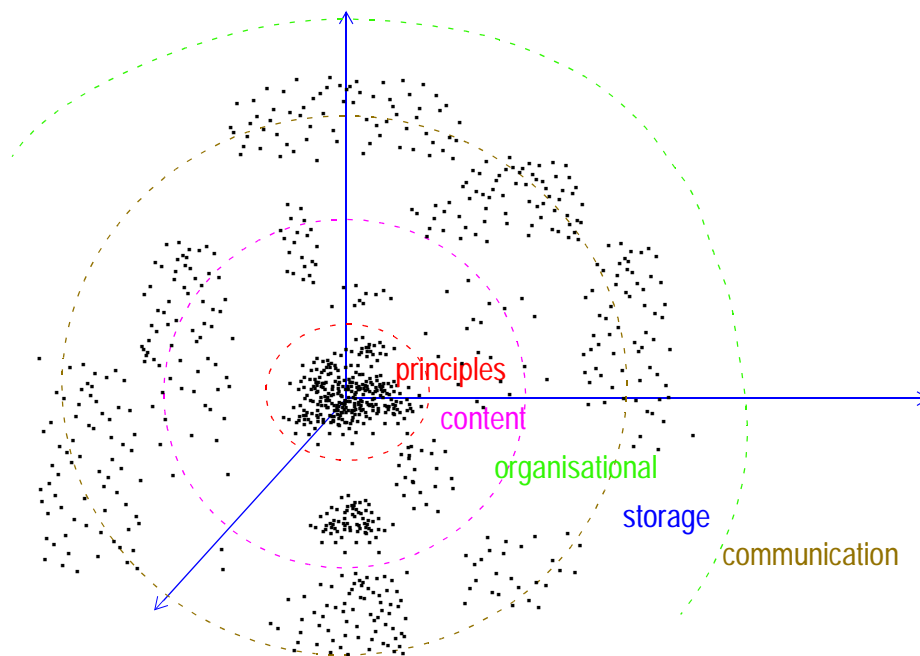


FIGURE 8 A multi-level knowledge space

Concepts: the Fine Structure of Ontologies

So far we have developed an idea of the gross structure of domains, using the example of clinical medicine. But in order to define the ontologies at the various levels, we need a formalism for doing so. The following discussion shows one method of understanding domain concepts which leads to convenient and technically viable means of representation.

As a starting point we will make a basic assumption, which is that the formalisation we seek will treat concepts as *discrete entities*, i.e. separately identified entities in the domain. We would thus talk of an ontology as the sum of (some of the) concepts in a domain, expressed formally. The primary motivation for having discrete concepts is to do with information management: it must be not only possible but also convenient to define, review, disseminate and use concepts without reference to all other con-

cepts. Not to do so invites paralysis - it would mean that nothing could be done until all concepts were defined, and each ontology completed. Note that the intention is not that concepts cannot refer to each other, indeed, as we have already seen, compositions of lower order concepts is likely to be very frequent. However, a well-structured ontology above level 0 should exhibit low coupling, i.e. contain a relatively small number of references from any concept to other concepts.

What is a “Concept”?

Whilst a truly formal definition of “concept” is unlikely to be possible outside of pure philosophy, it is useful to have an informal idea. Let us use the following definition.

Concept: coherent description of an idea in a domain, which is separately identified by domain users, and used in a self-contained way to communicate information.

Here we are saying that:

- Concepts are only concepts if recognised as such by domain users
- Concepts are identified (i.e. have a unique name)
- Concepts are self-contained
- Concepts are the granularity at which information is communicated (transmitted, recorded etc) in the domain

Concepts exist in each of the ontological levels described above as we have shown. What we are now concerned with is their formal definition and representation, particularly for computer use.

As mentioned above, at the “principles” level, concepts are often represented as a semantic network, whether in the form of textbooks or controlled vocabularies and rules. This level of knowledge can be understood as a large “sea” of interconnected facts, classifications, definitions etc. Since its main purpose is to provide a basic language for the domain, such a representation is reasonable, as long as there are ways of accessing and navigating it, and as long as it contains only non-volatile knowledge; if the latter is not true, change management is likely to become impractical.

Knowledge at each of the other four levels, however, is expressed in terms of the concepts at each previous level.

Let us start with the “blood pressure” example from the “content” ontological level (level 1 of the clinical medicine domain). Blood pressure is normally understood to be a grouping of two quantities, representing the systolic and diastolic blood pressure measurements for a patient. It is a meaningful concept, because the definition is clear, it is used ubiquitously in clinical practice, and it is a very common unit of information used to record and communicate blood pressure.

Compare this with “systolic blood pressure”. While the meaning of this term is perfectly clear in medicine (it is the pressure at the end of the systolic contraction period, or first Korotkoff sound), clinicians do not usually measure "systolic blood pressure" on its own (exceptions include hypotensive patients), and a description of the term would be understood by most clinicians as being a part of the definition of “blood pressure”; finally, where the clinician intended to record blood pressure, s/he would not record the systolic and then the diastolic blood pressures separately - they would be recorded as a single unit (indeed it would be surprising if the software were written any other way).

A Simple Solution: Templates

It would appear that the concept “blood pressure” in medicine could be represented by a model such as the one shown in FIGURE 9. The only variables are the actual pressure values. The third leaf element expresses the simple rule, or constraint, that guarantees that the relation of magnitudes is cor-

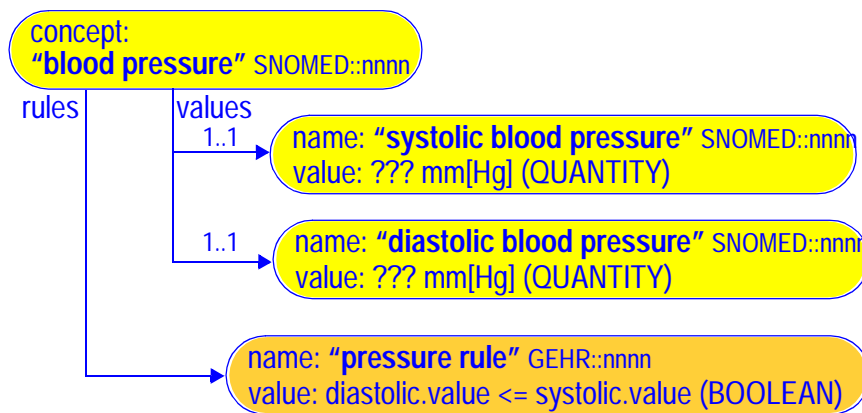


FIGURE 9 Simple Model of “Blood Pressure”

rect. Each leaf element in the model should theoretically be understood as a “behaviour”, i.e. in the functional sense. In other words, “systolic blood pressure” is not meant to indicate a stored data item, but a function returning a blood pressure. The distinction will seem fine for some, and indeed it is not that important, as long as abstract models consisting of logical types are used in general, rather than the kinds of database field specifications which say that “name” is a STR field, 20 characters wide, and so on.

A declarative definition of the above graphical model is also possible, as illustrated in FIGURE 10. Here we use a semi-formal notation based on the Eiffel object-oriented specification and programming language [8.], which is semantically equivalent to the combination of UML and OCL (Object Constraint Language).

```

class "blood pressure"
  feature -- values
    "systolic blood pressure" [SNOMED term nnnnn]: QUANTITY
    ensure: Result.units = UNIT mm [Hg]
    "diastolic blood pressure" [SNOMED term nnnnn]: QUANTITY
    ensure: Result.units = UNIT mm [Hg]
  invariant
    "blood pressure rule" [GEHR term nnnnn]:
      "diastolic blood pressure".value <= "systolic blood pres-
sure".value
end
  
```

FIGURE 10 Declarative expression of simple Blood Pressure Model

We can think of the model in FIGURE 9 and FIGURE 10 as a *template*: a predefined “form” of a complex object, in which the values simply need to be filled in. Expressing concepts as templates is certainly an attractive idea: it would mean that domain user and professional groups could get together and decide the structural definitions of concepts, compile a library of them, and make them available for systems to use. It would also seem to imply that an object-oriented specification language could be used for concept modelling.

The Problem of Variability

Unfortunately, the world is not so simple. In real clinical medicine, practitioners regularly employ variations on the basic model, and still think of it as “blood pressure”. For example:

- Not all clinicians or information systems use exactly the strings “systolic blood pressure” and “diastolic blood pressure”; due to coding systems, local norms etc, variations might occur, such as “systolic” / “diastolic” and “systolic pressure” / “diastolic pressure”.
- The addition of another datum, the “4th sound (pressure)”, which may be used by paediatricians or in patients with certain known disease states or abnormalities. In general, there may be a requirement to allow additional data items.
- The addition of a *protocol*, describing how the measurement was taken, including the data such as “patient position”, “device”, and “cuff size”.

An improved model is illustrated in FIGURE 11. All elements containing variability are shown in bold red.

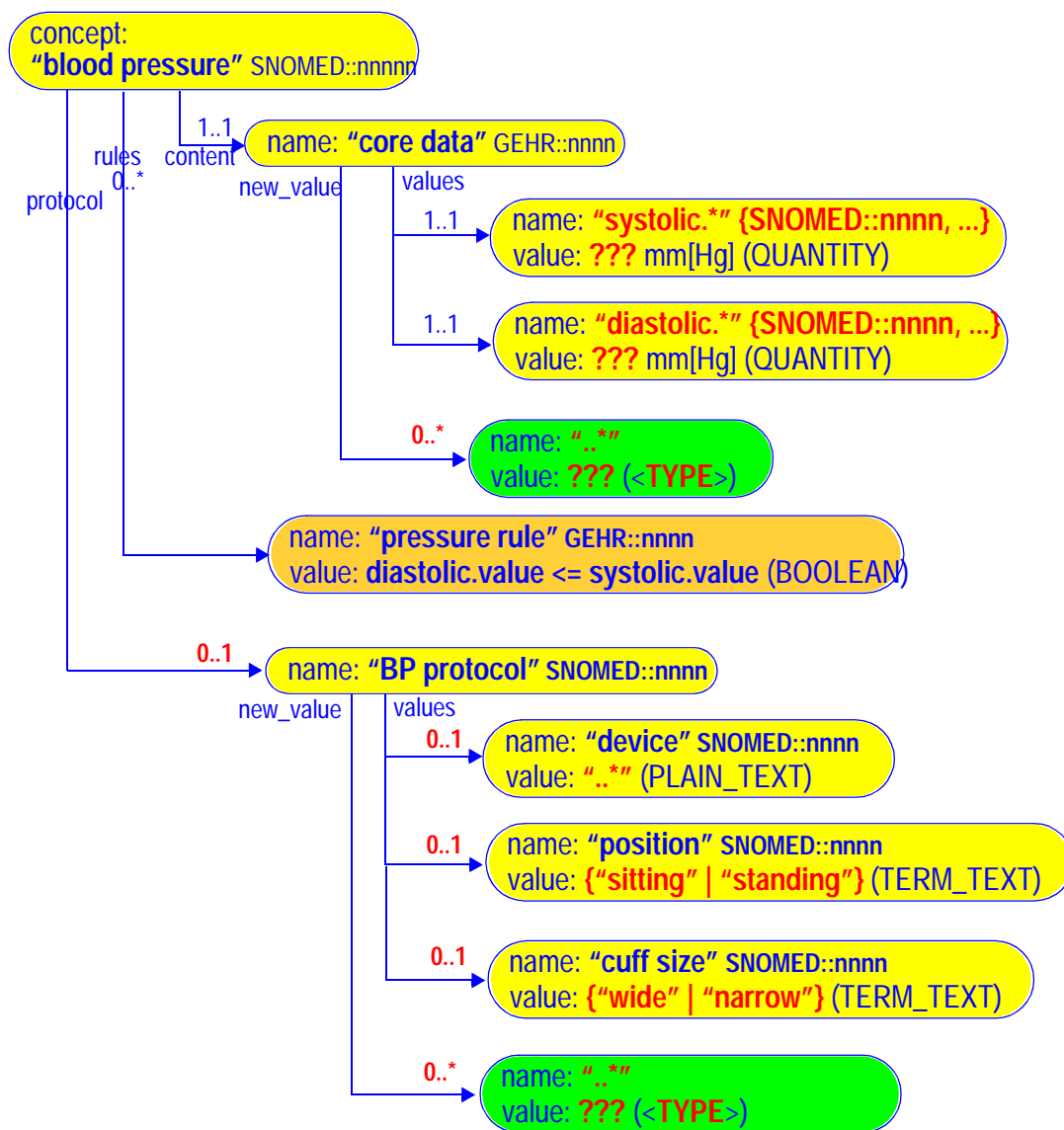


FIGURE 11 Improved Domain Model of “Blood pressure” Concept

Variation in names is handled by using coded terms, sets of terms, or patterns such as "systolic.*" (to be understood as a *regular expression* of the kind found in almost all computing platforms using languages such as Unix shell, C or PERL); extra data are catered for by the final item in both the "values" and "protocol" lists, which has the semantics of "the name, value and type of new data items are all unconstrained"; the protocol group is added to model clinical protocol, which is indicated as being optional by the "0..1" multiplicity marker.

The exact model of blood pressure used here is of course not the only possibility. But we can see that even a relatively simple concept can have a lot of variation. More complex concepts (e.g. "prescription", "ECG results"), and particularly more subjective, or ill-defined concepts (e.g. "family history", "address") have even more variability.

We can also see that trying to express the above model directly in an object-oriented design specification language might be difficult; such formalisms do not deal with class features with variable names or types, much less feature definitions for items that might or might not exist. Candidates for textual expressions of domain concepts exist, such as KIF (knowledge interchange format, [6.] - a functional prefix notation for predicate calculus). However, a better approach is likely to one derived from more structural object-oriented/functional constraint languages, where each concept model is a set of statements (i.e. an "instance") in the language; we will return to this point later.

Another particularly useful example of variability is that of state variables. In a clinical model for "medication order" (an item in a "prescription"), a status might be included, indicating the current life-cycle state of the medication administration. The allowable values are defined by a state machine, such as the one shown in FIGURE 12, which can be encoded conveniently in the form of a state/event table in the clinical model.

This approach is far richer than strict templates, in which the state variable would just exist as a single INTEGER or STRING attribute to be filled in at run time.

In general, a "simple" dynamic variable might in fact have quite a complex specification in a concept model. Further, if such a variable was represented in the ROM by a technical variable such as *status*:INTEGER, different concept models can provide completely different specifications for the same variable.

