Reifying Design Patterns to Facilitate Systems Evolution

Florida Estrella[1,2], Zsolt Kovacs[2], Jean-Marie Le Goff[2],
Richard McClatchey[1], and Norbert Toth[1, 2]

[1]Centre for Complex Cooperative Systems, UWE
Frenchay, Bristol BS16 1QY UK
Tel: +44 1179 656261, FAX: +41 22 767 8930
{Florida.Estrella,Richard.McClatchey,Norbert.Toth}@cern.ch
[2]ETT Division, CERN, Geneva, Switzerland
Tel +41 22 767 6559, FAX: +41 22 767 8930
{Zsolt.Kovacs,Jean-Marie.Le.Goff}@cern.ch

Abstract. In the Web age systems must be increasingly flexible, reconfigurable and adaptable in addition to being developed rapidly. As a consequence, designing systems to cater for change is becoming critical to their success. Allowing systems to be self-describing or *description-driven* is one way to enable this. To address the issue of evolvability in information systems, this paper proposes a pattern-based description-driven architecture. The proposed architecture embodies four pillars - firstly, the adoption of a multi-layered and reflective meta-level architecture, secondly, the identification of four modeling relationships that must be made explicit to be examined and modified dynamically, thirdly the identification of five patterns which have emerged from practice and have proved essential in providing reusable building blocks, and finally the encoding of the structural properties of these design patterns by means of one pattern, the Graph pattern. A practical example of this is cited to demonstrate the use of description-driven data objects in handling system evolution.

1 Background

A crucial factor in the creation of flexible web-based information systems dealing with changing requirements is the suitability of the underlying technology in allowing the evolution of the system. Exposing the internal system architecture opens up the architecture, consequently allowing application programs to inspect and alter implicit system aspects. These implicit system elements can serve as the basis for changes and extensions to the system. Making these internal structures explicit allows them to be subject to scrutiny and interrogation.

A reflective system utilizes an open architecture where implicit system aspects are reified to become explicit first-class *meta-objects* [1]. The advantage of reifying system descriptions as objects is that operations can be carried out on them, like composing and editing, storing and retrieving, organizing and reading. Since these meta-objects can represent system descriptions, their manipulation can result in change in system behaviour. As such, reified system descriptions are mechanisms that can lead to dynamically modifiable and evolvable systems. Meta-objects, as used here, are the self-representations of the system describing how its internal elements can be accessed and manipulated. These self-representations are causally connected to the internal structures they represent; changes to these self-representations immediately affect the underlying system. The ability to dynamically augment, extend

A. Olivé (Eds.): ER 2003 Ws, LNCS 2784, pp. 75-87, 2003.

and re-define system specifications can result in a considerable improvement in flexibility. This leads to dynamically modifiable systems that can adapt and cope with evolving requirements, essential in a web-oriented system.

There are a number of OO design techniques that encourage the design and development of reusable objects. In particular design patterns are useful for creating reusable OO designs [2]. Design patterns for structural, behavioural and architectural modeling have been documented and have provided software engineers rules and guidelines that they can immediately (re-)use in software development. Reflective architectures that can dynamically adapt to new user requirements by storing descriptive information that can be interpreted at runtime have led to so-called Adaptive Object Models [3]. These are models that provide meta-information about domains that can be changed on the fly. Such an approach, proposed by Yoder, is very similar to the approach adopted in this paper.

A Description-Driven System (DDS), as defined by the work reported in this paper, is an example of a reflective multi-level architecture [4]. It makes use of meta-objects to store domain-specific system descriptions, which control and manage the life cycles of meta-object instances, i.e. domain objects. The separation of descriptions from their instances allows them to be specified and managed and to evolve independently and asynchronously. This separation is essential in handling the complexity issues facing many computing applications and allows the realization of interoperability, reusability and system evolution since it gives a clear boundary between the application's basic functionalities from its representations and controls. As objects, reified system descriptions of DDSs can be organized into libraries or frameworks dedicated to the modeling of languages in general, and to customizing its use for specific domains in particular.

