Adex - A meta modeling framework for repository-centric systems building

S. Sreedhar Reddy
Tata Consultancy Services
54-B, Hadapsar Industrial Estate
Pune 411013, India
+91 20 687 1058
sreedharr@pune.tcs.co.in

Janak Mulani
Canoo Engineering AG
Kirschgartenstrasse 7
CH-4051 Basel
+41 61 228 9444
janak.mulani@canoo.com

Arun Bahulkar
Tata Consultancy Services
54-B, Hadapsar Industrial Estate
Pune 411013, India
+91 20 687 1058
arun@pune.tcs.co.in

ABSTRACT

Organizations are increasingly recognizing the need for building and maintaining their information systems around a central repository, which is essentially an information system of information systems. Models are the artifacts with which information systems are described. A repository should therefore provide a modeling framework that is flexible enough to support models of different methodologies, different implementation and deployment platforms, and enterprise specific domain knowledge, at various levels of granularity. In addition it should be possible to use the models for automation of code generation and coordination of various processes of system building and maintenance. Since information systems evolve over time and also since systems typically have many variants, it is essential that a repository should provide good versioning and configuration management facilities.

Adex is a modeling tool, with a meta modeling framework, which fulfils the above mentioned requirements. The meta models that come as part of standard modeling tools are often inadequate for capturing all the domain specific abstractions of an enterprise's data and processes. Adex's 'meta modeling' framework enables natural categorization of abstractions through creation and customization of enterprise specific meta models, diagramming notations, and methodologies. Unlike most standard modeling tools, Adex provides constructs for componentization and versioning of models. More importantly, Adex has primitives for defining notions of compatibility of components and completeness of an aggregate of components.

Adex is a customizable, multi-user, secure and open modeling environment. It has been successfully used in a wide range of large (> 100 persons) and small (< 10 persons) projects. Its 'meta modeling' framework has been used to create meta models for component-based information system modeling and generation tools, program analysis and reverse engineering tools, and performance modeling and simulation tools.

Keywords: meta modeling; enterprise data modeling; component-based modeling; repository.

ADVANCES IN DATA MANAGEMENT 2000
Krithi Ramamritham, T.M. Vijayaraman (Editors)
Tata McGraw-Hill Publishing Company Ltd.
©CSI 2000

1. INTRODUCTION

Today's enterprises develop, deploy and maintain a large number of complex information systems that need to evolve constantly to keep pace with the changing business and technology requirements. Enterprises use repositories for storing, and subsequently leveraging, the descriptions of diverse information systems and various complex relationships present in them. This information about information systems is stored in terms of interrelated models. A model is an instance of the abstractions from the domain of discourse of an information system. Every information system needs to capture abstractions that are unique to its domain. Every stakeholder in an information system has his own view of the information system, for example, analysis view, design view, code generation and construction view, product definition view, and business process view. The views and abstractions of an information system are specified as a meta model. It is the meta model that defines what will be modeled and how it will be modeled.

The views and abstractions of an information system are not static. In fact, they evolve continuously as the modelers gain deeper understanding of the information system and its domain over a period of time. The abstractions also evolve across information systems to capture and promote re-use. Refinement and evolution of abstractions is needed for capturing the experience, learning, new abstractions, and for designing new procedures and tools (e.g. code generators for new technology platforms) that actualize the models.

Most standard meta models, such as the ER Model [1], the UML Model [2], and the Process Model (DFD), are inadequate for capturing all the domain, view, and life cycle specific abstractions of an information system. Most modeling tools that are available in the market come with a predefined set of standard meta models and hence are insufficient for seamlessly modeling all aspects of diverse information systems. Some modeling tools like Rational Rose [3] allow limited changes to their meta model through mechanisms like StereoType, while others like System Architect [4] allow extensions to meta model through renaming of predefined objects and specification of "name-value pairs" in a text file. But, in these systems, the meta model is not 'first class', i.e., it cannot be modeled and hence is not available for browsing and extension in the real sense. Thus, there is a clear requirement for a framework that treats meta models as 'first class' models, i.e., a framework that allows modeling of a meta model for easy definition and evolution. Adex's 'meta modeling' approach provides a framework for defining, evolving, and installing meta models in repositories.

Large information systems have large models. Large models with inter-dependencies between model elements hamper concurrent development and maintenance leading to reduced productivity. To manage the complexity of size, information systems are normally partitioned into modules or components with well-defined interfaces. The associations between the entities contained in the components denote the relationships between the components, which define the notions of component compatibility, configurability, and completeness for the information system in terms of its components. A repository should therefore have constructs for modeling information system components, their inter-dependencies and their aggregation. As the information system evolves, its models also evolve. A successful information system typically also has many variants specific to various delivery platforms, geographical regions, and multiple development streams. Versioning is a natural consequence of evolution and variation. Therefore, there is a clear need to support base-lining and versioning of an information system's models vis-à-vis different stages of the information system's life cycle and its variants. Adex provides constructs and mechanisms for componentization, versioning, and configuration management of information system models, both at the model and meta model level.

This paper presents how Adex uses meta-modeling approach to build a flexible, open and robust modeling framework and how this framework can be used to build enterprise information systems.

