Abstract—In this paper we argue that we can effectively apply the paradigm of model-driven development to the definition and implementation of healthcare IT standards, leading to an approach that can be characterized as ‘standard-driven development’. Starting point is the observation that an IT standard itself is data that can be defined in terms of models with an unambiguous meaning. For existing standards these models may have to be extracted from the definitions of the standards. Using these models and interpretive and code generation techniques the standards can be systematically turned into operational objects, that can be effectively used in developing standards-based applications. We first discuss model-driven development and then define more precisely what we mean by standard-driven development. We illustrate the approach by our experiences with the application of standard-driven development to two standards, a simple and a complex standard: UCUM and HL7 version 3.

Keywords: standards; healthcare IT; interoperability; model-driven development; HL7

I. INTRODUCTION

In healthcare billions of messages and documents are exchanged electronically every day. The data contained in these messages and documents relates to any aspect of healthcare and includes diagnostic data, images, medication prescriptions, lab results, administrative data, etc. It is no wonder that information and communication standards play a key role in healthcare, more than in almost any other domain. In this paper we will refer to standards for the digital exchange of messages and documents as IT standards. The number of IT standards in healthcare, international as well as national, is overwhelming. New standards are being developed and introduced on a regular basis. For a survey of healthcare standards and standards developing organizations (SDOs) see e.g. [1].

A key requirement for IT standards in healthcare is semantic interoperability [2], which can be characterized in several ways. Put simply, it amounts to the fact that the meaning of data conforming to the standard can be comprehended unambiguously by both humans and computer programs. In the end, the actions that are inferred from the data are that count, in particular the actions that affect the patient. Any difference in interpretation between the originator of the data (such as a physician diagnosing a patient) and the person acting on the data (such as a surgeon operating the patient) can have serious consequences. In healthcare, semantic interoperability can therefore literally be a matter of life and death.

In the rest of this paper we will use the term ‘semantic interoperability’ in the narrower sense of computable semantic interoperability [3], implying that we focus on the exchange and interpretation of data by computers. Put more precisely, computable semantic interoperability refers to the property of a standard that data carrying semantic content can be interpreted unambiguously by computer systems without any implicit knowledge of the context of the data, i.e. by using information from the standard only.

Even if a standard satisfies the requirement of semantic interoperability, differences in interpretation of data may still occur due to implementations that interpret the standard differently. An example is ‘context conduction’ in HL7v3, which is so complicated that several implementations do not even support it. So, besides the meaning of the data defined by the standard also the meaning of the standard itself should be unambiguous. In most IT standards an effort is made to define a standard as precisely as possible. In practice it turns out that several of these standards contain grey areas that may give rise to different interpretations of parts of the standards. This is usually due to the use of natural language in defining (parts of) the standard.

In this paper we plea for a different way of dealing with IT standards in healthcare which aims at achieving semantic interoperability at the standard level (besides the data level). Starting point is the observation that an IT standard itself is data that should have an unambiguous meaning. It should be possible to process and interpret the standard at the semantic level in the same way as the data defined by the standard, i.e. by means of computers. We do not need another standard to do this, i.e. a kind of USL (Unified Standardization Language). We will argue that we can apply the paradigm of model-driven development (MDD) to the definition and implementation of standards, leading to an approach that can be characterized as standard-driven development (SDD).

Essential to SDD is that standards are defined in such a way that they can be used ‘out of the box’ for developing applications that use the standard. It should be possible to cover all aspects of the standard, including the semantic aspects of the artefacts defined by the standard, by developing code generators and/or interpreters that operate on the standard as a
data object, i.e. as a model. The required interpretation of the standard should be limited to the interpretation of this model. There should be no need to reconstruct the meaning of the artefacts defined by the standard from informal or implicit descriptions. With standards defined as ‘operational objects’ in this way, implementation efforts of standards can be drastically reduced, which makes it easier to cope with the increasing number of standards and versions of standards that healthcare information systems have to support.

Trends towards a model-driven way of defining healthcare IT standards can be observed in standards such as CEN/ISO EN13606 [4] and HL7 version 3 [5]. The idea of SDD is therefore not new. By identifying it, making it explicit and describing it as a systematic approach that can be used in the definition and use of healthcare IT standards we hope that it will contribute to improving the quality and ease of use of standards.