This paper shows, for the first time, how the approach of reifying a set of design patterns can be used as the basis of a description-driven architecture and can provide the capability of system evolution. (The host project, CRISTAL, is not described in detail here. Readers should consult [4] & [5] for further detail). The next section establishes how semantic relationships in description-driven systems can be reified using a complete and sufficient set of meta-objects that cater for Aggregation, Generalization, Description, Dependency and Relationships. In section 3 of this paper the reification of the Graph Pattern is discussed and section 4 investigates the use of this pattern in a three-layer reflective architecture.

2 Reifying Semantic Relationships

In response to the demand to treat associations on an equal footing with classes, a number of published papers have suggested the promotion of the relationship construct as a first-class object (reification) [6]. A first-class object is an object that can be created at run-time, can be passed as an actual parameter to methods, can be returned as a result of a function and can be stored in a variable. Reification is used in this paper to promote associations to the same level as classes, thus giving them the same status and features as classes. Consequently, associations become fully-fledged objects in their own right with their own attributes representing their states, and their

own methods to alter their behavior. This is achieved by viewing the relationships themselves as patterns.

Different types of relationships, representing the many ways interdependent objects are related, can be reified. The proper specification of the types of relationships that exist among objects is essential in managing the relationships and the propagation of operations to the objects they associate. This greatly improves system design and implementation as the burden for handling dependency behavior emerging from relationships is localized to the relationship object. Instead of providing domain-specific solutions to handling domain-related dependencies, the relationship objects handle inter-object communication and domain consistency implicitly.

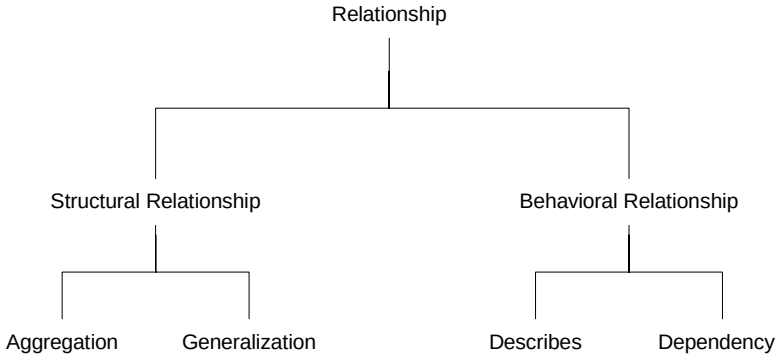

Fig. 1. Relationship classification

Reifying relationships as meta-objects is a fundamental step in the reification of design patterns. The next sections discuss four types of relationships, as shown in Figure 1. The relationship classification is divided into two types - structural relationship and behavioral relationship. A structural relationship is one that deals with the structural or static aspects of a domain. The Aggregation and the Generalization relationships are examples of this type. A behavioral relationship, as the name implies, deals with the behavioral or dynamic aspects of a domain. Two types of behavioral relationships are explored in this paper - the Describes and Dependency relationships.

It is not the object of this paper to give an exhaustive discussion of each of these relationships. Those which are covered are the links which have proved essential in developing the concepts of description-driven systems and these have emerged from a set of five design patterns: the Type Object Pattern [7], the Tree Pattern, the Graph Pattern, the Publisher-Subscriber Pattern and the Mediator Pattern [8]. Interested readers should refer to [9] for a more complete discussion about the taxonomy of semantic relationships.

2.1 The Aggregation Meta-object

Aggregation is a structural relationship between an object whole using other objects as its parts. The most common example of this type of relationship is the bill-of-materials or parts explosion tree, representing part-whole hierarchies of objects. The

familiar Tree Pattern [10] models the Aggregation relationship and the objects it relates. Aggregated objects are very common, and application developers often re-implement the tree semantics to manage part-whole hierarchies. Reifying the Tree pattern provides developers with the Tree pattern meta-object, providing applications with a reusable construct. An essential requirement in the reification of the Tree pattern is the reification of the Aggregation relationship linking the nodes of the tree. For this, aggregation semantics must first be defined.

Typically, operations applied to whole objects are by default propagated to their aggregates. This is a powerful mechanism as it allows the implicit handling of the management of interrelated objects by the objects themselves through the manner in which they are linked together. By reifying the Aggregation relationship, the three aggregation properties of transitivity, anti-symmetry and propagation of operations can be made part of the Aggregation meta-object attributes and can be enforced by the Aggregation meta-object methods. Thus, the state of the Aggregation relationship and the operations related to maintaining the links among the objects it aggregates are localized to the link itself. Operations like copy, delete and move can now be handled implicitly and generically by the domain objects irrespective of domain structure.