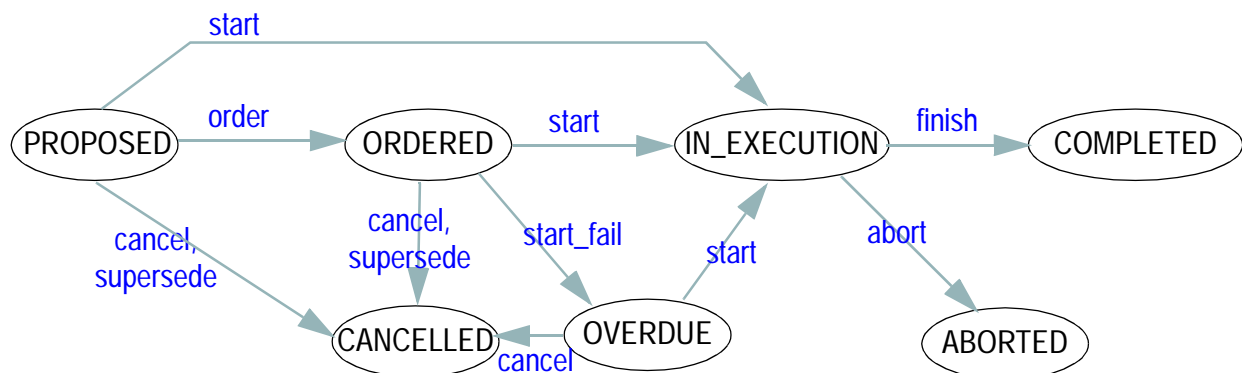


FIGURE 12 State Diagram for Clinical Instruction

In summary, variability in domain concepts occurs in:

Free text names: e.g. ".*" is a pattern defined to match any string etc.

Names coded with terms: which are matched by either a set of terms, or else terms whose meaning in their term-set is a parent of all the allowed terms.

Values: e.g. pressures are allowed only to be a QUANTITY type, consisting of a number followed by the units "mm[Hg]"

Types: e.g. new values may have to be of a particular type. The possible types are defined according to the domain. Where types are allowed to vary, there may be value constraints for each type.

Structure: e.g. protocol is shown as being optional; also, any number of new values may be added to the values list, as indicated by the "0..*" multiplicity marker.

Relationships: e.g. the "pressure rule" shown above. Constraints describe allowed relationships.

Behaviour: as modelled by dynamic variables.

A Better Solution: Constraint-Based Concept Definitions (Archetypes)

The discussion so far shows that one concept corresponds to potentially numerous variations on a notional "basic" definition. This is exactly as in real life: our concept of "person" for example, corresponds to a large number of variations on the notional "human being" concept, including all kinds of differences in physiognomy, personality and behaviour. An important question would seem to be "what amount of variability is allowed, before a given concept becomes something else"? Put another way, what we really want to know is what the *limits* of the definition of a concept are.

We are led to the idea of a concept model being a *structural constraint definition*, i.e. the set of constraints which together define the set of instances which can still be called the same concept. At this point, it is important to mention that the intention is not to come up with a single, "perfect" definition of each general concept; for example, there is no need to have only a single "blood pressure" concept in clinical medicine. Indeed, other related concepts might be "mean arterial blood pressure", "central venous pressure" and "target blood pressure". What is important is that for each concept the domain wants to use, a definition can be developed in terms of constraints on structure, types, values, and behaviours.

A constraint-based concept definition is far more powerful than a template, and for this reason, we call it an *archetype*, since it is a *prototypical model, encompassing numerous possible individual variations*.

Our reconception of information systems is now one in which domain concepts are formally modelled as archetypes, which are then *used* by the system at runtime. The "software" as we previously knew it is now an *implementation of a reference object model*.

An archetype effectively corresponds to a *constellation of valid combinations of ROM objects for a particular domain concept*. The "Lego analogy" (see box below) is sometimes helpful in understanding archetypes.

We can revisualise the ROM construction space, and the instance combinations described by a number of clinical medicine archetypes as per FIGURE 13. In this diagram each circled group of points can be understood as the set of information instances which can be called the one concept; in other words, each concept represents a boundary in information construction space. Compare this with templates: a template can only correspond to those points of identical structure, with different values.

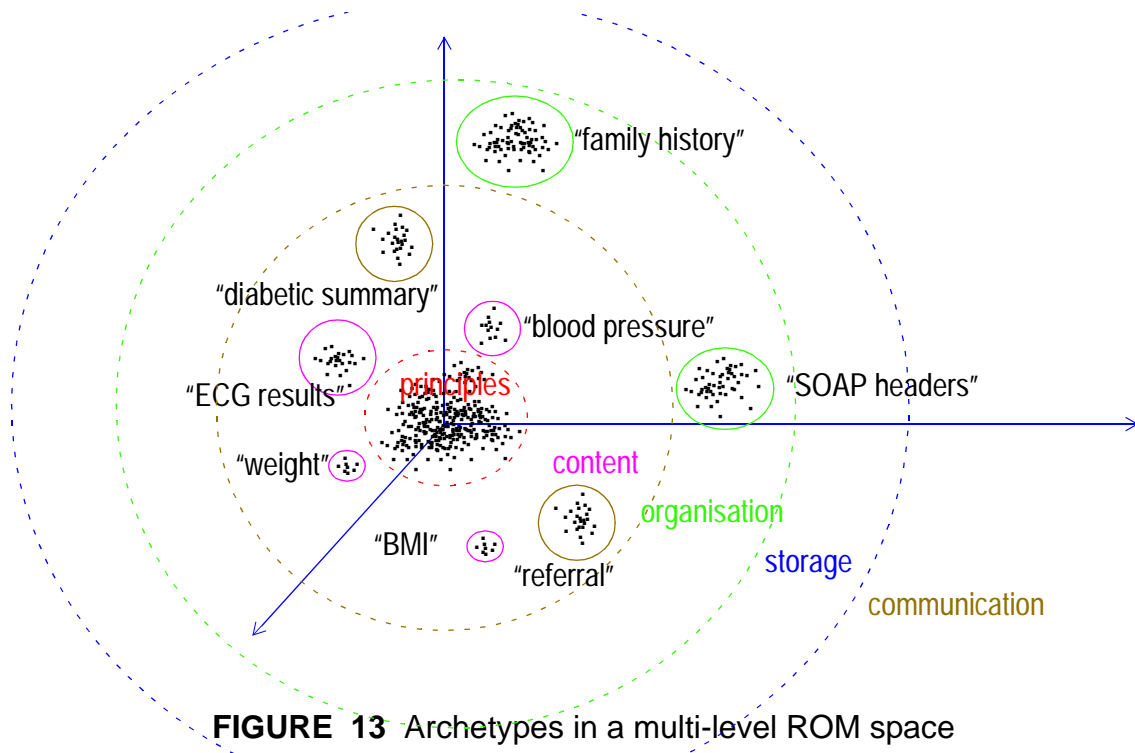
Of course, trying to express concepts explicitly as lists of particular information instances would be extremely unwieldy. Instead, we need a way to express archetypes as utterances in a formal language which will generate the intended set of instances.

Before going into a formalism for expressing archetypes, two further important requirements need to be mentioned.

The LEGO Analogy

One way to understand archetypes is to imagine that the Reference Object Model defines the engineering specification of LEGO® bricks from which, as every child, and not a few adults know, anything can be built. The semantics of the ROM are analogous to the “semantics” of Lego bricks, i.e. the engineering specification of the particular coupling and joining mechanisms built into the bricks. The set of all possible combinations of a particular set of bricks comprises a vast construction space. However, most combinations are meaningless - only a tiny proportion of the space consists of the interesting constructions of houses, dogs, and tractors; all other combinations are “legal” if the bricks are connected correctly, but have no meaning to us, the users. Likewise, a ROM defines a vast informational construction space, only a small proportion of which contains combinations valid in the domain.

Consider further that the valid Lego brick constructions cannot be divined from the bricks themselves: they come from fertile imaginations, or else printed plans included in Lego packages. It is often the case that small variations and optional add-ons are suggested for the one model; this means that the set of all possible variants on the model form a constellation of brick combinations corresponding to the one plan, or model definition. Such plans are the Lego versions of archetypes.



Composition

The blood pressure example provides an insight into how to construct a single concept model, but we still need a way to create archetypes in the higher ontological levels, which are themselves composed of lower level archetypes. A simple medical example of composition is illustrated in FIGURE 14.

Family history information is usually recorded by physicians under the simple heading structure shown in the top half of FIGURE 14, while the actual history information for each person to whom the patient is related in some way is recorded in a standard “subject family history” structure, shown at the bottom of the figure. The red elongated hexagons in the top figure represent references to other archetypes. The first concept is an organising concept, while the second is a content concept. Each concept can be understood in a standalone sense, but they are usually used together in order to build a

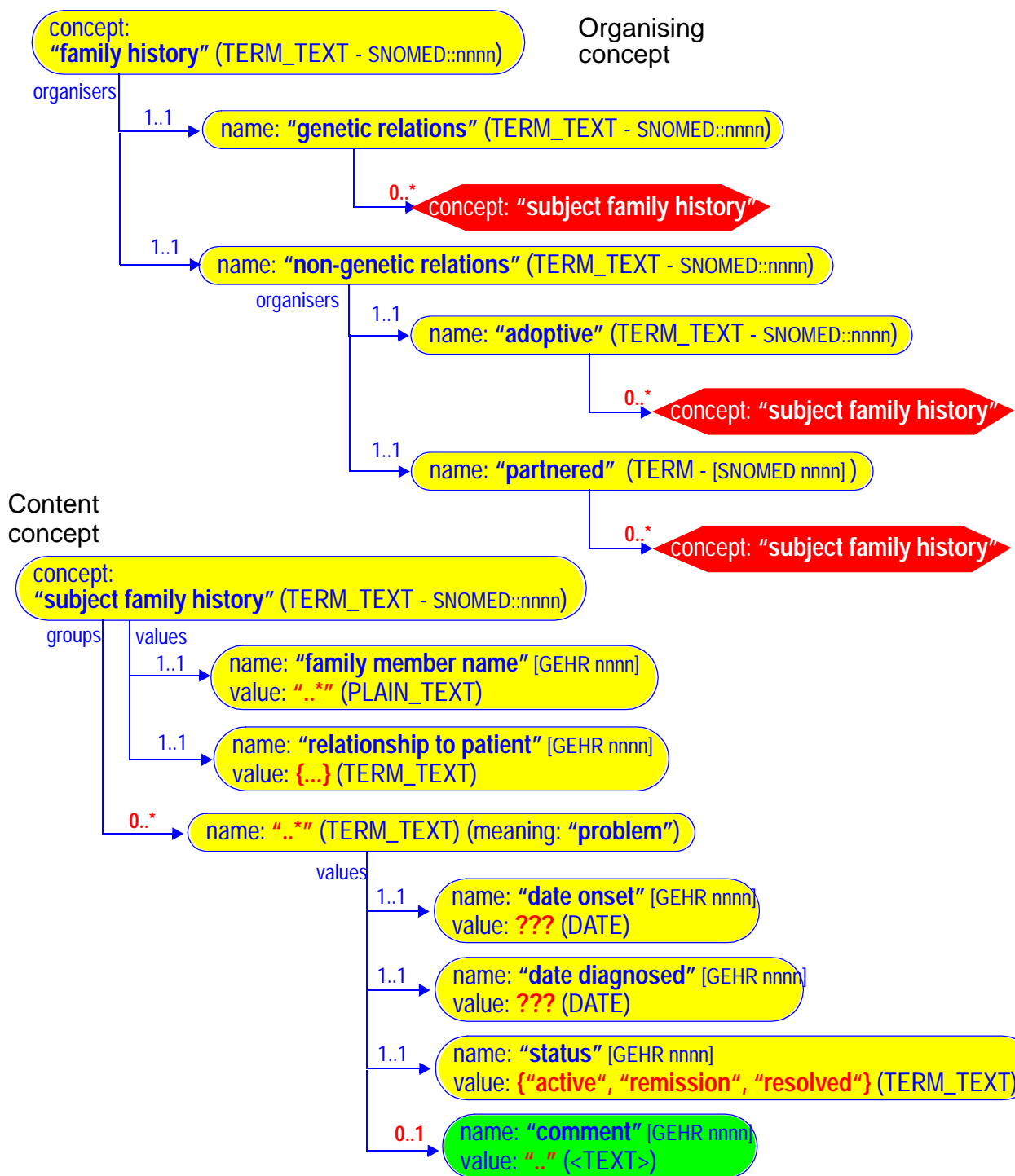


FIGURE 14 Composite Concepts

complete family history description for a patient. There is nothing stopping the use of the content structure in other organiser structures, or vice-versa.

Specialisation

Concepts are also related to each other in specialisation relationships. Specialisation might occur due to localised alterations to standard concepts, such as are typical with “medication order”. Concepts

which relate in any way to legislated norms for information are almost always different across countries, even though domain professionals think of them as being the same logical thing.

Specialisation might also occur due to professional specialisation. For example, a general practitioner's notes for an "ENT exam" (ear, nose & throat exam) are more basic than those for the same concept used by an otolaryngologist.

One very strong motivation for allowing specialisation is that while national specialisations of archetypes will be used in their own contexts, international sharing of information may be more effective if recipients can request a view of information according to a common parent archetype, or if they receive the information in its whole form, they can process it using the common parent archetype, disregarding irrelevant (to them) details.

In these cases, what is required is the ability to define a new concept on the basis of an existing definition. Exact rules need to be defined depending on the archetype language used, for including, extending, restricting or otherwise changing the definition of the existing archetype to create a new one.

A Formal Language for Archetypes

Introduction

The previous section showed that a knowledge domain can be conveniently formalised by first considering it as two or more ontologies, and then modelling concepts within the second, third, etc ontologies as distinct archetypes. Here we want to show exactly how such models can be expressed, and how they relate to the underlying reference object model.

Recall that the ultimate aim of the runtime system is to create instances of concepts. The building blocks from which such instances are constructed are defined by the concrete model (as is the case in classical system development), which is now designated a *reference object model* (ROM).

One way to define an archetype language is as a “constraint model”, or as we will call it here, an *archetype model* (AM), formally related to the ROM. Such a model allows constraints on structure, attributes and types to be expressed. Not all constraints are conveniently expressed structurally; more complex constraints, such as on covarying attributes, may need to be expressed as rules. Workflow process definitions may eventually be expressed in archetypes, and these may require a special sub-language as well.

Structured Model versus Formal Constraint Language

Before delving into the details of archetype models, it is useful to consider an alternative, namely a constraint language consisting of statements about instances of the ROM. This is the approach favoured in the AI community, and by semantic network specialists.

In fact, the two approaches are not really alternatives, but equivalents. Consider archetypes expressed in a constraint syntax. The steps in processing them in a typical computer system are as follows:

- Retrieve statements representing an archetype from database
- Parse and validate statements
- Generate a parse tree
- Visit the nodes of the parse tree, in order to execute the statement

When authoring such statements in rule-based systems, a typical approach for a GUI authoring application is:

- Build a structured representation of the statement being authored
- Allow iterative modifications until user done
- Visit nodes of tree to generate the syntactic representation of the statement
- Store the statement to database as a string

Most language processing systems based on a standard lexical analysis/parsing theory do this, including systems built on the well-known *lex* and *yacc* tools found on most computing platforms.

The important point to observe is that a constraint statement is equivalently expressed structurally or as utterances of a language; the structured expression is an instance of a model, while the syntax expression is an instance of an equivalent language. The difference is that the structured form is nearly always better for computer processing, while the syntactic form is sometimes better for human comprehension.