Most existing standards require re-engineering to make them suitable for standard-driven development. Besides discussing standard-driven development in general, we will also discuss our experiences with the application of SDD to two standards, a simple and a complex standard: UCUM [6] and HL7v3 [5]. Short descriptions of these two standards can be found in the following two subsections. We will use these standards as running examples in the discussion of standard-driven development. Before doing that, we first discuss model-driven development.

A. UCUM

UCUM (Unified Code for Units of Measure) is a code system intended to include all units of measure being contemporarily used in international science, engineering, and business [6]. The purpose is to facilitate unambiguous electronic communication of quantities together with their units. As such it is used in several healthcare standards such as HL7 and DICOM. UCUM defines a small collection of ‘base units’ (for length, time, mass, etc.) and provides rules for composing units using algebraic operators. The algebraic terms are referred to as unit terms. In addition, UCUM defines a large collection of ‘derived units’ that are defined in terms of other units by means of unit terms. An example is the derived unit “N” (Newton) which is defined as “1 kg.m/s²”. The semantics of units is defined relative to a system of base units (such as the base units defined in the standard itself) by a numeric factor and a vector of exponents by which the base units contribute to the unit.

B. HL7 Version 3

HL7v3 [5] is a standard for the exchange of healthcare related data, supporting a message-based as well as document-based paradigm of information exchange. It includes a model-based methodology for the development of artefacts, the Hierarchical Development Framework (HDF). The cornerstone of this methodology is the Reference Information Model (RIM) from which all other message and document models are derived. Besides the RIM-derived models, the standard defines various other artefacts such as data types, vocabularies and storyboards. A good introduction to and survey of HL7v3 can e.g. be found in Chapters 7–9 of [7].

HL7v3 is an example of an extensible standard. The standard itself defines how the standard can be extended with new models that define new artefact types, such as new message types. After going through a balloting process, newly defined models can become a normative part of the standard. The formal language used to define extensions of the standard is called MIF (Model Interchange Format). This language was introduced in a later stage of the development of the HL7v3 standard and does not cover all formal aspects of HL7v3 yet (such as ‘templates’).

II. MODEL-DRIVEN DEVELOPMENT

Standard-driven development is like model-driven development, where standards are used in the same way as models in model-driven development. In model-driven development domain models (also referred to as metamodels) are used to define types that characterize the objects in a specific domain at a high level of abstraction. Typically, the types define attributes and associations that define the structure and relations of the various kinds of objects in the domain. In addition, constraints and operations may be used to define semantic aspects of the objects. Domain models as well as the objects defined by their types are data, implying that they can be processed by computers and can be used to drive the development of applications in the domain by means of code generation and/or interpretation. Model-driven development is therefore data-driven development: rather than writing code, we construct data objects from which code is generated. More information on model-driven development can be found in [8][9].

Starting software development with the development of models rather than code helps in raising the abstraction level in the development process, which will in the end pay off in raising productivity with potentially several factors. In models we focus on the definitions of types that characterize the objects in a specific domain and abstract from implementation details. These type definitions are data objects that can be visualized, edited, analyzed, etc. in ways that appeal to domain experts. In traditional software development type definitions are buried in code. They can also be inspected as data objects by a mechanism called reflection, but these data objects are inherently blurred by implementation detail. MDD works the other way around: instead of looking backwards from the code (reflection) to the definitions of the types, it takes implementation-independent type definitions (the model) as the initial point of view and considers the implementation code as a derived view, thus establishing a clean separation of concerns.

Unlike e.g. object-oriented development, model-driven development is not yet main-stream although more and more model-driven development tools are entering the market. An example is Microsoft’s Entity Framework 4, which offers a ‘model first’ approach with respect to database development (besides the classical way of database development). In this approach a database and high-level code to access the database is generated from an abstract entity data model defined by the user. One of the problems with model-driven development is that it requires a paradigm shift from the application developer. It moves away from the classical way of software development to developing applications by constructing and transforming...
(domain) models and writing code generators or interpreters. The indirection introduced by the use of models may not always seem to be worth the investment, especially because the development of good domain models is hard and these models may be short-lived.

In the discussion of standard-driven development we will not make any assumptions on the model-driven technology that is used. To a certain extent it does not matter which technology is used, at least not in the discussion of the standard-driven development approach. Prominent examples of MDD technologies are OMG’s Model-Driven Architecture (MDA) [10], the open source Eclipse Modeling Framework (EMF) [11] and Microsoft’s Domain-Specific Language Tools (DSL) [12]. In the implementation of the examples of SDD that we will discuss, we used the VAMPIRE approach developed in-house in Philips [13], but the discussion is not dependent on this.