Figure 2 illustrates the inclusion of the reified Aggregation relationship in the Tree pattern. In the diagram, the reified Aggregation relationship is called Aggregation, and is the link between the nodes of the tree. The Aggregation meta-object manages and controls the link between the tree nodes, and enforces the propagation of operations from parent nodes to their children. Consequently, operations applied to branch nodes are by default automatically propagated to their compositions.

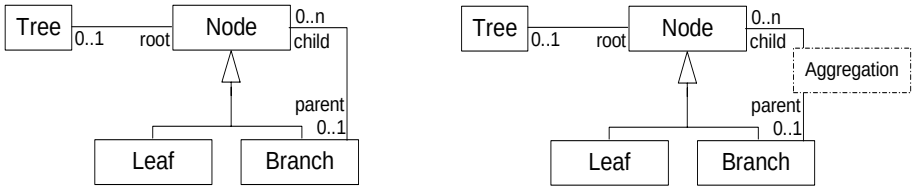

Fig. 2. The Tree Pattern with Reified Aggregation Relationship

2.2 The Generalization Meta-object

Generalization is a structural relationship between a superclass and its subclasses. The semantics of generalization revolve around inheritance, type checking and reuse, where subclasses inherit the attributes and methods defined by their superclass. The subclasses can alter the inherited features and add their own. This results in a class hierarchy organized according to similarities and differences. Unlike the Aggregation relationship, the generalization semantics are known and implemented by most programming languages, as built-in constructs integrated into the language semantics. This paper advocates extending the programming language semantics by reifying the Generalization relationship as a meta-object. Consequently, programmers can access the generalization relation as an object, giving them the capability of manipulating superclass-subclass pairs at run-time. As a result, application programs can utilize mechanisms for dynamically creating and altering the class hierarchy, which commonly require re-compilation for many languages.

As with the Aggregation relationship, generalization exhibits the transitivity property in the implicit propagation of attributes and methods from a superclass to its subclasses. The transitivity property can also be applied to the propagation of versioning between objects related by the Generalization relationship. Normally, a change in the version of the superclass automatically changes the versions of its subclasses. This behavior can be specified as the default behavior of the Generalization meta-object. Figure 3 illustrates the Tree pattern with the Generalization and Aggregation relationships between the tree nodes reified.

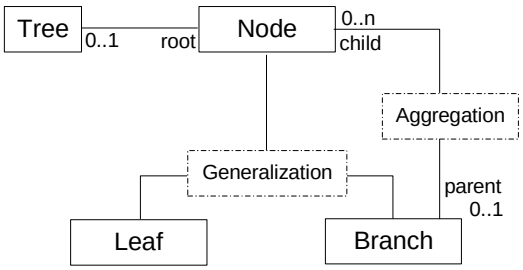

Fig. 3. Reification of the Generalization and Aggregation Relationships

2.3 The Describes Meta-object

In essence the Type Object pattern [7] has three elements, the object, its type and the Describes relationship, which relates the object to its type. The Type Object pattern illustrates the link between meta-data and data and the Describes relationship that relates the two. Consequently, this pattern links levels of multi-level systems. The upper meta-level meta-objects manage the next lower layer's objects. The meta-data that these meta-objects hold describe the data the lower level objects contain. Consequently, the Type Object pattern is a very useful and powerful tool for run-time specification of domain types.

The reification of the Describes relationship as a meta-object provides a mechanism for explicitly linking object types to objects. This strategy is similar to the approach taken for the Aggregation and Generalization relationships. The Describes meta-object provides developers with an explicit tool to dynamically create and alter domain types, and to modify domain behavior through run-time type-object alteration.