The following sections describe Adex's 'meta modeling' framework; mechanisms for partitioning and versioning of information system models; Adex's approach to designing and evolving an information system repository; and Adex's architecture in terms of its components.

2. META MODELING

An information system's model is a description of the system and of its relationships with other systems, in terms of a meta model. A meta model in Adex consists of:

- A model schema
- A meta model specific diagramming notation
- Rules or constraints for meta model and domain specific consistency and completeness checks on models
- Meta model specific tools to support the modeling process and generation of code and reports

The modeling framework comprises facilities for defining meta models and their corresponding diagramming notations, constraints, views, and tools for supporting the modeling process. In Adex, a meta model is modeled and browsed in the same environment in which the information system models are modeled. Thus, within Adex, meta models are self-descriptive and 'first-class'.

The following sections describe the various models in Adex followed by the specification of diagram notation, rules and constraints, and scripts to support the modeling process and generation tools.

2.1 Models in Adex

Adex has models at three levels (figure 1):

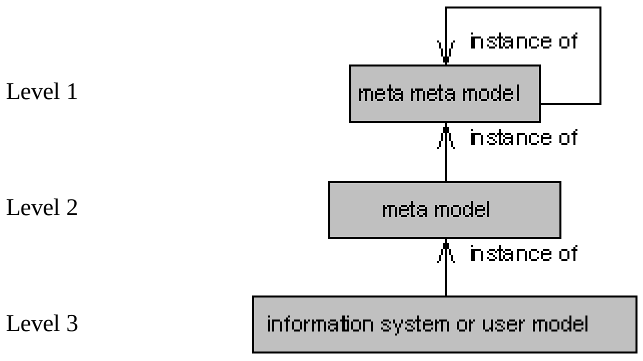

Figure 1. Models in Adex

A model at each level is an instance of the model at the previous level. The model at level 1 is an instance of itself.

2.1.1 Meta meta model

The meta meta model (figure 2) is the base model in Adex. It is the schema for modeling meta models. The meta meta model is capable of describing itself, i.e., it can model itself. It is the root of the instantiation hierarchy.

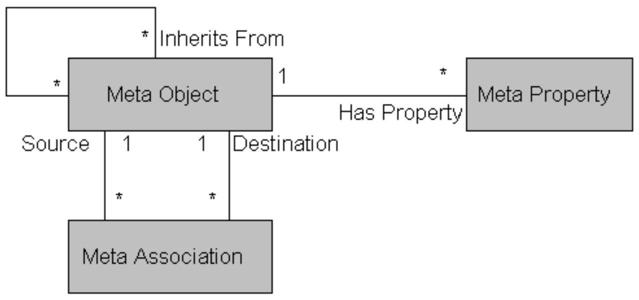

Figure 2. Meta meta model

The meta meta model consists of objects, associations and properties:

Objects in Meta Meta Model	Properties of Object
MetaObject	Name, Description, AbstractConcerte

MetaProperty	DataType = Char, Number, Binary
	Size = Size of Char String and Number

MetaAssociation	Forward and Reverse Name Source and Destination Optionality = Association is optional or mandatory for source or destination object Source and Destination Cardinality = One or Many Owner of the Association = Source or destination object is the owner of the association
Associations in Meta Meta Model	**Properties of Association**
MetaObject InheritsFrom MetaObject	A MetaObject inherits associations and properties from another MetaObject. Many to Many; Optional
MetaObject Has MetaProperty	MetaObject has MetaProperties. One to Many; Optional
MetaObject SourceOf MetaAssociation	MetaObject is source of a MetaAssociation. One to Many; Mandatory for MetaAssociation
MetaAssociation HasDestination MetaObject	MetaObject is destination of a MetaAssociation. One to Many; Mandatory for MetaAssociation

2.1.2 Meta model

A meta model is an instance of the meta meta model. It consists of Meta Objects with associated Meta Properties, and Meta Associations defined between Meta Objects. A meta model defines the structure and semantics for the information system model. It captures the abstractions from an information system's domain, stakeholder views, life cycle stages, and tool specific requirements.

2.1.3 Information system model or User model

The information system model, also referred to as the user model, is an instance of a meta model. It captures the description of the information system from various points of view as specified by the meta model. The analysis model, design model, construction model, dataflow model, and workflow model are some of the examples of information system models. This knowledge about the information system, stored in a repository, can then be leveraged by an enterprise for the purposes of analyses, communication, reuse, and for constructing tools that actualize it as code and reports.

2.1.4 Examples

MasterCraft [5] – an integrated, model-driven software development environment - uses Adex to define and maintain its model repository. We will use MasterCraft as a running example to illustrate various concepts throughout the paper. MasterCraft meta-model extends the standard object model in a number of ways, to capture various aspects of an information system like database design, GUI model, process model, rulebase model, etc. MasterCraft meta-model is an illustration of the power of modeling meta models, in terms of the meta meta model, which enables aggregation and creation of relationships between meta data from diverse domains. In traditional approaches, each of these domains is implemented in a separate tool, in a disjointed manner. The meta-modeling approach allows integration of these diverse domains, thus facilitating an integrated tool environment. Figure 3 and the accompanying table describe a portion of MasterCraft's meta model as an instance of the meta meta model. This meta model represents the Object Model and Relational Model related abstractions for an information system. In this model the standard Object meta model has been extended to include abstractions from the domain of Database design.