III. STANDARD-DRIVEN DEVELOPMENT

In standard-driven development the idea is to treat a standard as a domain model, where the types defined in the domain model represent the artefacts defined by the standard. We will refer to this domain model also as the artefact model. Examples of healthcare IT standards and the artefacts defined by them are:

- LOINC: artefacts are codes for clinical observations.
- UCUM: artefacts are units of measurements.
- DICOM: artefacts are information objects, attributes, commands, etc.

Healthcare IT standards are not usually defined as domain models in the sense of MDD, so it may require considerable effort to re-engineer an existing standard as a domain model. There is one advantage over general model-driven development: standards are long-lived models so the return on investment will generally be higher. Furthermore, standards, and especially extensible standards (such as HL7v3), usually incorporate a lot of domain knowledge. By casting that knowledge in terms of domain models and creating code generators and/or interpreters, we can operationalize that knowledge and thus profit from the work of others.

A. Operationalizing a Standard

The process of operationalizing a standard is schematically indicated in Figure 1. The first step is to take the definition of the standard, which is often written in natural language or in some kind of ‘standard legalese’, and develop a model extractor that reconstructs the artefact model underlying the standard. If the underlying model of the standard is not obvious, this may require a creative act. Ideally, the reconstruction is performed completely programmatically but in practice manual steps are often necessary. In some cases there may be obvious errors such as typos in the sources of the standard. Rather than trying to intercept these errors in the model extractor code, it is better to make the corrections in the standard definition itself (and pass the errors to the authors of the standard).

Model extraction can be hard if the standard is only available in a format in which much of the underlying document structure is lost (such as in PDF). Microsoft Word documents are already better, because the Word object model can be used to analyze the documents. Standards that are defined, in whole or in part, in terms of XML or other structured languages are even better.

In the case of UCUM, the definition of the standard consists of an informal HTML document. Strictly speaking, the UCUM artefacts are strings because units and unit terms are defined by their textual representation. From the SDD point of view, the UCUM artefacts are the types identifying the various kinds of entities that play a role in the composition of a unit term and the textual representation of a unit term is viewed as the serialization of that unit term.

The UCUM standard provides a BNF syntax of unit terms from which part of the artefact model can be more or less directly derived (after abstracting from some of the syntactic details, such as the use of parentheses). The other part of the artefact model is concerned with ‘unit atoms’ and ‘prefixes’ and can be constructed from the informal sections that define the various kinds of these entities. The UML representation of the artefact model that we derived is shown in Figure 2. It
shows the normative as well as the non-normative information (such as full names and print symbols).

The second step in operationalizing a standard is to turn the artefact model into an operational object that can be used in applications that require the standard. A typical way to do this is to use a code generator (as indicated in Figure 1) and map the artefact types in the artefact model to types in an object-oriented programming language such as Java or C# and provide an API to access these types. The advantage is that applications can use strongly-typed code to create and manipulate artefacts that conform to the standard. In the case of UCUM, the API offers types such as Term, SimpleTerm, UnitAtom, and PrefixedUnitAtom that have properties corresponding to the attributes defined in the artefact model.

Besides providing the functionality by means of generated code that is compiled, it is also possible to use an interpreter that interprets the artefact model as data (see Figure 3). In pictures we will show the generative approach only but one should realize that an interpretive approach can always be used instead.

![Diagram of Compilation versus Interpretation](image.png)

The advantage of the interpretive approach is that a generic API can be provided that supports uniform access to all artefact types. The disadvantage is that interpretive APIs are less strongly typed and are thereby harder to use and more error-prone in application development. Ideal is the combination of the two: an API that provides generic as well as strongly typed access. For example, because an HL7v3 CDA document is an instance of the Act type of the RIM, we can provide a generic RIM interface as part of the document API which can be useful in performing operations that are not document-specific (such as traversing all sections of the document). However, in order to process a specific section or entry in a CDA document, it can be useful to also have a strongly-typed interface. For example, in a Continuity of Care Document (which is a CDA document), such an interface would typically provide a ‘FamilyHistory’ property to directly access the family history section of the document. Using the generic RIM interface we would have to search for the section with a templateId equal to “2.16.840.1.113883.10.20.1.4” (see [14]).