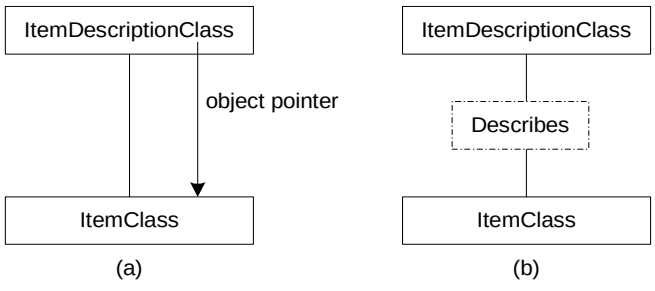

Fig. 4. The Type Object Pattern with Reified Describes Relationship

The Describes relationship does not exhibit the transitivity property. This implies that the propagation of some operations is not the default behavior since it cannot be inferred for the objects and their types. For example, versioning a type does not necessarily mean that objects of that type need to be versioned as well. In this particular case, it is the domain that dictates whether the versioning should be propagated or not. Thus, the Describes meta-object should include a mechanism for specifying propagation behavior.

Consequently, programmers can either accept the default relationship behavior or override it to implement domain-specific requirements. Figure 4 illustrates the transformation of the Type Object pattern with the use of a reified Describes relationship. The object pointer (in Figure 4a) is dropped, as it is insufficient to represent the semantics of the link relating objects and their types. Instead, the Describes meta-object (in Figure 4b) is used to manage and control the Type Object pattern relationship.

2.4 The Dependency Meta-object

The Publisher-Subscriber pattern models the dependency among related objects. To summarize the Publisher-Subscriber pattern, subscribers are automatically informed of any change in the state of its publishers. Thus, the association between the publisher and the subscriber manages and controls the communication and transfer of information between the two. Reifying the Publisher-Subscriber dependency association (hereafter referred to as the Dependency association), these mechanisms can be generically implemented and automatically enforced by the Dependency meta-object itself and taken out of the application code. This represents a significant breakthrough in the simplification of application codes and in the promotion of code reuse.

The reification of the Dependency relationship is significant in that it provides an explicit mechanism for handling change management and consistency control of data. The Dependency meta-object can be applied to base objects, to classes and types, to components of distributed systems and even to meta-objects and meta-classes. This leads to an homogeneous mechanism for handling inter-object dependencies within and between layers of multi-layered architectures.

The Event Channel of the Publisher-Subscriber pattern [11] and the Mediator of the Mediator pattern are realizations of the Dependency relationship. The Event Channel is an intervening object, which captures the implicit invocation protocol between publishers and subscribers. The Mediator encapsulates how a set of objects interacts by defining a common communication interface. By utilizing the Describes relationship, an explicit mechanism can be used to store and manage inter-object communication protocols. Figure 5 illustrates the use of reified Dependency meta-object in the Publisher-Subscriber pattern (a) and the Mediator pattern (b).

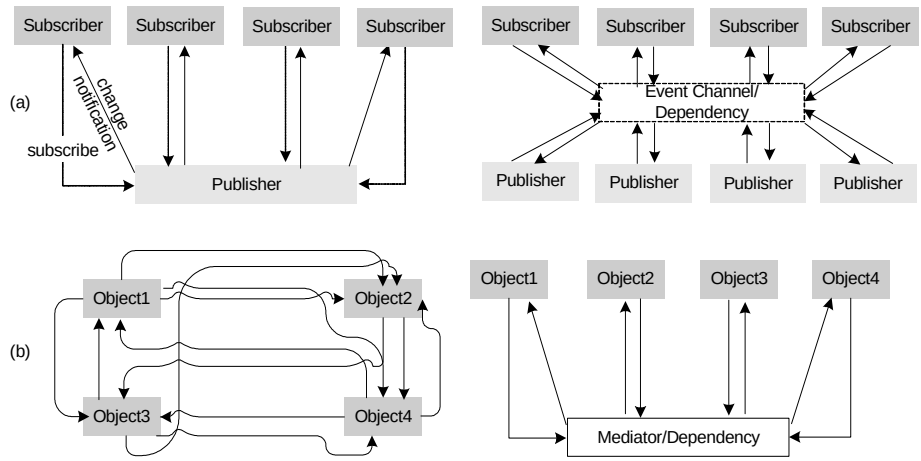

Fig. 5. The Event Channel and the Mediator as Reified Dependency

Reifying relationships as meta-objects is a fundamental step in the reification of design patterns. The four relationship meta-objects discussed above manifest the links that exist among the objects participating in the five design patterns listed in the introduction to this section. With the use of reified relationships, these five patterns can be modeled as a single graph, using the Graph pattern. Consequently, the five design patterns can be structurally reified as a Graph pattern, as shown in the next section, with the appropriate relationship meta-object to represent the semantics relating the individual pattern objects.