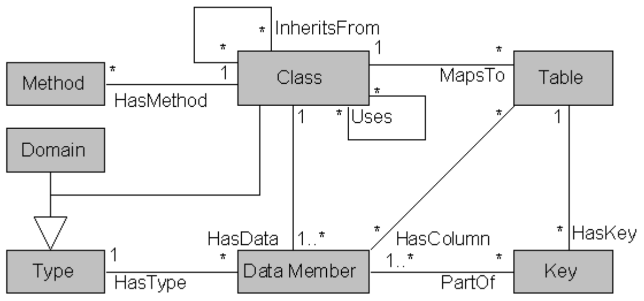

Figure 3 Object-Relational Meta model.

Meta Meta Model – Schema	Meta Model – Instance
MetaObject	Class, Type, Domain, Data Member, Table, Key, and Method
MetaAssociation	'HasData', 'HasType', 'HasMethod', 'MapsTo', 'HasColumn', 'HasKey', 'InheritsFrom' and 'Uses'
MetaProperties (not shown in figure 3)	For Class: Abstract, Persistent For Domain: Size, Type, Precision

In the above example, the MetaObjects 'Class' and 'Domain' are derived from 'Type'; therefore, they inherit the MetaProperties and the MetaAssociation ('HasType') of MetaObject 'Type'. Each MetaAssociation has values defined for properties: Forward and

Reverse Name, Source and Destination Cardinality, Source and Destination Optionality and Association Ownership. For example, the association 'Class HasMethod Method' has cardinality 1:M (Source-One, Destination-Many) and optionality of O:M (Source Mandatory for the existence of the destination object and Destination Optional, i.e., a 'Method' must be associated to a 'Class' but not necessarily vice versa).

Association ownership is a property of the MetaAssociation. This property is defined for each MetaAssociation to declare which of the source or destination object is the owner of the association. The owner of an association is defined on instantiation of the MetaAssociation in the user model. The owned-association is then treated like a property of the owner object. In the example meta model, for the MetaAssociation 'Class Uses Class', the source Class is specified to be the owner of the association. This means that a Class depends on or is incomplete without its associated Class that it 'Uses', and that the used Class should go together with the user Class. An association, with its ownership property, is the primitive used by Adex to define dependence relationship amongst objects and their container components. It is also used to define the completeness of a configuration with respect to its contained components. These concepts are described in Section 3.

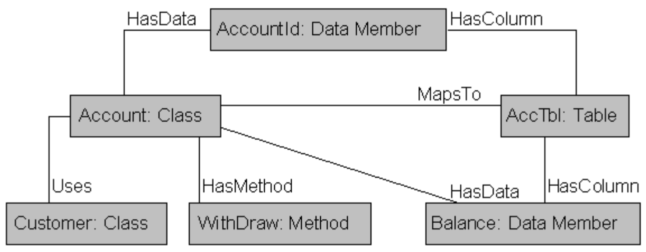

Figure 4. Information system model.

Figure 4 shows an information system's Object and Relational Database Design model that is an instance of the meta model of figure 3. The objects 'Account' and 'Customer' are instances of MetaObject 'Class'. The Class 'Account' has DataMembers 'AccId' and 'Balance'. It maps to Table 'AccTbl'. The Class 'Customer' uses Class 'Account' and is the owner of its association with Class 'Account'.

In an information system model, objects are instances of MetaObjects and an object's properties are instances of MetaProperties. Every object in the repository has a unique identity called ObjectId. Properties of an object carry the object's ObjectId. Associations between objects are instances of MetaAssociations and carry the ObjectIds of the source and destination objects.

As another example, figure 5 and the accompanying table describe a meta model representing a bridge between XML and UML. It maps the XML DTD structure [6] to the UML Class structure [2]. In the above example, the MetaObject 'XMLElement' and the MetaAssociation 'childElement' capture the structure of XML DTDs. XMLElement has MetaProperties to specify separator and occurrence. XMLAttribute specifies the attribute list for an element. XMLElement and XMLAttribute are mapped to Class and Attribute in UML. Using this mapping, it is possible to derive XML DTDs from an UML model and vice versa. The same mapping can also be used to generate code, which can translate between XML messages and application objects.

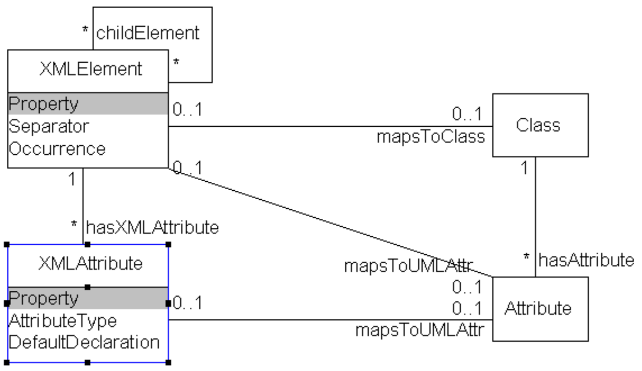

Figure 5. Meta model for XML-UML mapping.