Here, we will take the structural approach for basic semantics, and use syntactic representation for more complex semantics. The primary justification for the structural approach is that fundamentally we

want the AM to relate formally to the ROM, and it is easier to engineer this with two related models than with a model and a language.

The general approach here will be to consider how each semantic construct in a ROM would appear in the AM.

Example Reference Object Model

In order to facilitate the discussion, we will introduce an example ROM, in the form of a simplified version of an electronic health record model, shown in FIGURE 15. However, any model could be used, and not just from health, but from any domain.

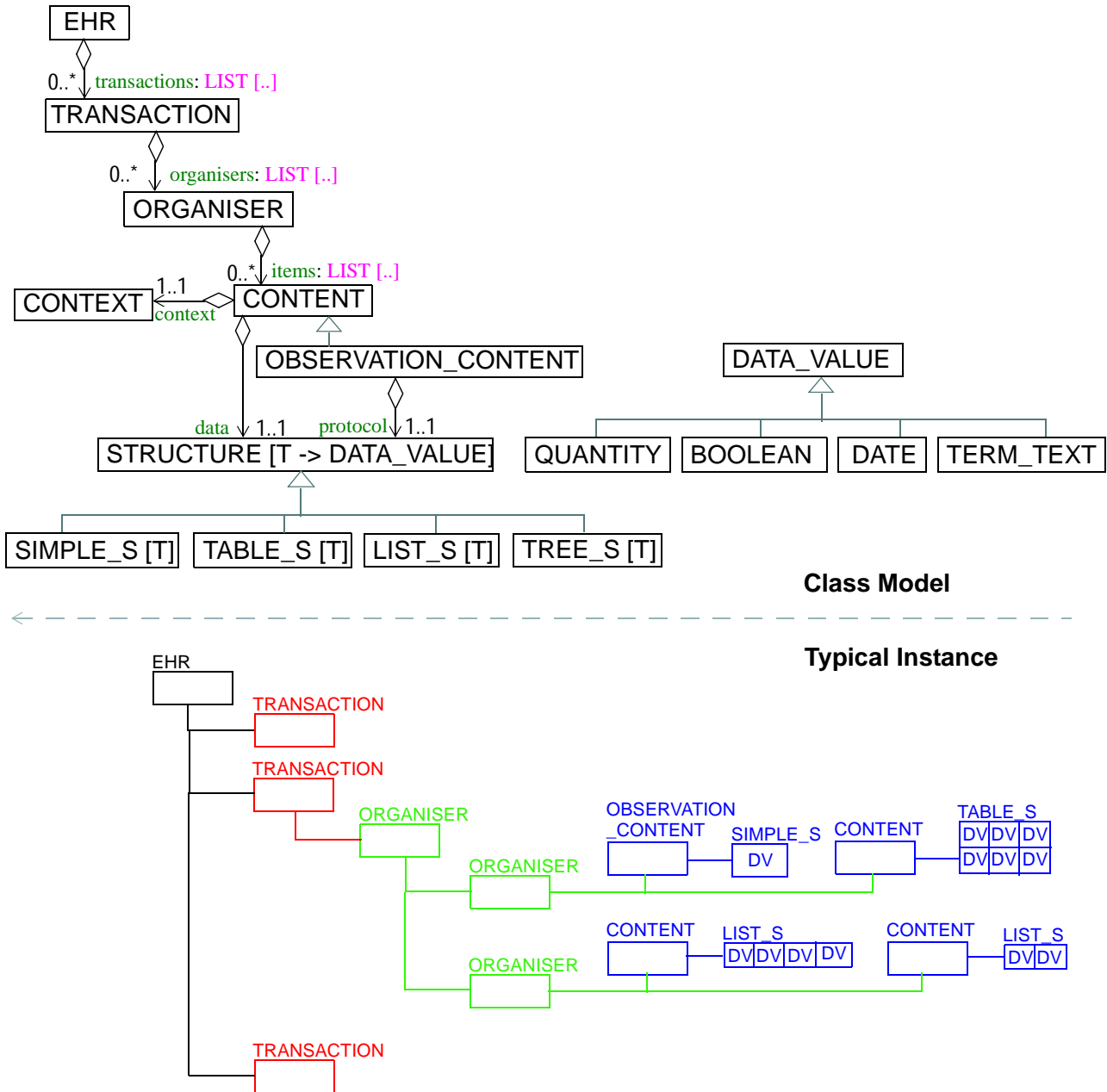


FIGURE 15 The GEHR Reference Model (simplified)

The model illustrated does not include all classes in the actual model; in particular, only one of the subtypes of the CONTENT class is shown, for the purposes of examples developed in the following dis-

cussion. The lower half of the figure illustrates a typical instance, in order to give some idea of the compositional shape of data generated by the model.

Types

We will firstly consider types in the reference model. Any object model consists of classes which define types. Designers often mentally categorise types informally into various groups, namely: “basic” types - either available in the formalism, or else simple types defined in the model; “container” types - generic types (“template” types in C++) such as lists and arrays whose job is to contain objects of other types; and “constructed” types, i.e. the important complex types of the model. We can now propose a way to construct an archetype model in terms of each of these groups of types.

Basic Types

Consider the following simple example. Assume that in the ROM, the basic type `QUANTITY` (a level-0 type) is defined. FIGURE 16 shows a class for `QUANTITY`, along with a class called `A_QUANTITY`, the archetype model correspondent. The basic types `REAL` and `STRING` in the ROM also have archetype model counterparts `A_REAL` and `A_STRING`. The `A_RELATIONSHIP` class in the archetype model corresponds to a 0:1 relationship in the ROM, to constrain the optionality of relationships between non-basic types.

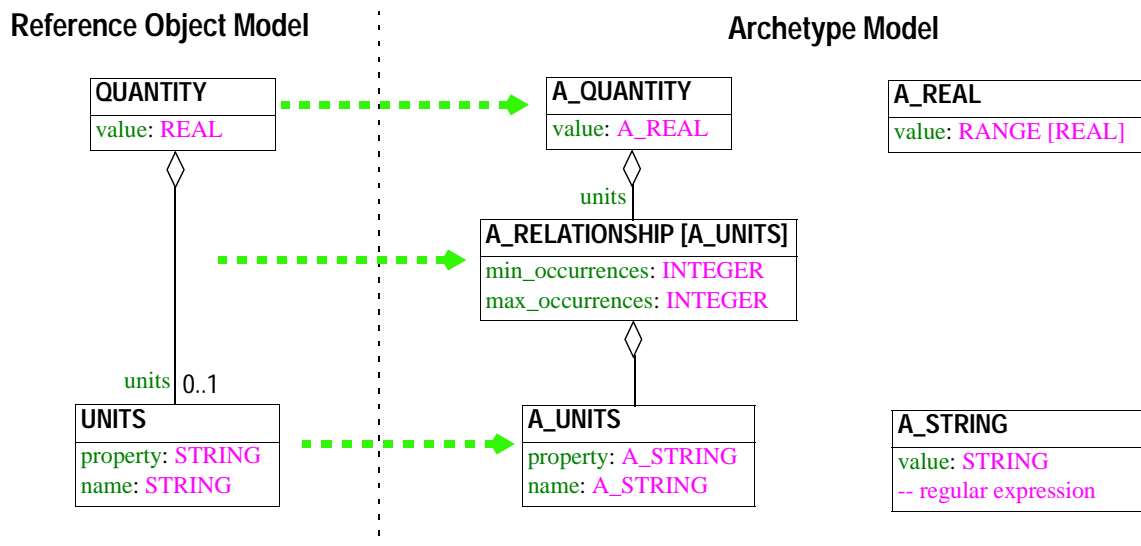


FIGURE 16 QUANTITY and A_QUANTITY

An instance of `QUANTITY` is a datum such as 110 mm[Hg] (`units.property = "pressure"`), as we would expect to find in a normal information system. An instance of `A_QUANTITY` has the *semantics of constraint* on instances of `QUANTITY`, such as:

- `0 <= value <= 500, units.name = "mm[Hg]"` {allow any pressure between 0 mm[Hg] and 500 mm[Hg]}
- `value >= 0, units.property = "length"` {allow any positive-valued length}

We can see that the archetype model on the right-hand side of FIGURE 16 will allow these constraints to be expressed. `A_REAL` is modelled as a range of `REAL`, allowing the range 0 - 500 to be expressed (a better model would probably be `LIST [RANGE [REAL]]`, allowing disjoint segments of the `REAL` domain); `A_STRING` is modelled using a regular expression, enabling constraints such as `"km/h | m\.s-1 | mph"` to be expressed (this could also be modelled in other ways).

In general, reference object models will contain a number of basic types, including both the formalism's own basic types, e.g. `STRING`, `INTEGER`, `BOOLEAN`, `REAL` etc, and the basic types introduced in the model. In health, as in other domains, the latter typically include

- Time types, e.g. `DATE`, `TIME`, `DATE_TIME`, `DURATION` etc
- Quantity types, e.g. `QUANTITY`, `QUANTITY_RATIO`, `QUANTITY_RANGE`
- Money or currency types
- Special text types. In the health arena, these include types representing plain text, coded terms, and paragraphs.
- Internet-related types: various kinds of URI types including hypertext links, email addresses and so on.
- Multi-media types, for images (including medical images), sound and movies.

Each of these will have a corresponding type in the archetype model. The actual definition of each archetype model basic type is up to system designers - it will depend on exactly what constraints need to be expressed on the leaf data types in the system. Thus, as implied above, the model of `A_REAL` might be as simple as a `RANGE` of two `REALS`, or it might require a `LIST` of `REAL RANGES`, or it might require something else altogether. In general, *the base types of the archetype model will be custom engineered.*

1:1 and 0:1 Relationships

The attribute `QUANTITY.units` is a reference in object-oriented terms, and can be understood as a 0:1 relationship in entity-relationship terms. It is indicated as 0:1 in the UML model above, meaning that `QUANTITY.units` may be Void or non-Void. Accordingly, the `A_RELATIONSHIP[T]` class appears in the AM, enabling constraints on 0:1 relationships to be expressed, as follows:

- Mandatory existence - occurrences = 1..1
- Optional existence - occurrences = 0..1
- Mandatory non-existence - occurrences = 0..0

If 1:1 multiplicity had been given in the reference model, `A_RELATIONSHIP` can be omitted from the archetype model, since there are no further constraints which can be expressed on a 1:1 relationship: it must always be there.

Special Basic Types - Dynamic Variable Types

Recall the state variable example described earlier. Where a `INTEGER` or `STRING` (for example) status attribute appears in a ROM class, representing the state of a process, a state/event table can be used in archetypes. In order to make this work on the basis of types, a `STATE_VARIABLE` type could be used in the ROM, differentiating it from normal `INTEGER` or `STRING` fields; accordingly, objects of the archetype model type `A_STATE_VARIABLE` in the archetype model will define state/event tables, and each archetype referencing the ROM attribute will encode a *particular* state/event table.

The same argument holds for any attribute with interesting or complex behaviour: the attribute will appear in the ROM as a distinct type, and the constraint form of the type describing its dynamics will appear in the archetype model. Additionally, *each archetype, being an instance of the archetype model, will be able to encode a unique constraint specification for ROM instances*, such as state/event machine specifications.

Container Types

In most domains, information is structured in various ways, for example in lists, tables, hierarchies, dictionaries, and time series. Some of these types may be available from the modelling formalism

(types such as List, Array, Bag, Set are typical), others from the standard libraries of programming languages (such as trees, hash tables and so on).

Still others will be defined in the ROM itself - these will be the specialist container types of the system; in health (and science in general) types such as `TIME_SERIES` and `MATRIX` may be required.

FIGURE 17 illustrates a part of a ROM and its corresponding archetype model, which define the low level hierarchical structure used to implement the `STRUCTURE` class in the example health record model.

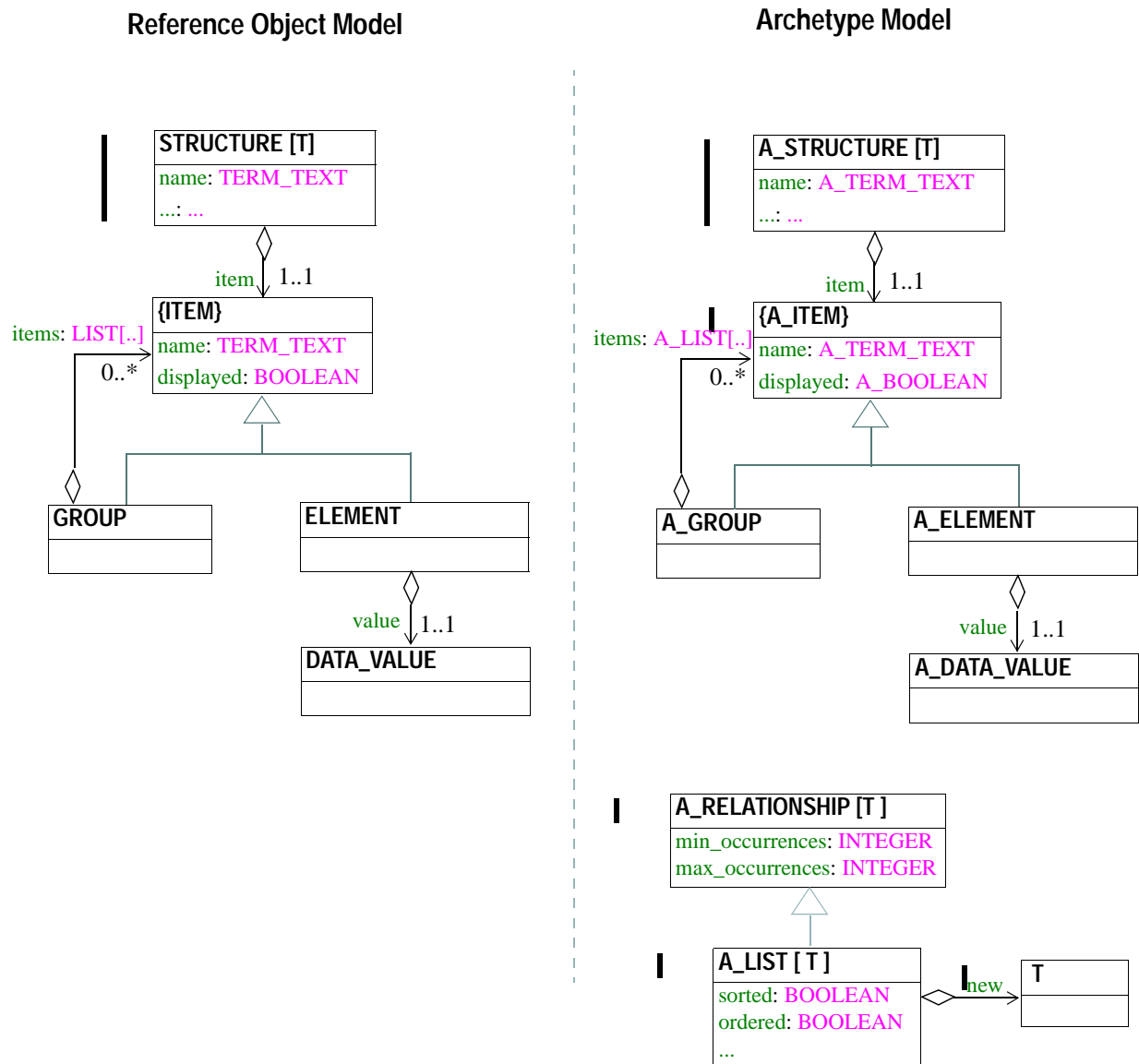


FIGURE 17 Structure Archetype Model

1:N Relationships

In the reference model, the reference `GROUP.items` is modelled using a linear container, namely `LIST[]`, which may be equivalently understood in entity-relationship terms as a 1:N relationship. Correspondingly, the type `A_LIST[]` appears in the archetype model as the type of `A_GROUP.items`. `A_LIST[]` is a type which allows constraints on 1:N relationships formed using `LISTs` to be expressed; it inherits from `A_RELATIONSHIP[]` since it is a specialisation of this class. Classes such as `A_LIST[]`, `A_TABLE[]` etc also add other semantics, including sortedness, ordering, column-naming

and so on. Table 2 describes a number of ROM container types, along with their archetype model counterparts.