3 The Reified Graph Pattern

The graph and tree data structures are natural models to represent relationships among objects and classes. As the graph model is a generalization of the tree model, the graph model subsumes the tree semantics. Consequently, the graph specification is applicable to tree representations. The compositional organization of objects using the Aggregation relationship also forms a graph. Similarly, the class hierarchy using the Generalization relationship creates a graph. These two types of relationships are pervasive in computing, and the use of the Graph pattern to model both semantics provides a reusable solution for managing and controlling data compositions and class hierarchies and a valuable approach to enabling system evolution.

The way dependent objects are organized using the Dependency association also forms a graph. Dependency graphs are commonly maintained by application programs, and their implementations are often buried in them. The reification of the Dependency meta-object ‘objectifies’ the dependency graph and creates an explicit Publisher-Subscriber pattern. Consequently, the dependency graph is treated as an object, and can be accessed and manipulated like an object. The same argument applies to the Describes relationship found in the Type Object pattern. The link between objects and their types creates a graph. Reifying the Describes relationship

results in the reification of the Type Object pattern. With the reification of the Type Object pattern, the resulting graph object allows the dynamic management of object-type pairs. This capability is essential for environments that dynamically change.

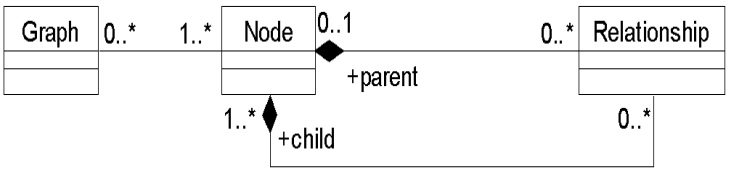

Fig. 6. An UML diagram of the Graph Meta-object

A UML diagram of the Graph meta-object is shown in Figure 6. The Node class represents the entities of the domain objects, classes, data, meta-data or components. The Relationship is the reification of the link between the Nodes. The aggregated links between the Node and the Relationship are bi-directional. Two roles are defined for the two aggregated associations - that of the parent, and that of the child. The parent aggregation, symbolized by the shaded diamond, implies that the lifecycle of the relationship is dependent on the lifecycle of the parent node. The child aggregation behaves similarly.

The use of reflection in making the Graph pattern explicit brings a number of advantages. First of all, it provides a *reusable* solution to data management. The reified Graph meta-object manages static data using Aggregation and Generalization meta-object relationships, and it makes persistent data dependencies using the Describes and Dependency relationships. As graph structures are pervasive in many domains, the capture of the graph semantics in a pattern and objectifying them results in a reusable mechanism for system designers and developers. Another benefit of having a single mechanism to represent compositions and dependencies is its provision for interoperability. With a single framework sitting on top of the persistent data, clients and components can communicate with a single known interface. This greatly simplifies the overall system design and architecture, thus improving system maintainability. Moreover, clients and components can be easily added as long as they comply with the graph interface.

Complexity is likewise catered for since related objects are treated singly and uniformly. The semantic grouping of related objects brings transparency to clients' code and the data structures provided by the Graph meta-object organize data into atomic units, which can be manipulated as single objects. Objectifying graph relationships allows the implicit and automatic propagation of operations throughout a single grouping. Another benefit in the use of the reified graph model is its reification of the link between meta-data and data. As a consequence, the Graph meta-object not only provides a reusable solution for managing domain-semantic groupings, but can also be reused to manage the links between layers of meta-level architectures.