The functionality that is provided by the API depends on the kind of standard but typically includes aspects such as:

- Saving and loading of artefact instances to and from a persistent medium. For UCUM this is not very useful, but for HL7v3 it certainly is.
- Semantic aspects. For UCUM these consist of: validation of units, reduction of units to their canonical form, commensurability and equivalence of units, and conversion from one unit to another.

With a properly defined standard, large parts of the code supporting these aspects can be generated directly from the standard definition. This certainly applies to the first three aspects referred to above, which can be covered either by general-purpose code generators (as provided by the modeling framework used) or by custom code generators developed for the specific standard.

For the semantic aspects, not all code can usually be generated and custom code may have to be written and added to the code generated from the model. Conceptually, that ‘fixed code’ can also be viewed as generated code. In line with the data-driven development paradigm, the amount of custom code can be reduced to a minimum by encoding semantic (and other) information as much as possible as data that is contained in the model itself. The interpretation of that data should be simple, while the encoded information itself may be complicated. The custom code can then be implemented as a simple interpreter that is ‘driven’ by the encoded data. Constraints can e.g. be expressed as data in the model, either using a general-purpose constraint language (such as OCL [16]) or by properties expressing constraints specific to the model. The ‘IsMetric’ property of a ‘UnitAtom’ is an example of such a model-specific constraint.

Although representing semantic information as data instead of code is generally a good idea, there is a trade-off. Some knowledge, such as procedural knowledge, is inherently algorithmic and may be better defined in an algorithmic language. Other knowledge is of an inherently mathematical nature and may be better defined using standard mathematics, as is done in the UCUM standard with the algebraic properties of units. The essential thing is that the meaning of the data is clear and unambiguous. In the operationalized version of the standard that meaning can then be mapped to code in a verifiable way. If there is a lot of ‘random detail’ in the meaning of artefact types, such as many exceptions to general rules, it makes sense to encode that as data in the model, which helps in keeping the code as clean as possible.

A basic semantic aspect that is extremely important is conformance because it is the precondition for semantic interoperability. An artefact that does not conform to the standard has no meaning. Because communication is a two-way process, there are two sides to the conformance medal.

First of all, in the case of artefacts that come from an external source, we can only check conformance after the fact, i.e. by validating the artefact. As already stated, by including the conditions to be validated as data in the artefact model, a validator can be implemented as a relatively simple interpreter of that data. Secondly, in the case of outgoing artefacts that we construct ourselves, the best way to guarantee conformance is to build conformance into the construction process itself, i.e. to
implement conformance by construction. This can be achieved by generating code from the artefact model that prevents the construction of non-conforming components of the artefacts, either by automatically filling in dependent parts of the components (such as specific IDs or codes required by the standard) or by generating exceptions when preconditions for the construction of the components are violated.

Besides artefact types, many standards also define specific artefact instances that are considered part of the standard. For example, the UCUM definition contains tables containing predefined units, i.e., instances of the artefact type “UnitAtom”. The final step in operationalizing a standard is therefore to process the instances defined in the standard and provide an interface to access them. With support for creating instances of the artefact types in place, this is a simple job. The information defining the instances has to be extracted from the standard and used for creating and initializing the objects that represent them. The semantic support can be used to validate the instances. This is schematically indicated in Figure 4.

![Figure 4. Operationalizing a Standard (Refined View)](image)

In the case of UCUM, the implementation of the upper level of Figure 4 was relatively simple. Besides the HTML definition, the standard contains an XML file with the definitions of all units and prefixes (all ‘instances’) defined by the standard. The XML file can be viewed as an extraction of the HTML definition, which also contains the definitions of the units and prefixes. This saved us the implementation of the ‘instance extractor’. The only problem was that there turned out to be minor differences between the HTML and XML definitions, which raises the question which of the two is normative. Problems such as these can be avoided if the complete standard including the informal parts is represented in XML and the HTML version is generated from it. We also included the example units defined in the standard as instances. These are only defined in HTML part of the standard and therefore required a separate instance extractor.