ROM class	archetype model class
LIST [T]	A_LIST [T]- constrains: <ul style="list-style-type: none"> • number of items (range) • types and values of members (including empty members) • ordered / not ordered • sorted / not sorted
TABLE [T]	A_TABLE [T] - constrains: <ul style="list-style-type: none"> • number of columns (range) • column names (patterns) • column types and values • column ordered / not ordered • number of rows (range)
TIME_SERIES [T]	A_TIME_SERIES [T] - constrains: <ul style="list-style-type: none"> • number of data-points (range) • time-point values; regular or irregular; min/max number • min/max time-point interval • data-point types and values

Table 2 ROM and archetype model classes for container types

Archetypes (again, *instances* of the archetype model) can now include instances of the types A_LIST, A_TABLE etc, to express both predefined aspects of instance data structures for the archetype concept (e.g. actual values) as well as constraints (e.g. ordering, sorting, allowed types). For example, in clinical medicine, the data for almost any biochemistry test will be in the form of tables, with specifically named columns, and column values of particular types. Archetypes expressing such tests can each include distinct instances of A_TABLE to define the result tables for each type of test.

As for basic types, the exact definition of container types in the archetype model will be a question of design in the particular system being developed, so they may also need to be custom engineered.

Constructed Types & Automatic Class Generation

Basic types and containers form the “bits and pieces” from which all other types in the reference model are expressed. Any other type is constructed recursively from other constructed types, containers, and eventually, basic types. How are we to create the equivalents of such types in the archetype model? Although constructed types are more complex, their constraint analogues can in fact be *generated automatically*, assuming the reference object model formalism is systematic. Consider how powerful a possibility this is: once basic and container types have been engineered in the archetype model, the rest of the model can be regenerated at any time.

The basis of the transformation algorithm is to traverse the ROM, generating A_* classes as follows:

- Where a non-basic type T appears in the ROM, the archetype model will include a type A_T whose meaning is “constraint definition for T”.

- Where a relationship appears, the archetype model will contain a “relationship constraint” type, whose semantics define the allowed multiplicity of the relationship. For 0:1 relationships the class `A_RELATIONSHIP[T]` will be used. A 1:1 relationship, will be replicated as a 1:1 relationship in the AM, since there are no alternatives to 1:1 multiplicity. In the case of 1:N relationships, the appropriate `A_LIST[T]`, `A_SET[T]` etc container classes will be used depending on the ROM.
- Where a basic type appears in the ROM, a custom-defined basic type will appear in the archetype model.
- Where inheritance occurs, replicate it.

The resulting archetype model will be very nearly isomorphic to the ROM, where the existence of the special relationship class `A_RELATIONSHIP[T]` is the exception. The archetype model can be thought of as a “constraint-transform” of the underlying ROM. This model constitutes the language of archetypes, and archetypes themselves are utterances in the language, or in computing terms, instances of the model.

FIGURE 18 shows the archetype model for the reference object model shown earlier.

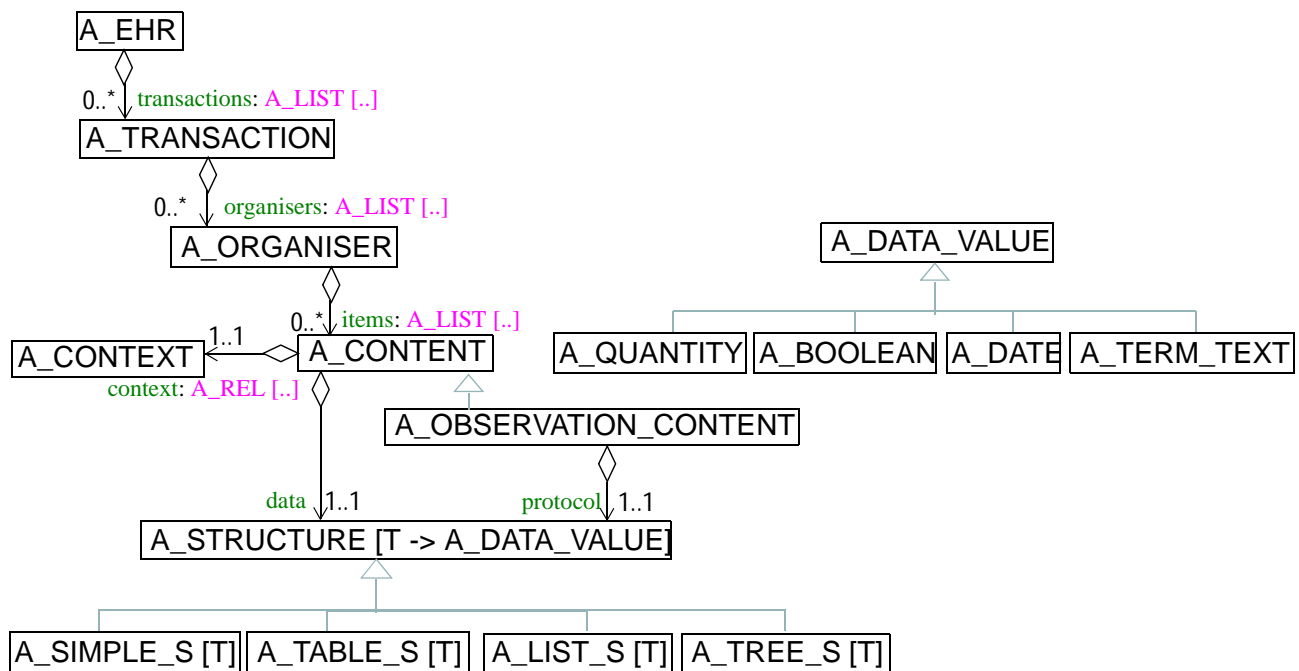


FIGURE 18 GEHR Archetype Model (simplified)

The General Archetype Model

So far we have seen that the archetype model consists of a collection of “A_” classes parallel to those in the ROM. Each of these classes defines constraints on instances of the classes of its ROM counterpart. The above example shows what an actual archetype looks like. However, there are some general aspects which require further consideration, necessitating the addition of a few abstract classes to both the ROM and the AM. To understand these classes, we must return to the notion of ontological levels.

Ontological Levels in the ROM and Archetype Models

It was suggested that in the clinical medicine domain, ontologies would exist at the *principles*, *content*, *organisation*, *storage* and *communications* levels. Data must therefore exist at these levels in

actual clinical systems; by implication, the reference object model must contain the classes of which such data are instances. For each level, there must be at least one ROM class. Using the EHR example, the classes `CONTENT`, `ORGANISER` and `TRANSACTION` and `EHR` are defined in the ROM, each corresponding to a distinct ontological level (in this model, “transactions” are the unit of storage). Of course there are other adjunct classes which flesh out the details of content, organiser and transaction objects - but the classes mentioned define the root objects of data at each level. Thus, a “blood pressure” datum is actually an instance of `OBSERVATION_CONTENT` (a subtype of `CONTENT`) and its associated classes.

We can therefore say that such root classes are “archetyped”, and we can model this by making them inherit from a class `ARCHETYPED`. Accordingly in the archetype model, there will be a class `ARCHETYPE`, which from which the corresponding key `A_` classes inherit. Thus, the archetype for a blood pressure is an (indirect) instance of the class `A_CONTENT`, which now inherits from `ARCHETYPE`. These relations are shown in FIGURE 19.

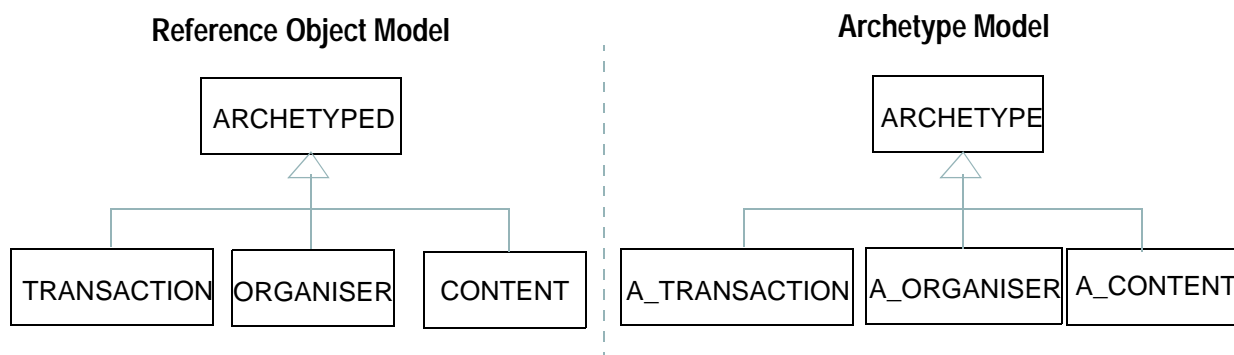


FIGURE 19 ARCHETYPED and ARCHETYPE Classes

In general, we can say that:

- In the ROM, there should be a “root” class at each ontological level, each of which inherits from the special ROM class `ARCHETYPED`.
- In the archetype model, the correspondents of the ROM root classes will inherit from the class `ARCHETYPE`.

What is the design of the abstract classes of the archetype model? FIGURE 19 illustrates some possibilities. Specific `A_` classes are implied by the inheritance relations on the right-hand side.

Archetype Fragments

Archetypes are shown as consisting of *archetype fragments*. This relationship is a generalisation of composition relationships which would normally occur in specific archetype relationships (i.e. between particular `A_` classes), and is potentially useful for archetype tools, allowing re-use of archetype pieces. For example, in a blood pressure archetype, diastolic and systolic elements may be archetype fragments, but they are not archetypes in their own right.

Naming and Archetype Paths

Archetypes are, in their most basic sense, recipes for the compositional possibilities of domain information. It is always useful to be able to navigate through information using paths constructed from identifiers of each item; we do this with directory systems, document headings and many other constructs. A naming system for archetype items would allow paths to be constructed, in order to refer to particular sub-compositions or leaf items.

The question is how to identify archetype items. The simplest way to name things would be to add to the `ARCHETYPE_FRAGMENT` class a `STRING name` attribute, in the same way as files and directories

<some content here>
...
"etc"

In this heading structure, the top level headings are named “diabetes”, “back” and so on - these are the particular problems the clinician is recording information for. The value of the *meaning* attribute in an instance of `A_ORGANISER` in the archetype is “problem”, while the values of the name attributes in instances of `ORGANISER` in data created at runtime are “diabetes”, “back” etc. If the *meaning* values are written into the data instances, we can see how data can easily be re-attached to its generating archetype structure.

Relationships Between Archetypes

Three possible relationships are shown between archetypes. The first is the *parent* relationship, representing the relationship between a specialised archetype and its parent. Specialised archetypes can only have one parent, but more general archetypes may have many children. A *children* relationship is not explicitly modelled, since it would be undesirable to have to maintain a list of children in each archetype; this is more properly the function of an archetype library. (Specialisation is discussed in detail in Archetype Specialisation on page 51).

Archetypes which are related as new versions of earlier ones are shown as the *parent_version* relationship. (Archetype versioning is explained in The New-version Relationship on page 53).

Composition

Composition is shown as a pseudo-relationship, (the dotted line *sub_archetypes*). In reality, archetype composition is expressed as pattern on archetype identifiers, or perhaps something more sophisticated.

Composition of archetypes from other archetypes was mentioned earlier, in the sense of defining domain concepts from higher ontological levels in terms of those at lower levels. What is the meaning of this idea in terms of the reference and archetype models? As we have said, if the domain modelling process (and its tools) include the notion of distinct ontological levels, such as “organising” and “content”, then the archetype model must formally model these levels, as well as a way to join instances from different levels. Further, instance data (i.e. actual information) will occur at the same distinct levels, implying that the class definitions in the ROM must exist along the same ontological levels; put another way, the major “business classes” of the ROM will exist at distinct ontological levels.

What we want to do here is to allow archetypes to be defined at different levels, and to be combined at runtime. However, only some combinations are legal. In the example given in Composition on page 29, the “family subject history” content concept can only be used under a “family history” organiser concept. Similarly, it makes no sense to allow “blood pressure” inside “ECG results”.

In general, composition between archetypes enables parts of the final information to be created according to different hierarchical combinations of archetypes, rather than just one monolithic archetype. *This vastly reduces the number of archetypes required to produce a very large number of information instances.*

The composition of archetypes can be formally represented in a number of ways. The simplest is for one archetype to refer to another by using its identifier. If structural, semantically meaningful identifiers are used, a more generalised approach is to use a regular expression for identifiers, i.e. to specify with a pattern the identifiers of archetypes allowed in a particular place in a referencing archetype.

More sophisticated references are also possible, expressed not just in terms of constraints on identifiers, but as constraints on the content of a referenced archetype. For example, under the “genetic relations” heading in the organiser archetype “family history”, we might want to say that the allowed

archetypes must be “subject family history” where the allowed values of the “relation to subject” item are terms representing genetic relations, e.g. “mother”, “brother” and so on. We can reference the identifier for the “subject family history” archetype easily enough, but expressing the other constraint requires a formal statement about the allowed values of the “relationship to patient” item in the “subject family history”.

Exactly what form of reference composition will take is likely to be dependent on the domain and context of information use.

Runtime Archetype Validation of Data

Each `ARCHETYPE_FRAGMENT` instance implements a function `is_valid`, taking as an argument an instance of the corresponding ROM type. So for example, the class `A_QUANTITY` would implement the function:

```
is_valid(q:QUANTITY):BOOLEAN
```

The `is_valid` function is the key to runtime validation by archetype objects of their ROM instance counterparts: an entire network of ROM instances representing say, biochemistry test results, can be validated by its generating archetype via the use of this function at each object in the archetype. If these functions are called during data construction, it is possible to ensure that *archetype-invalid data can never be built*.