Meta Meta Model – Schema	Meta Model – Instance
MetaObject	XMLElement, XMLAttribute, Class, and Attribute
MetaAssociation	'childElement', 'hasXMLAttribute', 'hasAttribute', 'mapsToClass', and 'mapsToUMLAttr'
MetaProperties	For XMLElement: Separator, Occurrence For XMLAttribute: AttributeType, DefaultDeclaration

2.2 Diagramming notation

Information system models are typically depicted using diagrams for ease of understanding and for presentation of overview. Diagrams are drawn using a notational language of icons, where an icon represents an object or an association from the information system's meta model. The notational language needs to be enhanced with new icons as new abstractions are added to the meta model. Adex provides facilities to create icons and map them to the MetaObjects and MetaAssociations from the meta model. Figure 6 and the accompanying table show mapping of various icons to meta model entities. In the figure, MetaObject 'Class' is mapped to rectangle and bar icons, MetaAssociation 'Inheritance' is mapped to an arrow icon, and MetaObject 'Message' is also mapped to an arrow icon. Adex's diagrammer interprets this mapping, at run time, to present abstractions as icons for drawing information system models.

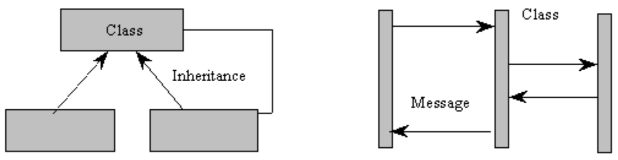

Figure 6. Icons to Meta Model mapping.

Sample symbol specification:	Sample map specification:

SYMBOL CLASS, 1004, 180, 120
{
DESCRIPTION "Class"
SHAPE
{
RECTANGLE 0, 0, 180, 120 ;
TEXT 1, 0, 0, 160, 120, "Class", 0
:TX_HEIGHT=10
:TX_FONT="Arial"
}
BOUNDARY
{
0, 0, 0, 120, 180, 120, 180, 0, 0, 0
}
}
	symbol Class
	{
	icon = "1004"
	MetaObject = Class
	symbolprop
	{
	"1" =
	MetaProperty(Name);
	}
	}

2.3 Constraint specification

A meta model is specified in terms of MetaObjects, MetaProperties and MetaAssociations. Cardinality and optionality constraints are specified on MetaAssociations. However, logical constraints on properties and associations cannot be specified using this notation. For example, a constraint such as “An Abstract Class can not be Persistent and should not be stored (MapsTo) in a Table” cannot be captured in the meta model. Predicate logic is required to express such constraints.

Adex provides a declarative Rule Language to specify predicates over the objects, associations and properties in the meta model. For example, the constraint mentioned above can be specified as follows:

```
/* Abstract class cannot be persistent and should
not be mapped to a table */
Rule : 1 :Abstract_Class
{
    Class c;
    Table t;
    APPLYTO(c);
    /* Apply this rule to an instance of Class */
    IF (c.Abstract = `A’) THEN
    /* Pre-condition for constraint evaluation */
    /* Constraint … */
    c.Persistent <> `P’
     AND
     NOT_EXISTS (t | REL(c, t, 'MapsTo'));
     ISSUES(I0001, c);
    /* raise issue# I0001 for object bound to c */
}
```

A set of rules forms a rulebase. A rulebase is executed by Adex’s rulebase engine to check consistency, completeness and the well-formedness of the information system model with respect to the meta model. A rulebase can be executed online or in a batch mode. Constraint violations are raised as issues against objects. The issues for an object can be removed by resolving the constraint violations.

2.4 Scripting

A meta model specifies what can be modeled structurally; however, by itself it cannot specify dynamic behavioral aspects of the model. Adex provides a ‘meta model aware’, object oriented scripting language that can be used to impart dynamic behavior to the model. The scripts, when linked to MetaObjects and MetaAssociations in the meta model, act as class methods that are executed at the time of creation, modification, and deletion of instances. Such scripts can be used to perform desired actions on occurrence of these events; some of the typical actions are: initialization of instances, value computation and propagation to related instances, performing validations, and triggering external actions. Scripts can interface with external ‘C’ programs, allowing custom software and third-party utilities to be linked in. Adex’ scripting feature can be effectively utilized to integrate modeling activity into the overall process of system building.

The script language’s model navigation and input/output constructs enable writing of information system specific code and report generators. The script interpreter is capable of operation from the repository database or from an exported CDIF (CASE Data Interchange Format [7]) file. A sample script to generate C++ header from Object Model (Figure 3) is shown below:

```
FOREACH c IN Class
{ // : => print - class < classname>
    : class  $c.Name {
    // For each DataMember, associated to a Class
    by association `HasData’
    FOREACH d IN c.HasData->DataMember
    { // print DataMember type and name
    : \t$d.Type $d.Name ;
    }
    :}
}
```

3. PARTITIONING and VERSIONING

Information systems are usually broken up into components to manage complexity and enable concurrent development and maintenance. Compatible components are then put together to configure a complete information system. Adex provides constructs for modeling and aggregating the component structure of an information system. Also, the primitives to support the notion of completeness of a configuration of components are available. The model of an information system evolves as the system progresses through the stages of its life cycle, i.e., different versions of the model are created and baselined during each stage. Adex provides mechanisms to create and manage the versions of models. Figure 7 shows the Adex's model for componentization and versioning. This model comes as a predefined meta model in Adex repository. The users instantiate this model in order to partition, version, and configure their information system model.