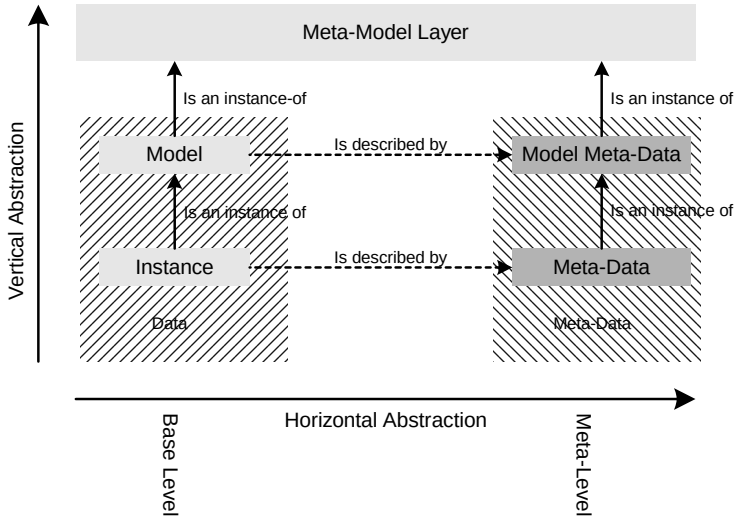

Fig. 7. Three-layer reflective Description-driven system architecture

4 Putting It All Together – Reified Patterns as the Basis of DDSs

This paper proposes that the reified Graph pattern provides the necessary building block in managing data in any DDS architecture. Figure 7 illustrates a proposed description-driven architecture. The architecture on the left-hand side is typical of layered systems such as the multi-layered architecture specification of the OMG [12]. The relationship between the layers is *an Instance-of*. The instance layer contains data that are instances of the domain model in the model layer. Similarly, the model layer is an instance of the meta-model layer. On the right hand side of the diagram is another instance of model abstraction. It shows the increasing abstraction of information from meta-data to model meta-data, where the relationship between the two is also *an Instance-of*. These two architectures provide layering and hierarchy based on abstraction of data and information models.

This paper proposes an alternative view by associating data and meta-data through description (the *is Described by* relationship). The Type Object pattern makes this possible. The Type Object pattern is a mechanism for relating data to information describing data. The link between meta-data and data using the Describes relationship promotes the dynamic creation and specification of object types. The same argument applies to the model meta-data and its description of the domain model through the Describes relationship. These two horizontal dependencies result in an horizontal meta-level architecture where the upper meta-level describes the lower base-level (see figure 8). The combination of a multi-layered architecture based on the Instance-of relationship and that of a meta-level architecture based on the Describes relationship results in a description-driven architecture (DDS). The reified Graph pattern provides a reusable mechanism for managing and controlling data compositions and dependencies. The graph model defines how domain models are created. Similarly,

the graph model defines how meta-data are instantiated. By reifying the semantic grouping of objects, the Graph meta-object can be reused to hold and manage compositions and dependencies within and between layers of a DDS (see figure 9). The meta-level meta- data are organized as a meta-level graph. The base-level data are organized as a base-level graph. Relating these two graphs forms a further graph whose nodes are related by the Describes relationship. These graphs indicate the reuse of the Graph pattern in modeling relationships in a DDS architecture.

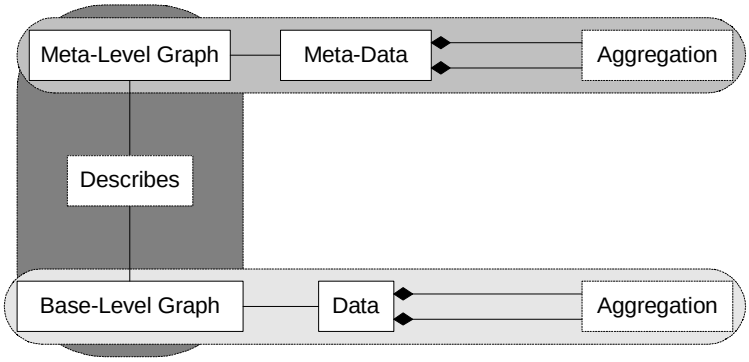

Fig. 8. The reuse of the Reified Graph Pattern in a description-driven system

5 CRISTAL as an Example of a Description-Driven System

The research which generated this paper has been carried out at the European Centre for Nuclear Research (CERN) based in Geneva, Switzerland. CERN is a scientific research laboratory studying the fundamental laws of matter, exploring what matter is made of, and what forces hold it together. Scientists at CERN build and operate complex accelerators and detectors whose construction processes are very data-intensive, highly distributed and ultimately require a computer-based system to manage the production and assembly of components. In constructing detectors like CMS, scientists require data management systems that can cope with complexity, with system evolution over time (primarily as a consequence of changing user requirements and extended development timescales) and with system scalability, distribution and interoperation.