An Example

We are now ready to see what an archetype looks like. We simply need to remember that an archetype is the following:

- A model of a domain concept
- An instance of the archetype model

A simple example is the “blood pressure” concept, which was semi-formally illustrated in FIGURE 11 on page 26. We now want to express it as instances of the above model. We have already suggested that “blood pressure” is a “content” concept, so appears at the ontological level 1, the content level. Data representing a blood pressure observation will be instances of the ROM class `OBSERVATION_CONTENT` (and related classes), while its archetype will be an instance of the archetype model class `A_OBSERVATION_CONTENT`. Using the structure previously shown of a list of (usually) two data items, and a list of a number of “protocol” items, we can construct the archetype shown in FIGURE 21.

This particular instance network is a constraint model for instances of the ROM representing blood pressures. Some of the constraints expressed:

- `OBSERVATION_CONTENT.data` must be of type `LIST_S[T]`, rather than any other descendant of `STRUCTURE[T]`.
- The list must include at least two elements whose meanings are “systolic blood pressure” and “diastolic blood pressure”; it may also include any number of other elements whose meaning is “new item”.
- The element whose meaning is “systolic blood pressure” has a name which may be any one of a number of SNOMED terms with the same meaning (i.e. allowing variations on the text). Its value must be of type `QUANTITY` (and no other `DATA_VALUE` descendant) and its value must be a `REAL` between 0. and 500.0.

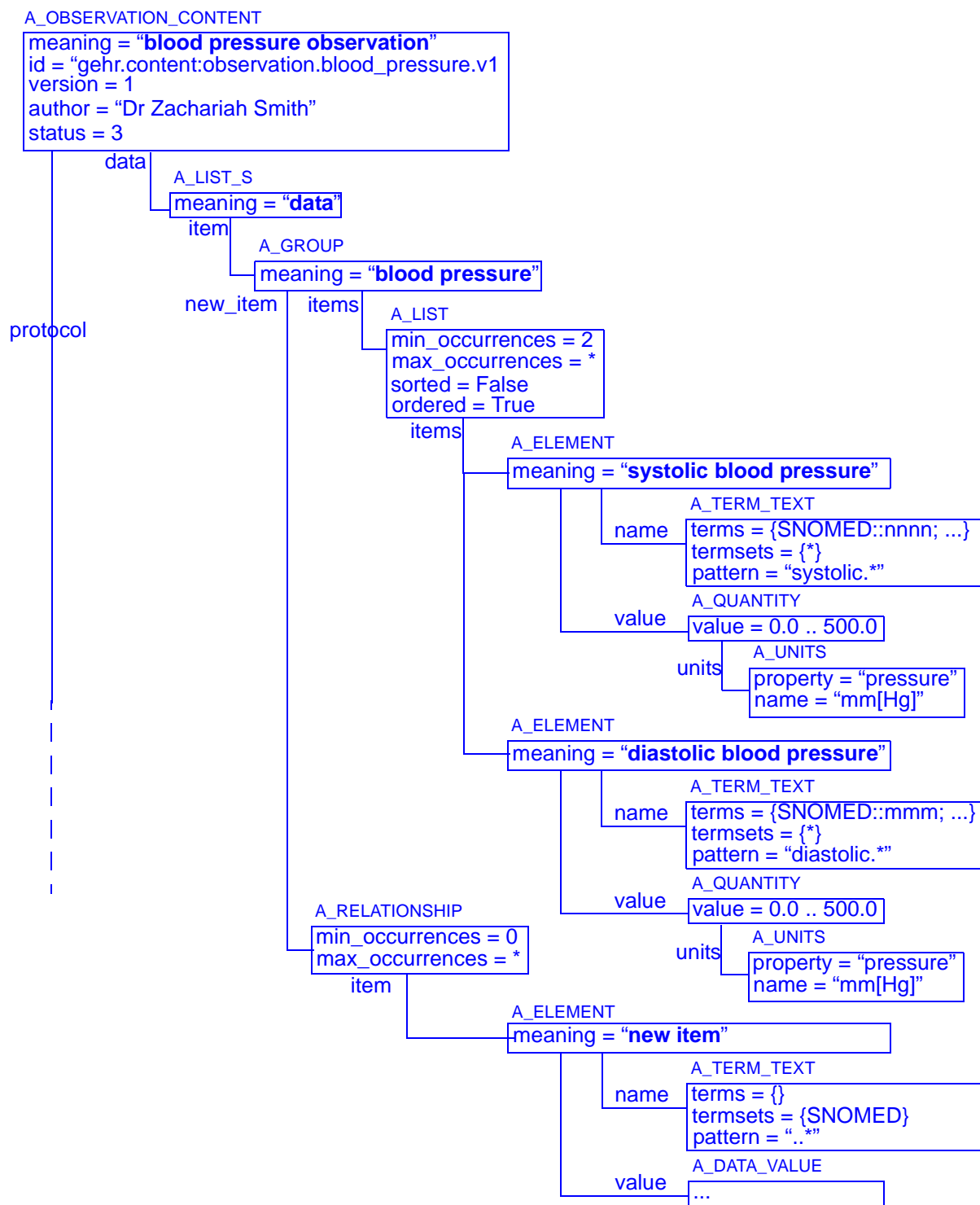


FIGURE 21 Archetype representing “blood pressure”

These are just a few of the constraints. The protocol part is not shown due to space, but contains the same kind of constraints. Although the structure is shown in all its detail, we can see that it is in fact a model of the domain concept “blood pressure observation”: it describes what data instances would be counted as blood pressures. When displayed on a screen with a GUI editor, the structure would of course be presented in a way directly comprehensible to clinical users, with unwanted detail hidden.

In this example, we can also see archetype paths, constructed from concatenations of the *meaning* attribute from the `ARCHETYPE_FRAGMENT` class, for example:

- “blood pressure observation” / “data” / “blood pressure” / “systolic blood pressure”
- “blood pressure observation” / “protocol” / “blood pressure protocol” / “position”

The purpose of such paths is to make every part of the archetype composition locatable with a simple text path string. There are a number of reasons to want to do this. One particular use is during data creation: archetype paths can be written into the data, enabling it to be later reattached to its archetypes, and also inspected by using paths derived from archetypes (rather than brute force searching for particular attributes). Other uses relate to querying, and are discussed in more detail in Archetype-based Querying on page 57.

Class-level Archetype (Local) Rules

Although the archetype model described can be very powerful, there are certain kinds of constraints it cannot conveniently express.

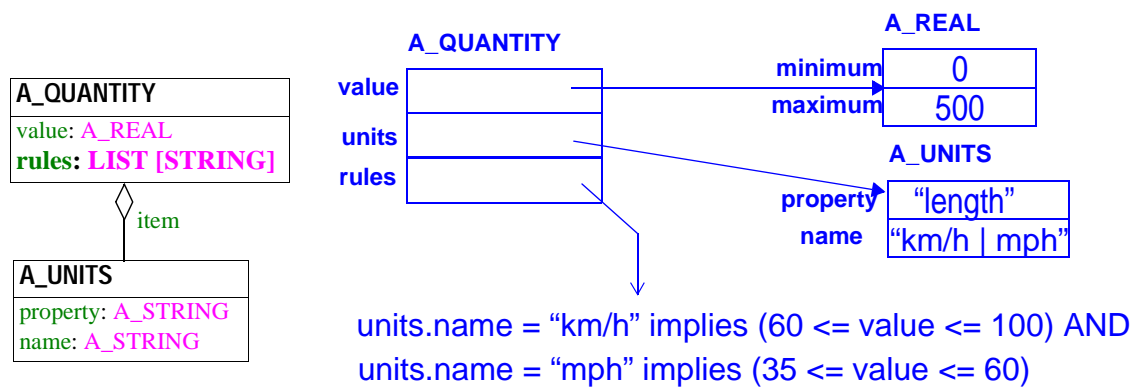


FIGURE 22 Class-level Archetype Rules

Let us return to the `QUANTITY` example. The model of `A_QUANTITY` shown earlier is not comprehensive, since it does not cleanly account for co-varying constraints, such as:

- `units name = "km/h | mph"; if units name = "km/h" then (60 <= value <= 100); if units name = "mph" then (35 <= value <= 60)`

To Be Continued: `XXX good clinical example needed XXXX`

This type of validation requirement is quite common in clinical medicine, and is an example of a constraint which is easier for humans to understand in syntactic than structural form. A possible syntactic expression is illustrated in **FIGURE 22**. To support this scheme, we need to add the ability to have rules in archetype classes. Actual instances of classes in archetypes will contain rule texts. A run-time rule-processor is required to process the rules.

These rules are denoted “local” rules because they do not refer to any attribute outside the class, nor any external variables (such as “patient’s weight” for example).

Local rules may mostly appear in the custom-engineered base and container types of the archetype model, but there is no *a priori* reason why constructed classes should not have such rules. This means that if the archetype model is automatically generated (or if it is updated by hand), rules in earlier versions of generated types will need to be kept intact.

FIGURE 23 shows the addition of local rules to the archetype model.

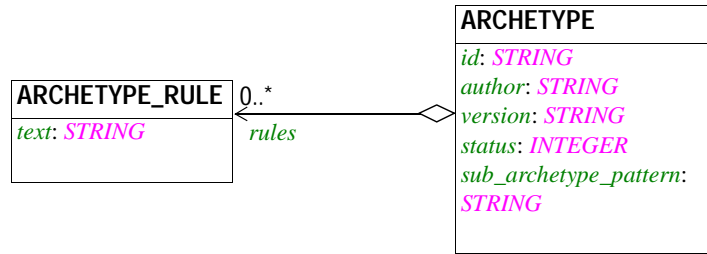


FIGURE 23 Archetype Rules in the Archetype Model

The Big Picture

Let us step back for a moment, in order to gain some perspective on the modelling approach for systems being proposed. FIGURE 24 illustrates the relationships between the reference object model, the archetype model, and their respective instances.

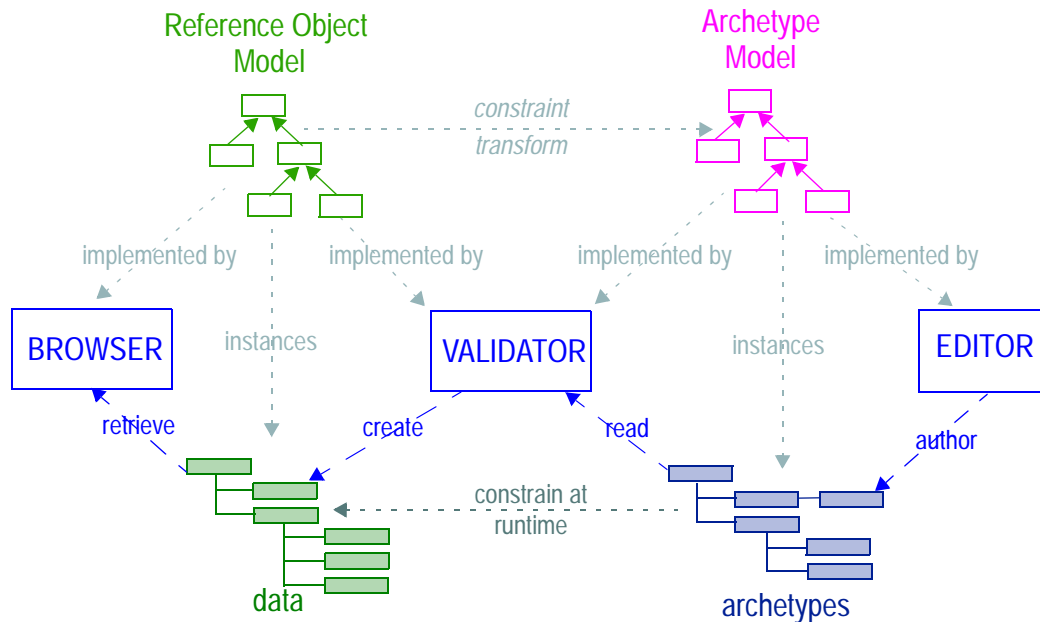


FIGURE 24 Relationship between Reference and Archetype models and their Instances

In this diagram, several pieces of software are shown, being based on the ROM and/or archetype models. These are:

Archetype Editor: a GUI application for creating new archetypes. This is based on the AM class model.

Validator: a component whose function is to create valid data, using archetypes. This is based on the ROM and AM classes.

Browser: a generic browser can be built, based solely on the ROM, although a smarter browser can be built using the AM as well.

Probably the most important property of systems based on this scheme is that instance data (shown at the bottom left) are not only technically conformant to the ROM (as per the usual object-oriented class/instance relation), but are also conformant to the constraining archetype instance (bottom right). That is, they are both *valid ROM instances*, and *logical instances of domain models*. Further, the variability expressed in archetype constraints enables more than one data instance to be identified as instances of the same archetype (the “constellation of instances” effect described earlier).

This approach is homologous to approaches in which a formal language (e.g. object-Z) is used to write concept specifications; here the archetype model is semantically equivalent to such a language. However the strength of this approach is that archetypes are *instances* in an object-oriented system implementation: they can be created and manipulated by GUI tools, altered as desired without ever changing database schemas, or the ROM or archetype models.

The constraint relationship between the ROM and the archetype model is a new kind of formal relationship between models, and is not typically treated in the object-oriented literature. However, it is not technically difficult to devise such a relationship, and it has been implemented in the GEHR [14.] and SynEx [24.] projects .

Alternative Approaches

The archetype model development approach described here is somewhat reductionist, but quite convenient and comprehensible, given that the model it generates is isomorphic to its reference model. Another approach to devising an archetype model is to completely custom design it, without using parallel classes as used above.

In doing this, the important principle to conserve is that of the ontological levels: the archetype model will at least need to contain classes corresponding to the root class at each level in the reference object model.

If we reflect on the nature of both the custom-designed and “systematic” archetype modelling approaches, we realise that they are the same approach, with a greater or lesser degree of custom classes. That is, at its *least* systematic, an archetype model must at least have classes corresponding to the root classes at each ontological level in the ROM; at its most systematic, it has these and more, down to a lower point where custom-engineered classes once again take over.

There appears to be no *a priori* way to know if a more custom-designed archetype model is better, and it may be a matter of particular circumstances. In any case, the more systematic model is at least (more) generally describable, and its production can also be automated, so we will concentrate on it here.

Archetype Identification

The design of the archetype namespace is of crucial importance since it reflects the domain space in a formalised way. It is also required before any scheme of archetype specialisation and composition can be devised.

One way of understanding the problem of identification is to pose the question: when are archetypes different? There are typically a number of dimensions in the domain space which provide an answer. The following dimensions are probably found in all domains, and are not necessarily exhaustive:

- Issuing authority: organisation accrediting archetypes for operational use in the domain.
- Reference object model on which the archetype is based.
- Ontological level in the reference object model.
- Thematic: the concept name, including specialisations.
- Use context: general or specific.
- Time-related versions.