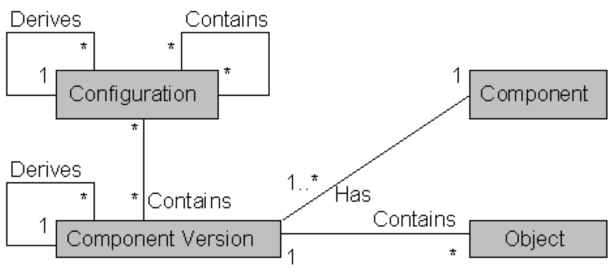

Figure 7. Model for partitioning and versioning.

R Conradi and B Westfechtel have published a detailed survey and classification of various version models [8]. Adex's version model provides *extensional versioning* with *configuration* as the recursive composition structure and *component* as the versioned entity, with predefined composability constraints.

The following sub sections describe the model in detail.

3.1 Component

Adex provides a construct called 'component' for partitioning an information system model. Each module of an information system can be mapped to one or more components. The model for each module can then be placed in its components. A component is a container for model elements like objects, properties and associations. An object on its creation is placed in a component and carries the identity of the component.

Versioning of models is done at the level of a component. A component is realized by its versions. The first version of a component is created when the component is created. All other versions are derived, directly or indirectly, from the first version. Versions of compatible components are assembled, within a configuration, to represent the complete model of an information system.

An object belongs to a component and is versioned with the component. Thus, an object gets the version number of its container component, but two versions of an object will always have the same ObjectId. When an object is contained in a component version, its properties and the associations owned by it are also contained in that component version. Component is a typeless construct in that an instance of any meta object can be placed in a component. Adex implements versioning at the component level and not at the object level because an object by itself is usually too fine-grained an entity for versioning. An object in a model is always seen in the context of its associated objects. Components provide a mechanism for grouping objects for the purpose of versioning; grouping can be done at any required level of granularity.

Associations can either be intra-component or inter-component. An association between objects belonging to different component versions is an inter-component association. An inter-component association establishes a relationship between two component versions and it belongs to the component version to which the owner object of the association belongs. Inter-component associations can exist only in the context of a configuration.

A component version guarantees change isolation. Changes made to the objects in a component version are not visible in other versions of the component or other components. Conceptually, a new version of a component contains copies of the objects of the version from which it is derived. A component enables concurrent development of different parts of the model. A component version can be shared between various configurations.

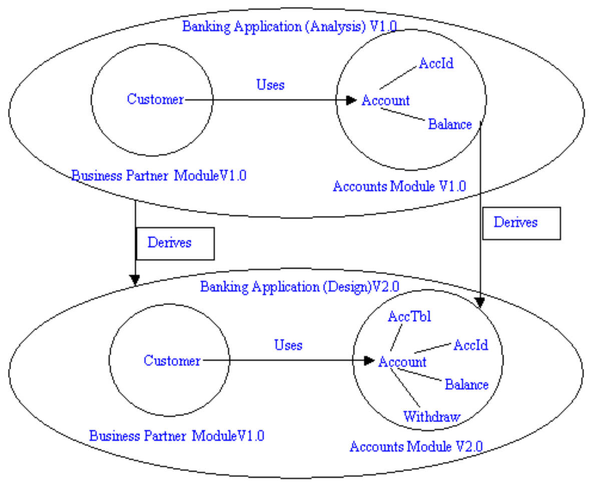

Figure 8. Components, versions and configurations.

Figure 8 shows two configurations (Banking Application Analysis and Design), versions of components (Business Partner and Accounts) and objects (Customer, Account, Balance, etc.). Two versions of Account object can be seen. The first version in the Analysis Configuration has associations with two Attributes: AccId and Balance. The second version of the Account object in the Design Configuration has associations with an Operation (Withdraw) and a Table (AccTbl) in addition to having associations with the Attributes.

The version management facility provides support for tracking the history of changes to the components, branch versioning, selection of different versions of different components to form a meaningful configuration, and 'diff and merge' of models contained in different versions of components.

3.2 Configuration

Adex provides configurations as containers for assembling different versions of components. A configuration represents the notion of 'a complete unit of work'; it can be seen as a product; as an input to a set of code generation tools, or as an input to a 'make' utility. A configuration is a complete set of compatible components and provides the context for creating relationships between various components via the inter-component associations. Only one version of a given component can belong to a configuration. A configuration can contain other configurations. Since a configuration is defined as an independent and complete entity, a container configuration can 'use' or 'depend on' the contained configuration but not vice versa.

A configuration is realized through its versions. Unlike component versions, two versions of a configuration can overlap, i.e., they can share component versions. For example, in figure 8, the component version 'Business Partner Module V1.0' is shared by the analysis and design configurations. The versioning of configurations is provided for the purpose of tracking the version history only. Thus, configuration versions are designed to support sharing, while component versions are designed to support change isolation.