A research project, entitled CRISTAL (Cooperating Repositories and an Information System for Tracking Assembly Lifecycles [4],[5]) has been initiated to facilitate the management of the engineering data collected at each stage of production of CMS. CRISTAL is a distributed product data and workflow management system that makes use of an OO database for its repository, a multi-layered architecture for its component abstraction and dynamic object modeling for the design of the objects and components of the system. CRISTAL is based on a DDS architecture using meta-objects.

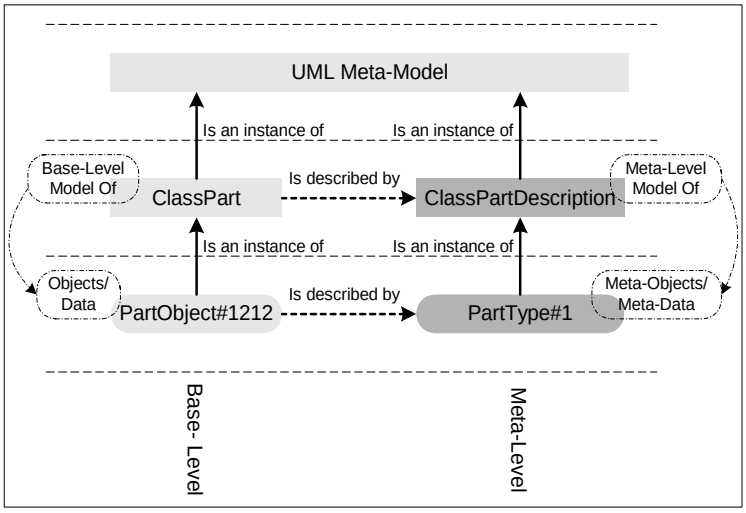

Fig. 9. The CRISTAL description-driven system architecture

The design of the CRISTAL prototype was dictated by the requirements for adaptability over extended timescales, for system evolution, for interoperability, for complexity handling and for reusability. In adopting a description-driven design approach to address these requirements, the separation of object instances from object description instances was needed. This abstraction resulted in the delivery of a three-layer description-driven architecture. The model abstraction (of instance layer, model layer, meta-model layer) has been adopted from the OMG MOF specification [13], and the need to provide descriptive information, i.e. meta-data, has been identified to address the issues of adaptability, complexity handling and evolvability.

Figure 9 illustrates the CRISTAL architecture. The CRISTAL model layer is comprised of class specifications for type descriptions (e.g. PartDescription) and class specifications for classes (e.g. Part). The instance layer is comprised of object instances of these classes (e.g. PartType#1 for PartDescription and Part#1212 for Part). The model and instance layer abstraction is based on model abstraction and *Is an instance of* relationships. The abstraction based on meta-data abstraction and *Is described by* relationships leads to two levels - the meta-level and the base-level. The meta-level is comprised of meta-objects and the meta-level model that defines them (e.g. PartDescription is the meta-level model of PartType#1 meta-object). The base-level is comprised of base objects and the base-level model which defines them.

Separating details of model types from the details of single parts allows the model type versions to be specified and managed independently, asynchronously and explicitly from single parts. Moreover, in capturing descriptions separate from their instantiations, system evolution can be catered for while production is underway and therefore provide continuity in the production process and for design changes to be reflected quickly into production. The approach of reifying a set of simple design patterns as the basis of the description-driven architecture for CRISTAL has provided the capability of catering for the evolution of a rapidly changing research data model.

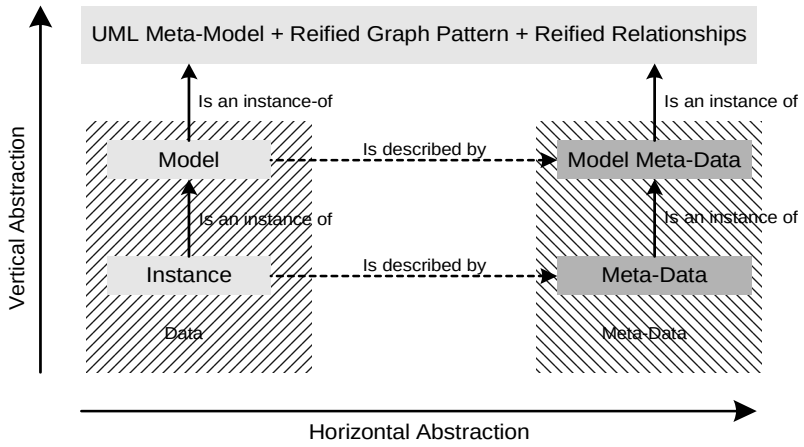

Fig. 10. Extending the UML Meta-model using a Reified Graph Pattern

In the two years of operation of CRISTAL it has gathered over 26 Gbytes of data and been able to cope with more than 20 evolutions of the underlying data schema without code or schema recompilations.