An identification scheme based on these dimensions leads to identifiers of the form:

issuing_authority . rom_id . ontological_level . concept[:specialisation] . use_context . version

Examples in health might include:

```
gehr.gehr-om.transaction.contact.any.v2
```

```
uk-nhs.gehr-om.organiser.family-history.uk.v2
```

```
nih.hl7_rim.act.biochemistry.us.v4
```

```
iama.gehr_om.content.energy:ayurvedic.any.v1
```

In fact, in health, at least one other dimension can be identified, which might need to be used in identifiers, that of “medical system”, reflecting the underlying philosophies of different medical systems, such as “western”, “ayurvedic”, and “chinese”. Since each of these systems actually represents an entire concept space of its own, and is quite likely to have some terms in common with other systems, although representing different concepts, it is a candidate for inclusion in the identifier. However, this topic is complex, and deserves proper study (consider for example the different understandings of “energy”, “mind-body interaction” and “digestion” in western, daoist and ayurvedic approaches).

Other ways of designating the space are no doubt available, such as with ISO-style OIDs. However, it will most likely be important in all domains to have identifiers which are comprehensible to humans, even if there is a machine-processable equivalent.

The Archetype Space

The identification system above formalises the structure of the archetype space for a given reference object model. Different ROMs will have distinct archetype spaces. FIGURE 25 illustrates the general form of an archetype space. The section on specialisation describes in more detail the structure of this space.

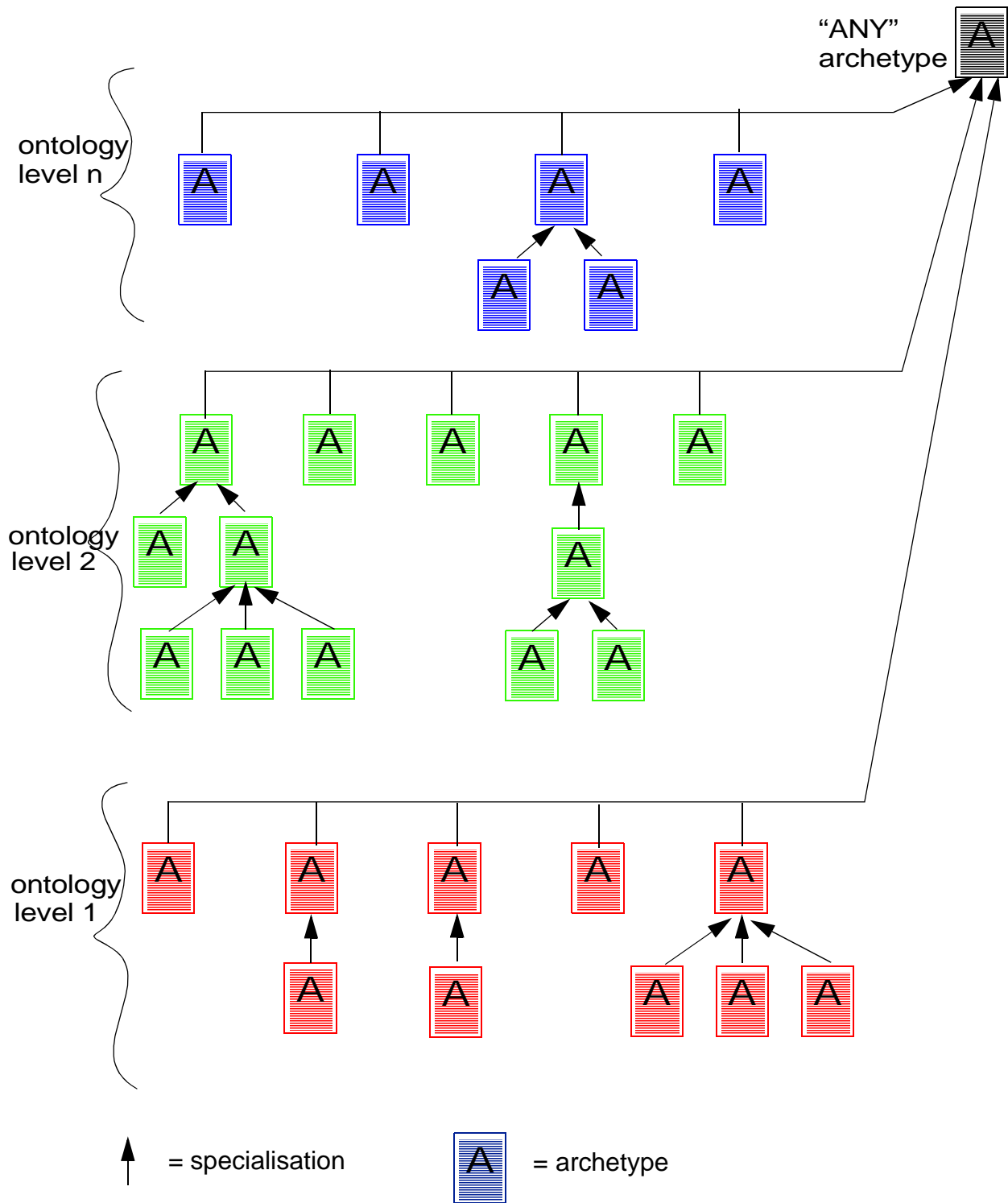


FIGURE 25 Structure of Archetype Space for a given ROM

Archetype Specialisation

As soon as an archetype is developed and released for use, there will be users who require similar archetypes, related to the original. Dimensions in which new archetypes might be created in health include:

- Localisation: particular health facilities might collect more data items than mandated by a standard archetype. Nationally or regionally legislation may require locally specialised archetypes. (But note: even standard archetypes allow for optional additions; the motivation to specialise is not to add options, but rather to further specify previously open parts of an archetype, i.e. to reduce the options).
- Other health systems, such as naturopathy, Chinese medicine, Ayurvedic medicine. In such systems, more information might also be recorded, typically qualitative information.
- Specialist disciplines: a psychiatric “family history” might well contain significant extensions to the “family history” archetype used by a GP; a diabetic referral is likely to be a specialisation of some standard model of a referral.

Technically, what constitutes allowable modifications in a specific version must be carefully considered. One important reason for this is that, at some point in time, information will be created according to both the original and the specialised archetypes. A crucial interoperability requirement (both inter-system and inter-application) is that the data created according to related archetypes should be convertible from one to the other form (typically in one direction only, but not always). Scenarios in which this occurs include conversion of data created according to older archetypes, and conversion of data requested from other systems.

To start with, we will note that there are three possible relationships between archetypes (apart from composition), as follows:

- Later versions, i.e. archetypes in which omissions or errors are corrected.
- Specialisations, i.e. archetypes which specify another archetype as a parent.
- Unrelated archetypes. If we posit a notional “any” archetype from which all other archetypes are derived, there is technically no such thing as unrelated archetypes, since all archetypes ultimately do have a common parent. But for the purposes of information modelling, this particular parent can be ignored.

In this section, we will define the specialisation and new-version relationships between archetypes, and describe what changes may be made to an archetype in going from one to the other.

The Specialisation Relationship

We will define the notion of *specialisation* as follows:

- An archetype N is a specialisation of an original archetype O if its data also conform to archetype O, i.e. its data can be considered a *special case* of O. Archetype O is then *substitutable* for N with respect to the data.
- Accordingly, archetype N may only *further constrain* the archetype O. If this were not true, data built with N might not conform to O.
- The above relations are recursively true, thus data created using N should conform to any previous parent starting from O back to the special archetype ANY. Each of these archetypes further constrains the previous one in the specialisation chain.
- The identifier is intended to be different.

The ability to specialise archetypes gives rise to an *archetype specialisation hierarchy*, somewhat reminiscent of the inheritance hierarchies in object modelling. For specialisation to work as intended, more general archetypes - those higher up the hierarchy - must be relatively unconstrained, or “open”. That is to say, they should provide structural and semantic elements which enable new data items to be created. Then more specialised versions reduce the freedom of such elements, by constraining them according to more precise models.

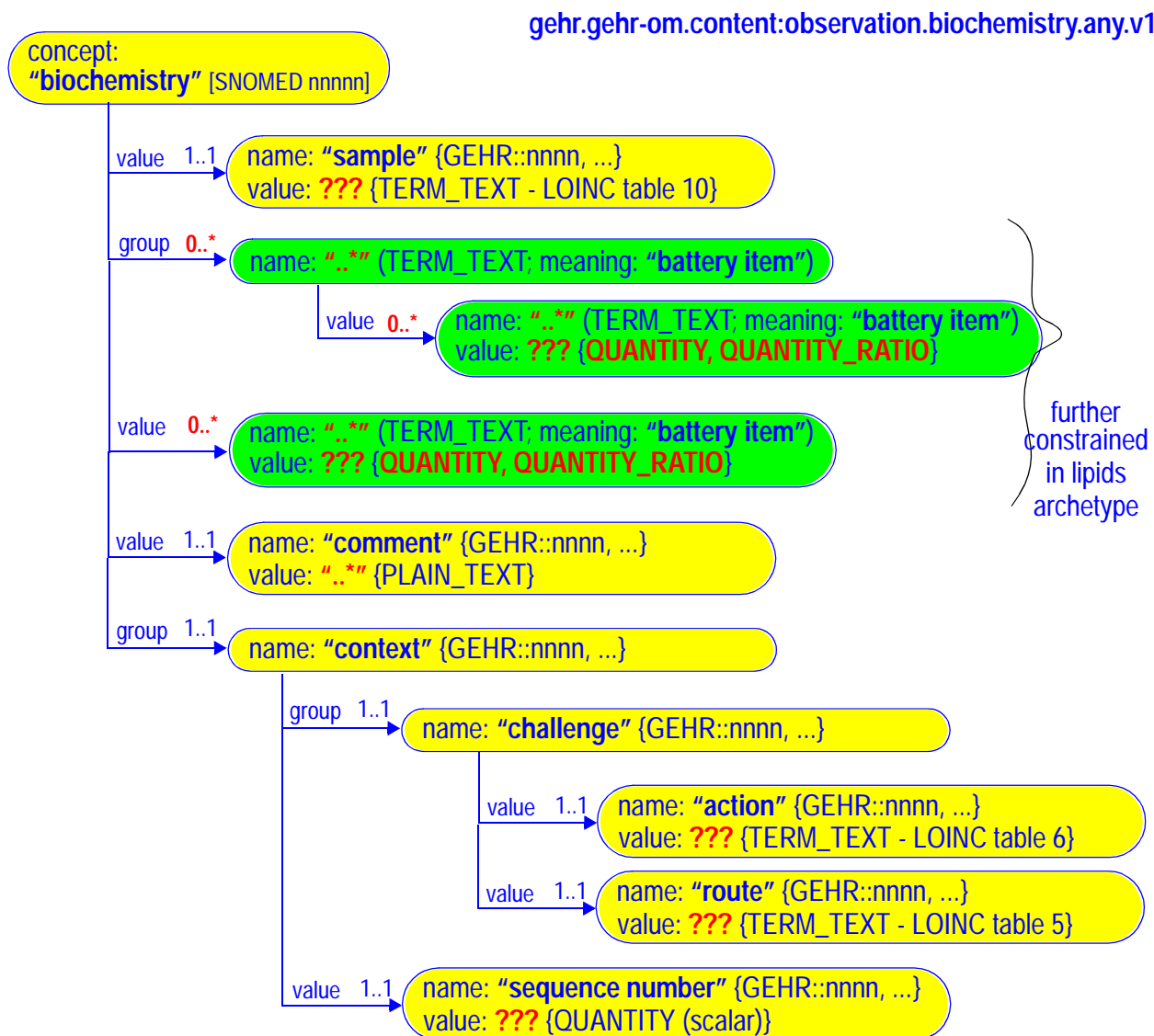


FIGURE 26 General Purpose Biochemistry Test Result Archetype

Consider the example shown in FIGURE 26, which shows a generic model of a biochemistry test: the core results are represented as 0..N batteries, each consisting of a number of battery items, which are just name/value pairs. Battery items can occur outside a named battery as well. Provision is also made for comments and a description of the test context in terms of challenge, action, route and sequence number.

More restrictive archetypes can be defined based on this one, for instance, a Blood Lipids test, as illustrated in FIGURE 27. In this model, the generic biochemistry archetype elements for battery and battery item are constrained to be those meaningful for a lipids test. Any data instance of this archetype is automatically an instance of the more general parent.

gehr.gehr-om.content:observation.biochemistry.lipids.any.v1

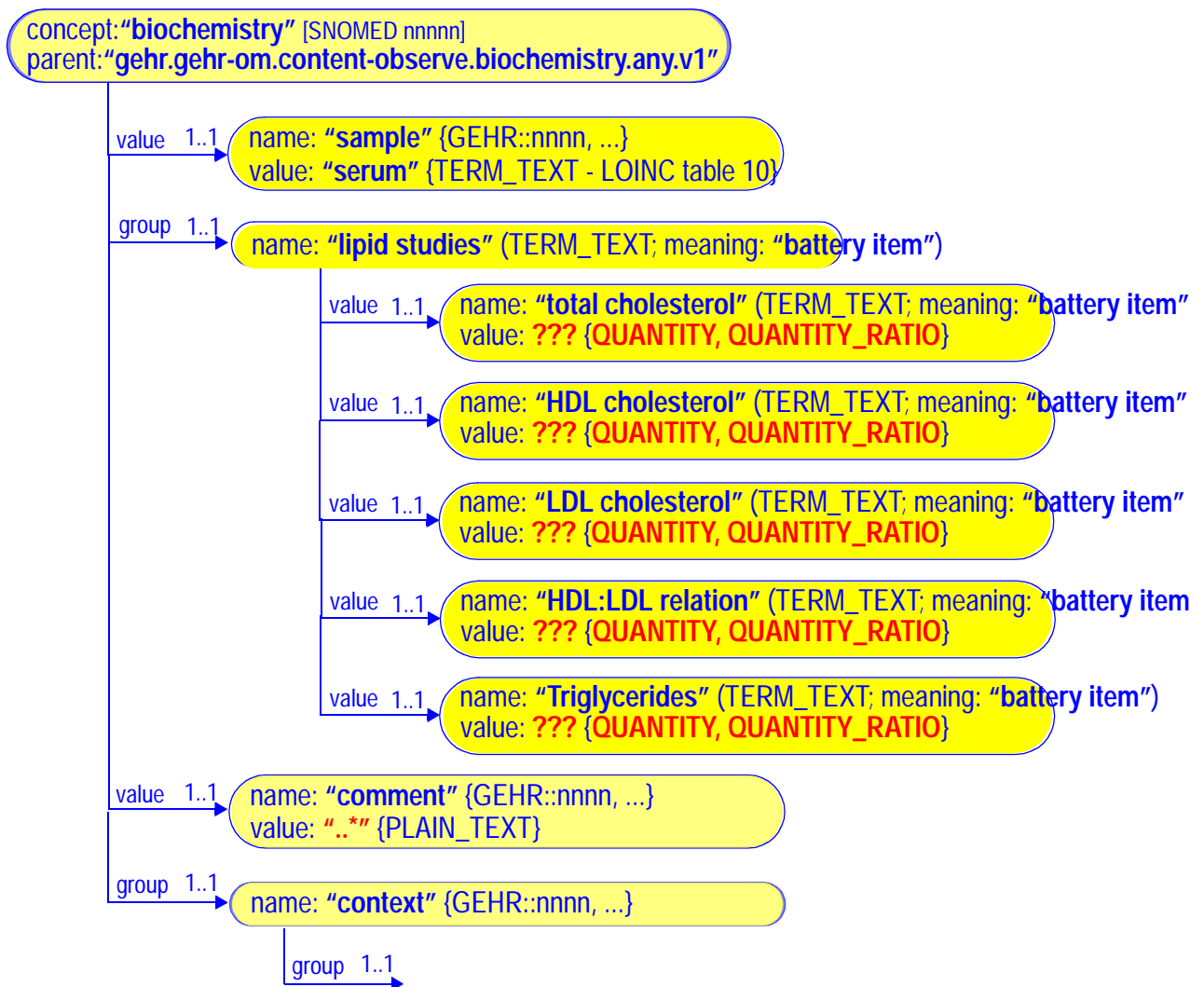


FIGURE 27 Sample Lipids Biochemistry Archetype

Specialisation gives rise to a hierarchy of archetypes, as illustrated in FIGURE 25. We can add an ultimate parent archetype called “any”, which all archetypes having no other parent can refer to as parent. This now gives us a view of all archetypes as a number of strict specialisation hierarchies, one for each of the ontological levels in the domain.