B. Operationalizing an Extensible Standard

So far we have implicitly assumed that a standard defines a fixed number of artefact types, which is indeed the case for many relatively simple standards such as LOINC or UCUM. There are also standards that do not define a fixed set of artefact types but provide a mechanism for extending the standard with new artefact types. They contain what may be called a metastandard: a standardized mechanism for defining new artefact types. A metastandard is truly a ‘standard in a standard’ and can be seen as a domain-specific formalism for defining artefact types.

Examples of extensible standards are HL7v3 and EN13606. The metastandard in HL7v3 is called MIF (Model Interchange Format). It supports the definition of new artefact types such as static models, value sets, trigger events and story boards. The metastandard in EN13606 is called ADL (Archetype Definition Language). It supports the definition of new ‘archetypes’, which are models of clinical information concepts that appear in EHRs.

The metastandard can be operationalized in exactly the same way as a non-extensible standard. Using a ‘metamodel extractor’ we can extract an artefact metamodel from the definition of the metastandard, if the metastandard is not already defined as an explicit metamodel. From the metamodel we can generate code that supports the construction and processing of artefact types. This code can be used to extend the standard with new artefact types, which is particularly useful for developers of standards (like SDOs), as opposed to the code for processing instances of artefact types, which is particularly useful for developers that use the standard. The architecture for operationalizing an extensible standard thus leads to the three-level approach indicated in Figure 5.

![Figure 5. Operationalizing an Extensible Standard](image)

A point of debate is whether each extensible standard needs its own artefact metamodel. The instances of the ‘metatypes’ defined by the metamodel are used as building blocks for artefact type definitions. One can argue that a general-purpose metalanguage such as UML is sufficient as a metastandard. UML provides mechanisms such as profiling and stereotypes that can turn UML into a domain-specific metalanguage. Another option is using the Meta Object Facility (MOF) [15] that is used for the definition of UML itself. There can nevertheless be good reasons not to use UML and to choose a dedicated artefact metamodel specific to the domain covered by the standard. The first is that UML is a big and complex language, while we probably only need a small subset. The
second is that there may be aspects of artefacts that have no correspondence in UML (such as vocabulary domains in HL7v3).

Below we will illustrate the process of operationalizing an extensible standard by discussing how we did this for (a part of) the HL7v3 standard. The goal at the time was to use standard-driven development to create a simple HL7v3-based repository for storing clinical statements (in XML format). In the discussion we will cover all three levels of Figure 5. The implementation of the upper two levels is similar to what we discussed in the previous section, but the extensibility and complexity of the standard creates additional issues. We did not do that for all HL7v3 artefact definitions, thus providing support for processing instances of the artefact types. The latter would e.g. have required the generation of full MIF validation support including the processing of the Schematron files that define the constraints on MIF definitions.

The first step was to take the metastandard (MIF) and extract a metamodel from it. MIF itself is an XML language that is defined by means of a set of inter-related XML schemas. We used our own XML-schema-to-model transformer to map the (flattened) schema definition to an object model, the ‘MIF object model’. The MIF object model corresponds to the ‘artefact metamodel’ in Figure 5. From this object model (containing serialization metadata) code was generated using the model-to-class generator that is part of the modeling framework. This provided a way to read MIF files as instances of the object model; see Figure 6. This was sufficient for our purpose as we did not need full support for authoring MIF files. The latter would e.g. have required the generation of full MIF validation support including the processing of the Schematron files that define the constraints on MIF definitions.

![Figure 6](image)

**Figure 6. Level 1: Operationalizing the Metamodel**

By being able to read MIF files as instances of the MIF object model, the abstraction level is raised from the XML schema domain to the ‘model domain’, although the types in the object model are still a clear reflection of the schema types. By ‘model domain’ we mean the modeling framework that we are using. The important thing is that once we are in the model domain, a substantial part of the software development process can be carried out by means of model-to-model transformations within the same modeling framework. Such transformations are a lot simpler than text-based or XML-based transformations such as XSLT transformations. The main reason is that in such model-to-model transformations we can focus on the pure structure and semantics of the models and do not have to deal with representational detail such as the distinction between attributes and elements in XML. Note that ‘model’ in ‘model-to-model’ can be ‘instance’, ‘model’, ‘metamodel’, etc., in any combination.

The second step in the process of operationalizing the HL7v3 standard (level 2 in Figure 5) was to take the MIF artefact definitions that are part of the standard, transform them into models in the modeling framework and generate code from these models, thus providing support for processing instances of the artefact types. We did not do that for all HL7v3 artefact definitions that have a MIF definition but restricted ourselves to the artefact definitions that we needed in our application:

1. Data types.
2. Static models, such as the RIM (Reference Information Model), the CDA (Clinical Document Architecture) and some CMEs (Common Message Element types).
3. Vocabulary.