6 Conclusions

As shown in figure 10, the reified Graph pattern and the reified relationships enrich the meta-model layer by giving it the capability of creating and managing groups of related objects. The extension of the meta-model layer to include constructs for specifying domain-semantic groupings is the proposition of this paper. The meta-model layer defines concepts used in describing information in lower layers. The core OMG/UML meta-model constructs include Class, Attribute, Association, Operation and Component meta-objects. The inclusion of the Graph meta-object in the meta-model improves and enhances its modeling capability by providing an explicit mechanism for managing compositions and dependencies throughout the architecture. As a result, the reified Graph pattern provides an explicit homogeneous mechanism for specifying and managing data compositions and dependencies in a DDS architecture.

This paper has shown how reflection can be utilized in reifying design patterns. It shows, for the first time, how reified design patterns provide explicit reusable constructs for managing domain-semantic groupings. These pattern meta-objects are then used as building blocks for describing compositions and dependencies in a three layer reflective architecture - the description-driven systems architecture. The judicious use and application of the concepts of reflection, design patterns and layered models create a dynamically modifiable system which promotes reuse of code and design, which is adaptable to evolving requirements, and which can cope with system complexity. In conclusion, it is interesting to note that the OMG has recently announced the so-called Model Driven Architecture as the basis of future systems

integration [14]. Such a philosophy is directly equivalent to that expounded in this and earlier papers on the CRISTAL description-driven architecture.

Acknowledgments

The authors take this opportunity to acknowledge the support of their home institutes.

References

1. G. Kiczales, “Meta-object Protocols: Why We Want Them and What Else Can They Do?”,Chapter in OO Programming: The CLOS Perspective, pp 101-118, MIT Press, 1993.
2. E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.
3. J. Yoder, F. Balaguer, and R. Johnson., “Architecture and Design of Adaptive Object-Models”. Proc of OOPSLA 2001, Intriguing Technology Talk, Florida. October 2001.
4. F. Estrella et al., “Handling Evolving Data Through the Use of a Description Driven Systems Architecture”. LNCS Vol 1727, pp 1-11 Springer-Verlag, 1999
5. F. Estrella et al., “Meta-objects as the Basis for System Evolution”. Proc of the Web Age Information Management (WAIM’2001) conference. Beijing China, June 2001.
6. F. Demers and J. Malenfant, “Reflection in Logic, Functional and Object-Oriented Programming: A Short Comparative Study”, Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI), Workshop on Reflection and Meta-level Architectures and their Applications in AI, Montreal, August 1995.
7. B. Woolf and R. Johnson, “The Type Object Pattern” in Pattern Languages of Program Design 3, Addison-Wesley, 1997. Originally presented at the Third Conference on Pattern Languages of Programs (PLoP), 1996.
8. F. Estrella, "Objects, Patterns and Descriptions in Data Management", PhD Thesis, University of the West of England, Bristol, England, December 2000.
9. M. Blaha, “Aggregation of Parts of Parts of Parts”, Journal of Object-Oriented Programming (JOOP), September 1993.
10. M. Blaha and W. Premerlani, “Object-Oriented Modeling and Design for Database Applications”, Prentice Hall, 1998.
11. F. Bushmann, et.al., “Pattern-Oriented Software Architecture: A System of Patterns”, John Wiley & Sons, 1996.
12. The Object Management Group (OMG), URL http://www.omg.org
13. The Meta- Object Facility (MOF) Specification, URL: http://www.dstc.edu.au/Products/CORBA/MOF/.
14. OMG Publications., “Model Driven Architectures - The Architecture of Choice for a Changing World”. See http://www.omg.org/mda/index.htm