Finding the common parents of different archetypes is a key to being able to translate the data from one form to another, enabling data with local (or other) specialisations to be understood by requesting sites.

The New-version Relationship

A new version of an existing archetype is created with the intention of improving it or repairing some identified problem. The identifier is intended to remain the same (apart from the version field). *The previous version is deemed to be obsolete.* Existing data, created according to the (or in fact any) previous version, must be convertible to the latest version in a well-defined way.

New versions may be created due to changes in legislation or technology which mandate the inclusion of new items, or the renaming of existing items. It may even be possible that existing items no longer need to be collected, and can be deleted, although this is likely to be rare (a possible example is the datum “sexual orientation” which has often been collected in health systems, but is of dubious clinical value in most circumstances).

If additions, some kinds of changes and deletions are allowed, the exact rules for such changes must be described, along with the means of converting old data to the new form.

A major consideration for whether a new version should be created is the scale of the knock-on effects on existing data created according to both the original version, but also any specialisations. In theory, a new version of an archetype O implies that the archetype N, a specialisation of O, is now obsolete; consequently a new version of N should also be created, incorporating the same changes as in O.

To Be Determined: details...

Automatic Data Conversion

To have a chance of performing automatic conversions between data instances created according to related archetypes, the formal relationships between the generating archetypes must be known; from these, rules for data conversion can be inferred, hence, we will consider the possible types of technical relationship between archetypes.

As an aid to the discussion, FIGURE 28 graphically illustrates relationships between archetypes and data. In this figure, three archetypes are shown on the left, which we can think of as “original”, and two “new” archetypes. The relationships between each new archetype and the original are marked R_{1O} and R_{2O} . Instance data created with each of the archetypes is shown on the right side.

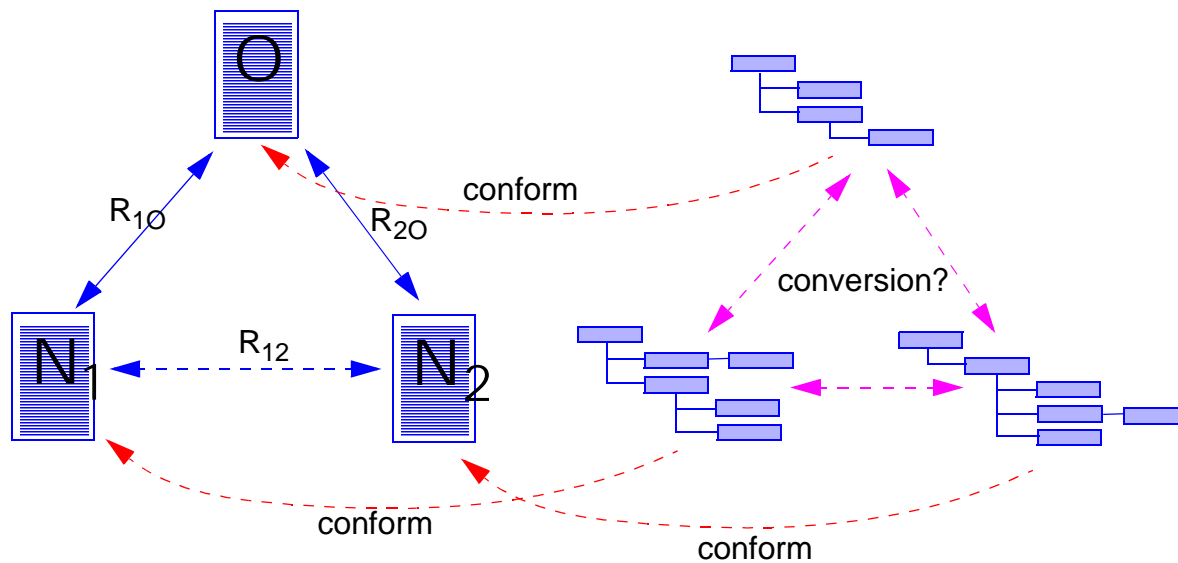


FIGURE 28 Archetype Relationships

The main question we want to answer is: what is the nature of the relationships between the archetypes O, N₁ and N₂ if we want to be able to convert data conforming to each to a form conforming to one of the others?

Possible relationships include the following.

Forward Compatibility

If data created according to an archetype N is valid when tested against an archetype O, the archetype O is *forward compatible*. Rules on N and O to achieve this are as follows:

- N cannot loosen names, types or value constraints, since some instances of its data would not conform to O.
- N cannot loosen structural constraints (including mandatory/optional) since this might result in data which were missing items considered mandatory by O.
- N may further restrict any name, type or value specification.
- N may further restrict structural specifications, i.e. require more mandatory items, or that items optional in O are mandatory.

All archetypes in the archetype specialisation hierarchy are forward compatible.

Backward Compatibility

If an archetype N is *substitutable* with respect to data created according to an archetype O, i.e. it is *backward compatible*. This implies the following:

- N cannot narrow the constraints of O in such a way as to make older data invalid with respect to N, i.e. it cannot:
 - Further restrict name matching specifications (older data might have been created with a name like “systolic blood pressure”; to now disallow this name would invalidate the older data).
 - Change any optional item to mandatory (older data may not contain items now considered mandatory).
 - Removal of any structural item (some instances of older data will contain this item).
- To Be Determined:* not sure if this is really true - could just ignore non-mapped data?
- Further restrict type-matching specifications (older archetypes might have allowed PLAIN_TEXT or TERM_TEXT items at some position in the data; a newer one restricting the field to be only TERM_TEXT will make some instances of older data invalid).
 - Further restrict value constraints (older data might contain previously legal blood pressure values of 200 mm[Hg]; this would become invalid if it were restricted to 0 - 175 mm[Hg]).
- Allowed changes include:
 - Any name, type, or value specification can be loosened, since older data will remain legal with respect to a looser specification.
 - Mandatory items in O may be optional in N. In general, the new cardinality range must be the same as the old, or else outside it. Thus the “optional” range 0..1 is outside the “mandatory” range 1..1

New *versions* of existing archetypes may in some cases be backwardly compatible.

Archetype-based Querying

A crucial function where archetypes provide significant help is in querying data. Inspection of an archetype in advance can yield a set of path fragments which can be used to query instances which conform to the archetype. Further, the map of the archetypes used in real data enables optimised querying to be implemented for some classes of queries. These possibilities are explored in the following sections.

It is in principle possible to know the possible locations of each leaf datum in information conforming to an archetype. All that is required is that the archetype support the concept of *paths*, and that every leaf item in the archetype is reachable by a unique path in terms of the archetype model. We have already included a meaning attribute in the `ARCHETYPE_FRAGMENT` class, which can be used to construct paths. To make the path concept useful, we require that descendants of the class `ARCHETYPE` must implement a function of the form:

```
item_at_path(some_path:STRING): ARCHETYPE_FRAGMENT
```

Archetype Query profile

A query profile of the entire archetype is given by a list of such items. Also, the optionality of data items is important in querying, since it enables a query engine to know whether the absence of something is an error or not.

Query Name	Archetype Path	Optional
<i>systolic blood pressure</i>	“blood pressure” / “systolic pressure”	N
<i>diastolic blood pressure</i>	“blood pressure” / “diastolic pressure”	N
<i>instrument</i>	“blood pressure” / “protocol” / “instrument”	Y
<i>cuff size</i>	“blood pressure” / “protocol” / “cuff size”	Y
<i>position</i>	“blood pressure” / “protocol” / “position”	Y

Table 3: Query Profile for Blood Pressure Archetype

Query Name	Archetype Path	Optional
<i>subjective</i>	“problem” / “subjective”	N
<i>objective</i>	“problem” / “objective”	N
<i>assessment</i>	“problem” / “assessment”	N
<i>plan</i>	“problem” / “plan”	N

Table 4: Query Profile for Problem/SOAP Organiser Archetype

The availability of named queries on archetypes is effectively a service interface for data, in which the “functions” are at the clinical knowledge level. Thus the query interface for the blood pressure and problem/SOAP headings is defined by the first column in the tables above.

The primary use of archetype query profiles is to find the archetypes containing a certain datum, for instance “LD cholesterol” as a query name, i.e. as a leaf element. Once this list is known, data can be searched very efficiently, using the archetype query map method described below.

Another use for this “flattened” interface is that it may be present in similar archetypes which have different underlying structures, enabling data to be processed as if the archetypes were the same, and potentially providing a means of converting being archetypes which are not strongly related structurally.

The addition of ARCHETYPE_QUERY_PROFILE to the archetype model is shown in FIGURE 29.

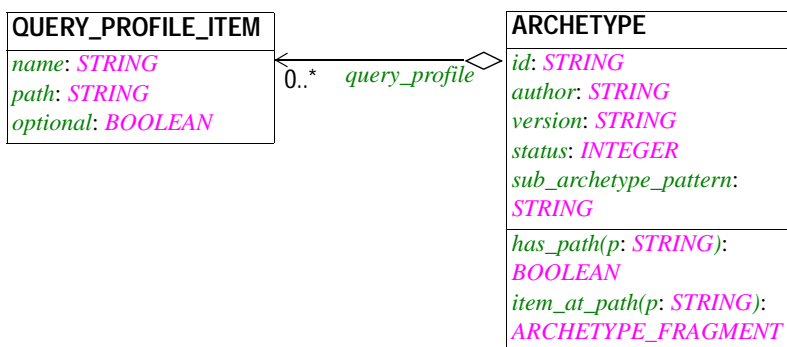


FIGURE 29 Query Profiles in the Archetype Model

Archetype Query Maps

When data is committed, its “archetype query map” can be computed, and stored separately to aid efficient querying. The map is simply a list of archetypes used to construct the data, keyed by the actual paths in the data where they were used; in addition, a flag can be used to indicate whether the item is actually present in the data (catering for archetype elements which are optional).

In most cases, relatively few archetypes are used to construct quite large slabs of data; a clinical example would be “ECG results”, where one archetype corresponds to 10 leads’ worth of time-series data, potentially hundreds of samples.

In general, the archetype map will be much smaller than its data, meaning that it is a suitable basis for optimised querying. FIGURE 30 illustrates an electronic health record transaction, consisting of test results content, organised under a number of headings, and grouped inside a transaction. Such a transaction might be quite large in terms of numbers of objects (or database column items). However, only four archetypes are used to generate the data, meaning that the archetype map (illustrated on the left) is a very small data instance indeed, consisting only of a few hundred bytes of archetype identifiers arranged as a tree (or alternatively, keyed by paths), plus the database key for its transaction.

An application wanting to query the data of the transaction can retrieve the query map, read the archetype identifiers, and use them to determine whether the data items being sought are in fact available in the data, by referring to the relevant archetype query profiles. For example, to answer the query “find the last 5 blood pressures, and retrieve their health record transactions”, an application could work as follows:

- Determine which archetypes “blood pressure” can be found in, via a query to an archetype repository.

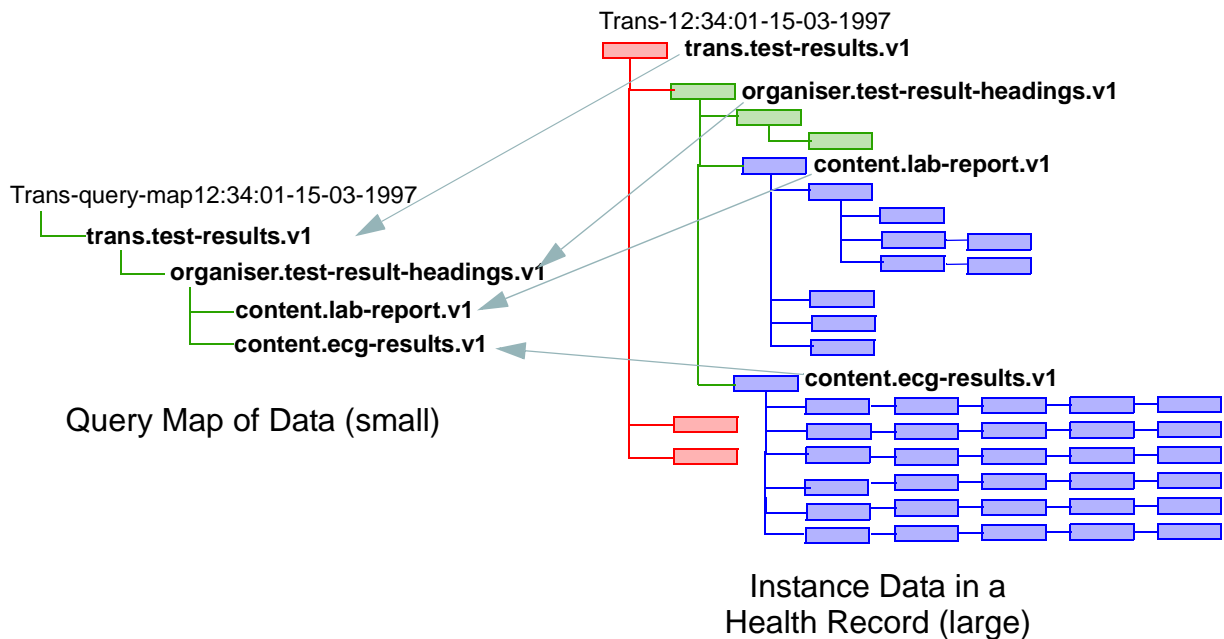


FIGURE 30 Query Map for a Health Record Transaction

- Starting in reverse date order, begin retrieving `TRANSACTION_QUERY_MAP` objects, checking for the identifiers of these archetypes.
- For each matching `TRANSACTION_QUERY_MAP` object, check the *presence* flag, and if True, add the Transaction id (stored in the query map) to a list of matched transactions, until five transactions are reached.
- Retrieve the five transactions by key.

Consider the query

To Be Determined: complex query example

Each datum referenced in the WHERE clause must be defined in an archetype somewhere, so the archetype map and query profiles can be used to determine candidate data instances which might satisfy the query; any data instance not containing the referenced data items will never be retrieved.