3.3 Completeness of a Configuration

The "ownership" property of associations (described in section 2.1.4) helps define and enforce the notion of completeness of a configuration with regard to the component versions it contains. The inter-component associations in a configuration establish 'consumer-supplier' relations between component versions. A component version that owns an inter-component association depends on the associated component version. The owner object (and hence its container component) of such an association is deemed 'incomplete' without the associated object (and hence its container component). Thus, a configuration is complete only if it contains a complete set of related component versions that satisfy all the 'owned' inter-component associations within the context of the configuration. Adex enforces this notion of completeness for all configurations ensuring that all the 'consumer-supplier' relations are satisfied when components are assembled into a configuration. Figure 9 shows examples of complete and incomplete configurations. The 'owned' association 'Uses' of Class Customer is not satisfied in configuration A, hence making it incomplete.

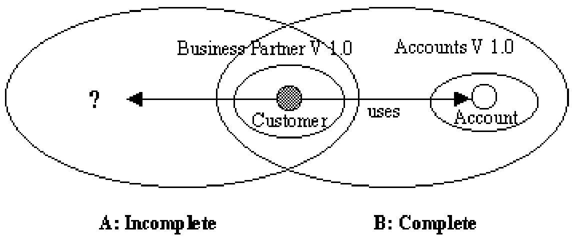

Figure 9. Completeness of a configuration

3.4 Compatibility of components

Inter-component associations in Adex provide a mechanism to establish and enforce compatibility semantics between two component versions. A version of an object can replace another version of the object in an association. Hence, two versions of an object are deemed compatible. An object is shared if its component version is shared in more than one configuration. Such a shared object must appear the same, with respect to its properties and owned-associations in all the sharing configurations. Changes made to a shared object within a configuration must be reflectable in all the sharing configurations. Specifically, addition and deletion of owned-associations to a shared object should be possible in all the sharing configurations. The configuration A in figure 9 shows an invalid sharing condition for class 'Customer' in the shared component 'Business Partner V1.0'. A set of component versions are deemed compatible if they can be put together in a configuration without causing any sharing condition violation with respect to inter-component associations. These notions of sharing and compatibility are always enforced by the system.

The notion of component compatibility in Adex is based on the premise that any two versions of an object are compatible with each other. This notion of compatibility is designed to allow maximum possible flexibility in component composition by enforcing only the minimum required constraints. Applications can use rulebase and scripts to enforce a stricter notion of object version compatibility where required.

3.5 Diff and Merge

Quite often a component needs to be developed concurrently by independent teams in a non-interfering way. This can be achieved by deriving a branch version of the component. At the end of the development effort, the branch version of the model needs to be merged back into the main stream. Adex provides a facility to compare ('Diff') and merge models contained in two component versions or two configurations. The 'Diff' operation can be done at the level of configurations, component versions, or objects.

Two versions of an object are compared based on their ObjectIds and in terms of their property values and associations. Often, comparison of two objects by themselves is not of much value unless it is done in the context of its associated objects. Such a context is also necessary while merging versions of an object. In general, in a given model, some objects play the role of primary objects and others play the role of companion objects. For example, in the OO model, Class is a primary object and Data Member is a companion object. Two Data Members should only be compared in the context of the Classes to which they belong. In Adex, such context sensitive comparison and merge operations can be performed by specifying object-association graphs or patterns. An example pattern for comparing classes in two components could be: 'Class-HasData-DataMember', 'Class-HasMethod-Method'. It is also possible to specify the list of properties to be compared. In addition to providing a context, a pattern also limits the scope of comparison to models of interest. Adex also provides a 'Patternless Diff' facility wherein each object in the source container is compared with its counter part in the destination container.

4. METHODOLOGY

Adex prescribes a methodical approach for the creation of a flexible enterprise repository and for managing the models in that repository.

Adex provides two separate repositories for managing the meta models and the information system models. As shown in figure 10, a meta model is defined in the Meta Model repository and the information system model (User Model) is defined in the User Model Repository. A meta model needs to be installed in the User Model repository before the information system modeling can begin.

The following sections describe the steps to be followed by the meta modelers and modelers of an information system. The steps describe the whole cycle comprising the creation of a meta model, the instantiation of the meta model as an information system model, the evolution of the information system model through the various stages of its life cycle, the evolution of the existing meta model itself, and the instantiation of the new version of meta model.

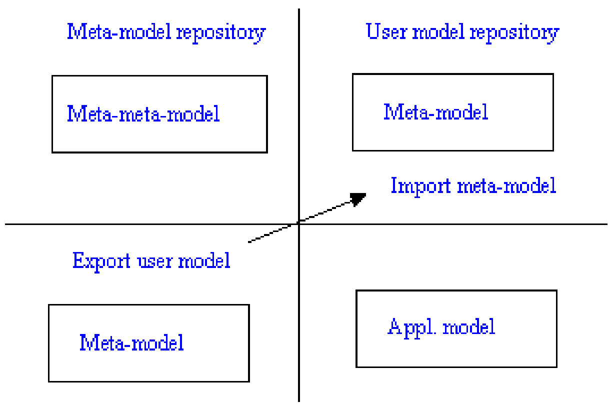

Figure 10. Meta model and user model repositories.