For each of the above three kinds artefact definitions we constructed a separate model extractor (see Figure 5). Each model extractor consists of a front-end that reads a MIF file as an instance of a type from the MIF object model and then transforms it to a model. The first step uses the code generated from the MIF object model and the second step is a simple model-to-model transformation.

The model extractor for the data types (‘DT to object model’ in Figure 7) reads the MIF definition of the data types as an instance of the type ‘DatatypeModelLibrary’ from the MIF object model. That instance represents a model and is mapped by the model extractor to a true object model in the modeling framework. The types in this model are the HL7v3 data types. Unfortunately, the axiomatic definitions of the data types in this model are too abstract to allow the automatic generation of an implementation from this model. Hence, instead of the MIF definition of the data types, we used the more concrete XML schema definition of the data types from the XML ITS to generate (and partially hand-code) the implementation of the data types. This is a problem encountered in other HL7v3 implementations that use MIF code generation as well [17].

![Figure 7](image)

**Figure 7. Level 2a: Operationalizing the Data Types**

The model extractor for the static models (‘SM to object model’ in Figure 8) reads the MIF definition of a static model as an instance of the type ‘StaticModel’ from the MIF object model. Like the MIF definition of the data types, such an instance represents a model and is mapped by the model extractor to an object model in the modeling framework. Because each type in this model derives from one of the main RIM classes such as Act, Role and Entity, which play a special role in HL7v3, we hand-crafted a small metamodel for constructing these types. This metamodel, the MetaRIM model, defines a metatype (MetaAct, MetaRole, MetaEntity, etc.) for each of the main RIM classes and associates metadata with the metatypes, such as the usual color codings (red for acts, yellow for roles, etc.). The model extractor creates types by creating instances of these metatypes. So, a type in a static model that derives directly or indirectly from the RIM Act class is created as an instance of the metatype MetaAct from the MetaRIM object model. From the object model created by the model extractor, code can be directly generated using the model-to-class generator that is part of the modeling framework. The serialization/deserialization code to be generated depends on the ITS used for a static model (e.g. XML ITS or RIM ITS) and
may require additions to the code generator. In our case, the default serialization/deserialization code (based on a proprietary XML format) was sufficient because we needed it to store data locally only.

Figure 8. Level 2b: Operationalizing the Static Models

The model extractor for the vocabulary (‘VOC to instance’ in Figure 9) reads the MIF definition of the vocabulary as an instance of the type ‘VocabularyModel’ from the MIF object model. This MIF definition is different from the MIF data type and static model definitions in that it cannot be interpreted directly in terms of a model. Instead of data types and classes it defines specific entities such as concept domains, code systems and value sets. Hence in this case it made no sense to try and map the vocabulary to a model first. Instead, the vocabulary was simply created as an instance of the type ‘VocabularyModel’ and a dedicated code generator (‘VOC code generator’ in Figure 9) was used to turn the vocabulary into an operational object. In the latter step, strongly-typed code was generated with separate classes for the concepts domains, code systems, value sets, etc. and inheritance relations that reflect the hierarchical relations between the concepts. The advantage is that it simplifies the use of the vocabulary in a development environment (e.g. through the use of Intellisense in Microsoft Visual Studio), but the disadvantage is that the generated code is quite big. An alternative is to simply map the vocabulary to a set of tables and provide a dedicated user-friendly tool to access the tables.

Figure 9. Level 2c: Operationalizing the Vocabulary

Figure 7-Figure 9 together constitute level 2 in Figure 5. This shows that reality can be a bit more complex than Figure 5 suggests, but the general structure of Figure 5 still applies. The implementation of level 3 in Figure 5 is simple, given the support for creating instances of the HL7v3 data types, instances of the types from static models, etc.

Note that we restricted ourselves to the MIF-defined artefacts. In order to create full standard-driven support for e.g. CDA templates, we would need a way to extract the constraints defined by these templates in a format that can be processed semantically by a computer. Since the templates are defined in implementation guides that are largely informal documents (see e.g. [14] for the ‘Continuity of Care Document’), this cannot be done fully automatically. Some parts of these documents, such as tables and conformance statements in structured English, can be processed automatically but other parts have to be processed manually.