In general, the query map in conjunction with archetype query profiles provide the information of what items will be available in any actual data, enabling even complex queries to be served quickly. Effectively, the use of archetypes makes data “transparent” to query engines, since a lot can be known about it prior to retrieval, and without having to construct arcane indexes.

Population Queries

Many queries are designed to be executed on populations of data, for example on large numbers of health records, and to return a result consisting of those records (patients, in the case of health records) matching some criteria. Consider the queries:

- Find women of child-bearing age who are sexually active and who have had no PAP smear in the last two years.
- Find the percentage of male diabetics over 50 who are also hypertensive, and who have a family history of cardiac disease.

Both queries would normally be initially processed by a query server which would convert them to computable items. In the first case, these would be records where:

```
gender = Female AND sexually active = Y AND (date of today - date of last PAP smear > 2y OR no PAP smear entry)
```

In the above, the underlined attributes would typically appear in archetypes, and consequently, the archetype query profile. This means that it can be known in advance which transactions of the record should be looked at, just by finding which archetypes contain the attributes `gender`, `sexually active` and `date of last PAP smear`, and then reading transaction query map objects. The record transactions with a positive match for the archetype still have to be retrieved of course, in order to determine the values of each datum, e.g. `gender = Male or Female`.

This query also contains a negation, whereby records not containing a PAP smear entry (and satisfying the other criteria) need to be in the result. Negations are typically more difficult to handle, and require some intelligence in the query server. In a GEHR EHR system, for example, the server would have to know that “PAP smear” is a kind of recurring instruction (as are prescriptions and therapeutic care); it would then investigate a transaction containing the “current medications” or “current instructions”, which would typically be found in an electronic health record, in order to determine that “PAP smear” was absent.

To Be Continued:

Constructing Query Statements From Archetype Paths

To Be Continued:

Other Query Requirements

Querying is a complicated business, and the above features will not directly support all classes of query, such as free text (sometimes known as “content-based retrieval” or CBR), and some kinds of *ad hoc* query.

Queries may also be posed in a much more abstract way than supported by archetypes, and at least part of the job of any query engine would be to convert abstract queries to concrete function calls to a back-end archetype-based query processor.

To Be Continued: a lot more work required here...!

Advanced Features

More Sophisticated Validation

To Be Determined: PROVISIONAL SECTION; may be better to use NORMAL/ABNORMAL/CRITICAL in clinical rules

As described earlier, instances of archetype model classes can be used to validate instances of the related class in the Reference Object Model, by means of a fairly simple comparison algorithm. For example, an instance of `A_QUANTITY` can be used to validate an instance of `QUANTITY`, by comparing the value in the quantity to the value constraint in the `A_QUANTITY` instance. Following the example, an instance of `QUANTITY` will either “accept” or “reject”. However, in clinical medicine, and laboratory science in general, this may be too simple a response. An alternative (proposed in the UCL SynEx project [24.]) is to allow the following results:

- Accept
- Confirm
- Reject

Now we have effectively two types of “accept” - unqualified, and qualified. The use of “confirm” allows applications to deal more intelligently with instances which are outside the normal acceptance criteria. This would mean that the signature of the `is_valid` function would be as follows:

```
is_valid(an_item:SOME_ROM_TYPE):ARCHETYPE_VALIDITIES
```

where the class `ARCHETYPE_VALIDITIES` is an enumeration of the three values.

To Be Continued:

External Rules

There is another category of rules which is extremely important in most domains. In medicine, these are clinical rules, typically used for decision support. These differ from local rules in a basic way in that they do not express informational validity, but clinical “validity”.

For example, a patient’s health record may include a blood pressure datum which is “very high”. This is informationally valid (assuming the archetype allows such a value), but it may be treated by certain clinical rules as “abnormal”. Such rules are commonly used in decision support systems to combine several data in order to produce a result, such as a recommended drug administration.

Other differences between external and local rules include:

- They may refer to variables outside the archetype
- They may be expressed in a special syntax
- They require an external processor which understands the syntax
- Time-related rules which are only evaluated according to external events.

Typical rules in this category include:

- Any rule used to determine whether a datum is “normal”, “abnormal” or “critical” (or similar such categories)

- Any rule which references a datum from elsewhere in the record, or from outside the record. For example, a rule designed to determine if the patient is of normal weight will require the following data (at least):
 - Patient's height
 - Patient's sex
 - Normal, abnormal (overweight), and critical (obese) ranges for the above two items from a particular height/weight chart (this may itself be a variable, if different charts are in use.)

Currently such rules are not stored in a standard place in information systems, or associated with any particular model, but archetypes give us a convenient place to put at least some such rules. In this capacity, the archetype is simply acting as a place-holder for items which will be used by external agents.

To Be Continued:

An Advanced Archetype Model

FIGURE 31 illustrates additions to the archetype model including external rules and tri-valued validation.

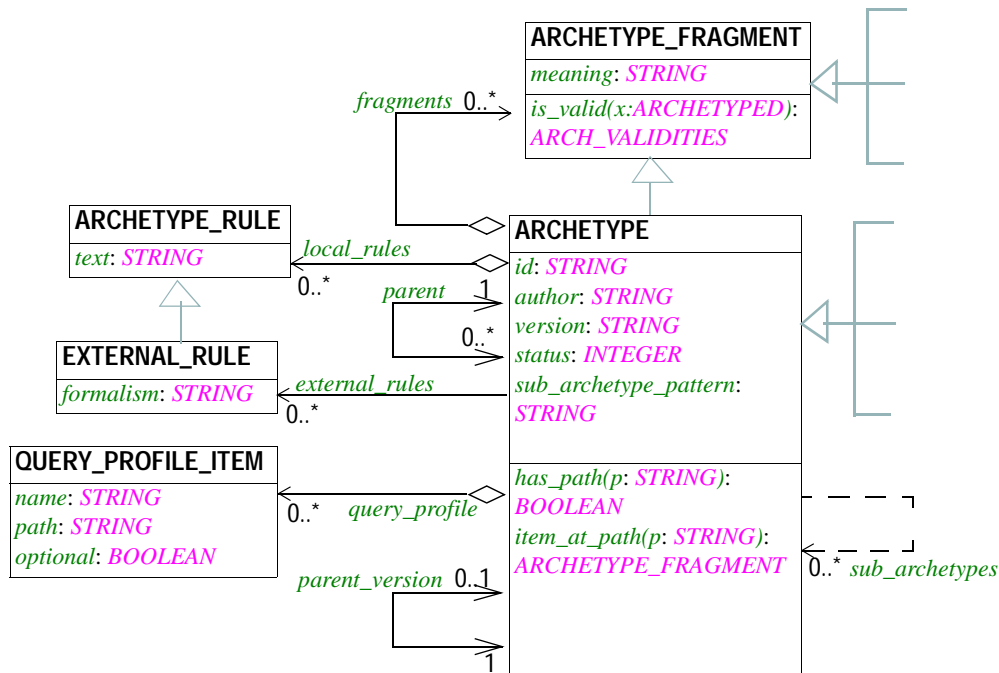


FIGURE 31 A More Advanced Archetype Model

Conclusions

Although there are numerous avenues for further research, it should be clear by now that the approach described here offers significant advantages over the classical mode of development.

Domain Empowerment

The separation of domain and technical concerns into two streams of development actually clarifies the purpose of each stream, and allows software and domain experts to work in a way which suits them best, and maximises their influence over the information systems they build and use. Two important practical considerations mandate such a separation:

- The scale of knowledge models and vocabularies in most domains may be vast. In the clinical health domain, there are $O(100,000)$ - $O(1,000,000)$ terms, and probably 200,000 - 300,000 concepts, taking into account specialist medical areas (but not nursing or allied health). At the ROM level, there are $O(100)$ concepts.
- The process of defining domain models is never likely to “finish”, whereas for technical models, it must finish at some point, in order to deploy systems.

The separation of concerns can ensure that domain experts and users have much better input and control over the information systems they ultimately use. They are able to change the behaviour of systems over time by introducing and evolving vocabularies and archetypes, unlike the usual situation where they are beholden to IT departments or vendors.

Future-proof Systems *and* Information

As a consequence of building software based on a carefully designed reference object model suited to the domain, systems have much greater longevity, due to the low rate of change in such models. Use of a reference object model *standardised for the domain* is likely to further increase the longevity.

Even more importantly, the *information created by such systems will be (nearly) future-proof*, since it is constructed from the standard elements of the ROM only, while expressing a myriad of domain concepts.

In some domains, *longevity of information is extremely important*. Public record offices need to retain records for decades; financial systems often deal with long-term securities; government statisticians and meteorology bureaux use data going back as far as possible. In clinical medicine, birth-to-death electronic health records need to be active for something like a century.

Systems which can support such long-lived information at minimal ongoing cost stand to add significant value to the information economy.

Interoperable Systems

Interoperability is probably the most important attribute of information systems today, due to the growing need to share complex information widely. A global technical infrastructure is available to do this, in the form of the internet.

However, even today, interoperability is a haphazard affair. Where it exists, users are often locked into certain vendors, and have limited flexibility. Outside of the finance arena, where OMG standards are fairly heavily used, interoperability technologies have had a surprisingly low uptake. Further, even when solutions for sharing information are put in place, systems can only understand each other at the concrete model level, i.e. at the level of basic data types and structures encoded into the software.

Contrast this with the archetype approach, under which systems interoperate technically at the level of concepts encoded in the concrete model, but are able to share knowledge rather than just data. Further, as time goes on, the introduction and refinement of archetypes over the long term allows the system to learn rather than become obsolete.

The approach can also be applied to legacy systems in such a way as to allow them to evolve toward standards-based interoperability, without requiring “big-bang” redevelopments.

Intelligent Querying

Archetypes provide a disciplined basis for querying, via the dual mechanisms *archetype query profiles* (via which the possible queries to data created according to archetypes can be known in advance), and instance data *archetype maps* (the small parallel data structures, describing the archetypes actually used during construction). As a result, even very large amounts of data are no longer opaque, and do not need to be queried by brute force methods.

Automatic processing

With standardised terms, archetypes and querying, information systems are not only far more useful to humans, they can be used more reliably by automatic processing systems, for example decision support and statistical systems. Such systems can make use of archetype query profiles to make assumptions about what data is available in a system, and these assumptions can even be built into the software.

References

General

1. Beale Thomas. Eiffel Business Systems. 1998 (available at http://www.deep-thought.com.au/it/eiffel/eif_bus_sys/main.book.pdf).
This tutorial was presented at TOOLS Eastern Europe 1999, and describes an investment system developed with an early idea of archetypes.
2. Booch, Grady. *Object-oriented Analysis and Design*. 1994, Benjamin Cummings.
3. B. Chandrasekaran¹, J. R. Josephson¹ and V. Richard Benjamins. The Ontology of Tasks and Methods. Available at <http://spuds.cpsc.ucalgary.ca/KAW/KAW98/chandra/>.
4. Fowler, Martin. *Analysis Patterns: Reusable Object Models*. 1997, Addison Wesley Longman.
This is one of the few software books dealing with meta-models. Fowler gives some good examples in health and finance, and particularly useful ideas in demographic modelling. He does not provide a general approach such as archetypes, however.
5. Fowler, Martin. *UML Distilled*. 2nd Ed. 2000 Addison Wesley Longman.
6. Genesereth, M. R., & Fikes, R. E. (1992). *Knowledge Interchange Format, Version 3.0 Reference Manual*. Technical Report Logic-92-1, Computer Science Department, Stanford University.
7. Gruber, Thomas R. *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*. 1993, Stanford Knowledge Systems Laboratory.
8. Meyer, Bertrand. *Object-oriented Software Construction*. 2nd Ed. 1997, Prentice Hall.
9. Leonid Mikhajlov and Emil Sekerinski. The Fragile Base Class Problem and Its Solution. Available at <http://www.tucs.abo.fi/publications/techreports/TR117.html>. TUCS Technical Report No.117, May 1997
10. Scott A. Renner, Arnon S. Rosenthal, and James G. Scarano. *Data Interoperability: Standardization or Mediation*. 1996, IEEE. Available at <http://computer.muni.cz/conferen/meta96/renner/data-interop.html>.
11. Rumbaugh, James. *Object-oriented Modelling and Design*. 1991, Prentice Hall.
12. Walden, Kim, and Nerson, Jean-Marc. *Seamless Object-oriented Software Architecture*. 1995, Prentice Hall.

Health

13. Beale, Thomas and Heard, Sam. *The GEHR Object Model Architecture*. 1999, the GEHR project (available at http://www.gehr.org/technical/model_architecture/gehr_architecture.html).
14. Beale, Thomas. *The GEHR Kernel Architecture*. 1999, the GEHR project (available at http://www.gehr.org/technical/kernel_architecture/kernel_architecture.html).
15. Beale, Thomas. *The GEHR Archetype System*. 1999, the GEHR project (available at http://www.gehr.org/technical/archetypes/gehr_archetypes.html).

16. CEN TC 251 EHCRA standards. See <http://www.centc251.org/>.
17. CORBAmed. See <http://www.omg.org/corbamed>.
18. GEHR (Good Electronic Health Record). See <http://www.gehr.org>.
19. GEHR (Good European Health Record). See <http://www.chime.ucl.ac.uk/HealthI/GEHR/Deliverables.htm>.
20. HL7 version 3. See <http://www.hl7.org>
21. ICD (International Classification of Diseases). See <http://www.who.int/whosis/icd10/>.
22. Open Infrastructure for Outcomes. See <http://www.TxOutcome.Org/>.
An open-source project using knowledge models called “forms”, but does not impose a strict Reference Object Model (i.e. only basic types and ad-hoc structures are used in forms).
23. SNOMED (Systematized Nomenclature for Medicine). See <http://www.snomed.org/>.
24. SynEx project, UCL. See <http://www.chime.ucl.ac.uk/HealthI/SynEx/>.
This EC-funded project looks for solutions for federating legacy systems, and uses the CEN ENV 13606 model as a Reference Object Model, and a “dictionary” of configuration concepts very similar in purpose and form to the GEHR archetypes.
25. D. Kalra et al. *Software Components developed by UCL for the SynEx Project & London Demonstrator Site*. UCL, 2001.
26. UMLS (Unified Medical Language System). See <http://www.nlm.nih.gov/research/umls/>.

Index

A

archetype 8, 28
archetype namespace 49
Archetype Rules 45

B

Backward Compatibility 55
Basic Types 35
black box effect 7

C

clinical protocol 27
common terminology 17
Constraint Models 28
constraint-transform 39
Container Types 36

D

decision support systems 9
definition of “concept” 24
domain concepts 8
Domain specialists 13
dual-model methodology 17
Dynamic Variable Types 36

F

Forward Compatibility 55

G

GEHR 9

I

identification scheme 49
international sharing of information 31
Interoperability 13

K

knowledge representation design 16
Knowledge System Ontology 17

O

object-oriented models 16
object-oriented systems 11

P

Problem of Variability 26

R

regular expression 27

S

Single-model Methodology 11
Software development 17
Standardised vocabularies 8
Strong typing 15

T

Templates 24

U

UML 13

END OF DOCUMENT