4.1 Steps for Meta Modeling

i. Define a meta model: Instantiate the Meta Meta Model to create Meta Objects, Meta Properties and Meta Associations. To manage the complexity of a large meta model, group the related meta model elements into separate components. Appropriate subsets and versions of the meta model components can then be configured and made available to diverse classes of information system modelers.

ii. Create diagram notation: Create notational icons and map them to the Meta Objects and Meta Associations from the meta model.

iii. Specify constraints for consistency and completeness checks: Write Rules and Scripts for validation of the information system model with respect to the meta model.

iv. Specify views on the meta model: Creating views on the meta model enables logical partitioning of a large meta model in the User Model repository. Different view specifications can be used for different stages of the modeling process. Views are used to customize the user interface by displaying only the relevant part of the meta model to better meet the needs of different classes of users.

v. Create tools to support the modeling process and generation: Write scripts to perform life-cycle-stage specific validations and actions. Also, specify scripts for generating code and reports from the information system model.

4.2 Steps for User Modeling

i. Install the meta model into User Model repository: The meta model is installed in the user model repository, from the meta model repository, using the meta model import utility (figure 10). Adex's meta model import utility merges, with certain restrictions, the changed meta model with the existing meta model while safeguarding the existing information system model. Warnings are issued if changes to an existing meta model's objects, associations and properties are going to affect their instances in the information system model. Install the Diagram Types, Rules and Scripts in the User Model Repository using the configuration file.

ii. Define a strategy for partitioning the model: Decide how the model is going to be partitioned in terms of Configurations and Components and at what granularity the Model is going to be versioned. The versioning granularity, i.e., which model elements need to be treated as a group for the purpose of base-lining, will affect the choice of components. Create the necessary components and configurations.

iii. Define access security: Create users and groups, and grant them write access to edit the configurations and components.

iv. Create models: Create instances of the MetaObjects and MetaAssociations using the Browser and the Diagram Editor. Group the related objects into corresponding components as per the partitioning strategy.

v. Create baselines of model: At each stage of the development cycle, create baseline of the model by versioning and freezing configurations and components. Create new versions of components and configurations, if required, for the models of the next stage of the life cycle.

vi. Validate the baseline model: Validate the baseline model by running the rulebase and scripts to check the consistency and completeness of the model

vii. Generate code and reports: Leverage the information captured in the model by running code and report generator scripts on the baseline configurations or components.

5. TOOLKIT AND ARCHITECTURE

Adex has a set of tools (figure 11) to support the meta modeling and information system modeling processes described in section 4. Adex provides two separate repositories for storing meta models and information system models (figure 10). An Icon-Editor tool is provided to define diagram notations and map them to the meta-model. Model browser and diagram editor (figure 12) are provided to create, maintain, browse, and draw diagrams of objects, properties and associations in the model, both in the meta model and user model repositories. The Rulebase Compiler and the Rulebase Engine enable specification and execution of the Rulebase. The Script Engine is used for executing the validation, process, and code and report generation scripts. The Script Engine can operate on the model data in the repository and also from a CDIF file. Configuration Management Module enables creation, management, versioning, archival, comparison, and merging of configurations and components. Adex provides Export and Import utilities, for models and meta models, for converting and reading the models as CDIF and XML files. The Administration utility enables management of users, groups, and their access and operational rights on configurations and components.

Adex has a two-tier client/server architecture. The topmost layer is the GUI comprising Diagramming and Browsing interfaces. The GUI layer makes use of the Repository API layer that implements the model semantics and provides a C++ API to the following services: meta model access, user model access, component, version and configuration management, definition of user security and access control at component and configuration level, rulebase engine, script engine, view definition, export and import of models. The Repository API accesses data through the Data Manager layer, which is implemented using ODBC for portable database connectivity to database platforms like Oracle, SQL*SERVER and MS-ACCESS.