IV. CONCLUSION

In the previous sections we have discussed standard-driven development as a way of turning healthcare IT standards into operational objects that can be effectively used as re-usable components in application development. What we refer to as ‘operational objects’ can also be read as ‘implementations’. Using the latter term would suggest that standard-driven development is merely a way of creating a code base that implements a standard, which would give a false impression. The essential idea is that the definition of a standard is represented at all of its metalevels in terms of models. These models should include as much semantic information contained in the standard as possible, such as conformance constraints. This way the code (code generators and/or interpreters) necessary to make the standard executable can be kept relatively simple. Because models are inherently platform-independent (since they are data), this also means that it is relatively simple to support these operational objects on different platforms.

Unfortunately, most existing IT standards are not or only partially defined in terms of models. In our experience with UCUM and HL7v3, as well as DICOM Structured Reporting (to which we are currently applying the same approach), most of the effort in operationalizing these standards goes into the extraction of the models from the documents that define these standards. This even applies to those standards that are already partially model-based (such as HL7v3). Once the definition of a standard is cast uniformly into models, the effort to create the executable code is relatively small. Although the initial effort of extracting the models may be substantial, we still think that it makes sense to apply standard-driven development to existing IT standards. The reason is not only that most standards are long-lived, but also that it makes it simpler to support new versions of standards. When a new version of a standard is released (such as a new normative edition of HL7v3, that is released every year), much of the model extraction code as well as the models can be re-used. Slight modifications may be necessary, but the overall effort to support a new version of the standard can be limited.

Standard-driven development can drastically reduce implementation efforts if we can eliminate the model-extraction process altogether. This requires that standards be defined in terms of formal models from the beginning, including the semantic aspects of the standards. It then also makes sense to create a reference implementation as part of the standard and to publish the standard as a truly operational object. By including informal text in the models the standard documentation can be generated from the models, which allows Figure 5 to be simplified to Figure 10.
We plea therefore for a different way of defining IT standards in healthcare, where models take the front seat. We think this way of defining IT standards is feasible if domain experts and modeling experts closely cooperate in standards committees. By defining standards as operational objects from the beginning, the adoption and implementation of standards, including the guaranteeing of semantic interoperability, will be made a lot easier. There remain of course issues to be resolved. For example, what platform should the code provided with the standard (the right part of Figure 10) be based on? As already discussed, this need not be a big issue because all ingredients for the code are available as data in the models. This makes it simple to either develop one’s own model-driven implementation or to provide reference implementations for multiple platforms as part of the standard definition.

The use of models in the definition and implementation of healthcare IT standards is certainly not new. Standards such as EN13606 [4] and HL7v3 [5] include formally defined models, although these models do not cover all (semantic) aspects of the standards. A similar approach as we used in ‘operationalizing’ HL7v3 is used in the development of HL7v3 support in the Open Health Tools project [18] where Eclipse and EMF are used as development tools (as opposed to Visual Studio and VAMPIRE that we used). A difference is that we are using the DSM approach to modeling [8] rather than the MDA approach [10] (which includes the use of UML and OCL). However, in operationalizing an existing standard such as HL7v3 it is to a certain extent irrelevant what modeling framework we are using because we create and use the models ourselves.

When we are defining a standard using the scenario indicated in Figure 10 the choice of a modeling framework may be an issue, because the models act as the public source of the standard. Preferably a single modeling framework should be chosen that is flexible enough to cover all metalevels of the standard. This could be EMF [11] but also a language-neutral framework such as the Intentional Domain Workbench [19] which is extremely flexible in supporting multiple metalevels.

An issue in itself is the role of XML which is rather dominant in several healthcare standards. XML Schema is frequently used as a modeling language (with Schematron playing the role of ‘constraint language’). This will probably be further stimulated by the advent of the new XML Schema 1.1 language [20]. XML is perfect as a serialization format, but in our opinion its role should be restricted to that. Domain-specific models are perfect for defining and transforming artefacts at a high level of abstraction, in contrast with XML representations. This provides a clean separation of concerns. The link between the two can be established by adding serialization metadata to domain-specific models and automatically generating the serialization/deserialization support (including an XML schema) from the models defining the artefacts.

REFERENCES

[9] Special Issue on Domain-Specific Modeling, IEEE Software (July/August 2009).