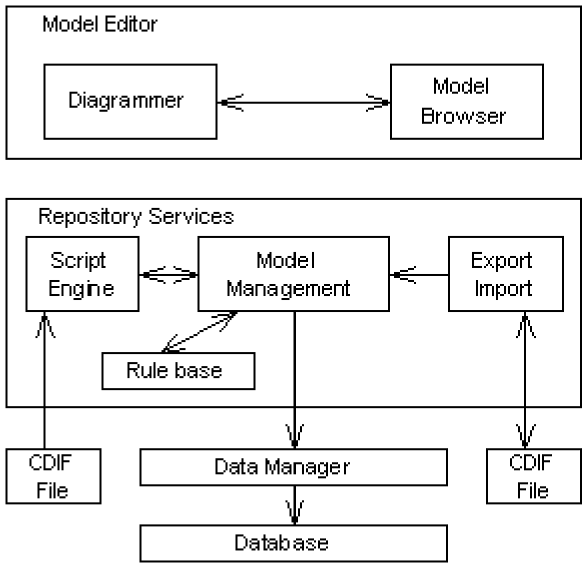

Figure 11. Tools and services

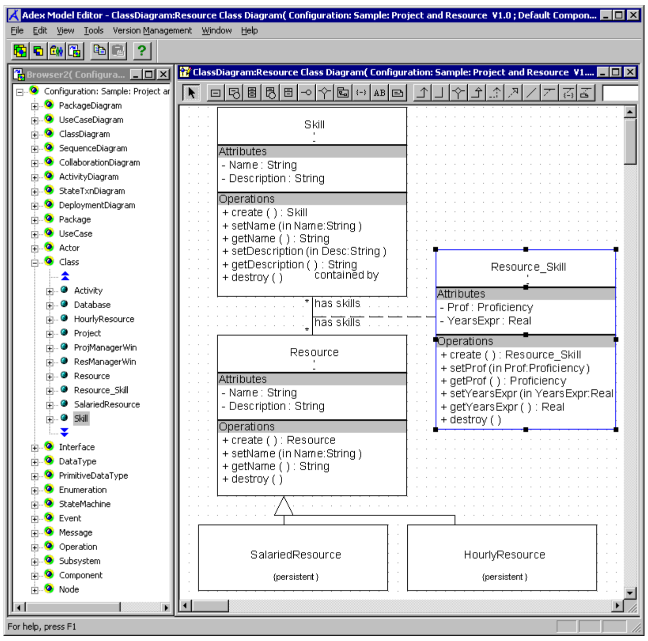

Figure 12. Browser and Diagram Editor

6. DISCUSSION and CONCLUSION

Adex provides a flexible meta modeling framework. It also supports componentization, versioning and composition of information system models with primitives to support the notions of compatibility and completeness.

The Adex's hierarchy of models is similar to the one specified by Meta Object Facility (MOF) from OMG [9]. The MOF specifies a four layer meta data architecture comprising of Meta Meta Model, Meta Model, User Model, and information. Adex does not support the fourth layer of instantiation as it typically exists in application databases and not in a model repository. The MOF model includes specification of methods on objects in the Meta Model, but Adex's Meta Meta Model does not allow specification of the methods for Meta Objects in the Meta Model. On the other hand, Adex provides for loosely coupled methods in the form of scripts, which can be associated to Meta Objects and Meta Associations in the Meta Model. Compared to Packages in MOF, Adex provides a much richer set of constructs that enable model partitioning, versioning, configuration management, and security.

Adex has been used by a wide variety of information systems to create and populate domain specific meta models. It has been used in MasterCraft [5] to create and store models for component based information system development and role based process of development. Information system development roles, information system components, component interfaces and 'consumer-supplier' relationships are modeled using the model partitioning constructs provided by Adex. Adex has been used to specify and store models from the domain of programming languages and program analyses by DARPAN [10]. It has also been used by GUI modeling and generation, and performance modeling and simulation tools.

Though Adex has been used successfully in many demanding information systems, we feel that it can be improved further in a number of ways.

Adex's browser user-interface is essentially generic in nature (though some amount of customization is possible). We feel that meta-model developers should be allowed to design user interfaces to suit their specific requirements. We have plans to implement a programmable GUI feature in our future releases by adapting GUI modeling ideas implemented in MasterCraft [5].

An information system repository should also support modeling of processes that coordinate various system-building activities. We have plans to include process-modeling support as a standard offering in Adex.

We feel that the 'Meta Modeling' framework can be further extended to make levels of instantiation seamless and unlimited [11]. For instance, support to capture the application data as a fourth layer of instantiation, as an instance of the user model, would enable Adex to facilitate rapid prototyping of the modeled information system.

7. ACKNOWLEDGEMENTS

We are grateful to Uday Ramteerthkar for the very first version of Adex, to Vipul Shah for the Script Language, to Girish Kulkarni for the Rulebase Engine, and to all the past and present members of the Adex team for their contribution in terms of ideas and implementation efforts. We thank K. V. Nori and Mathai Joseph for constant guidance and support. Many thanks to the reviewers for their constructive criticism towards making this paper more readable. Last but not the least, we would like to thank the many users of Adex for validating the ideas and providing valuable feedback.

8. REFERENCES

[1] P. P. S. Chen, Entity-Relationship Approach to Systems Analysis and design, North Holland, 1980.

[2] Unified Modeling Language V1.1, Object Management Group, November 1997.

[3] Rational Rose V4.0 System Documentation, Rational Software Corp.

[4] System Architect 2000 System Documentation, Popkin Software Corp.

[5] MasterCraft – Component-based Development Environment' Technical Documents, Tata Research Development and Design Center, 1997-8.

[6] XML-XMI 1.0 specification by OMG.

[7] CASE Data Interchange Format (EIA/CDIF) by EIA.

[8] R. CONRADI and B. WESTFECHTEL. 1998. Version Models for Software Configuration Management. In ACM Computing Surveys, June 1998.

[9] MetaObject Facility, MOF 1.3 by OMG.

[10] 'DARPAN Technical Documents, Tata Research Development and Design Center, 1998.

[11] Metaclasses are First Class : the ObjVlisp Model, Pierre Cointe, OOPSLA '87 proceedings.