Repository Manager technology

by J. M. Sagawa

IBM's Repository Manager™ enables specifications involved in the program application development process to be managed. On the basis of the technology, the Repository Manager/MVS™ was developed as a product. The primary concepts and services of the technology are introduced, and specific aspects of the product and its operation are discussed. A discussion of what is involved in designing and implementing a tool is also included.

Repository Manager™ (RM) provides a system approach to managing specifications. In IBM's Systems Application Architecture™ (SAA™) strategy, Repository Manager is a system to support the development and execution of software engineering tools for application development,[1] computer and network system management, and other application families. It uses an extended three-schema approach to enable the specification, transformation, and execution of tool systems, while enforcing specified corporate standards.

In the first four sections of this paper, the primary concepts and services comprising Repository Manager technology in Repository Manager/MVS™ Release 1 (RM/MVS) are introduced. In the next two sections, the concepts and facilities that are provided for tool development productivity are explained. Next, the relationship between the RM/MVS product and the Repository Manager portion of the SAA Common Programming Interface (repository CPI) is shown. Finally, some implementation-specific characteristics of the RM/MVS product are mentioned and current and future work is outlined.

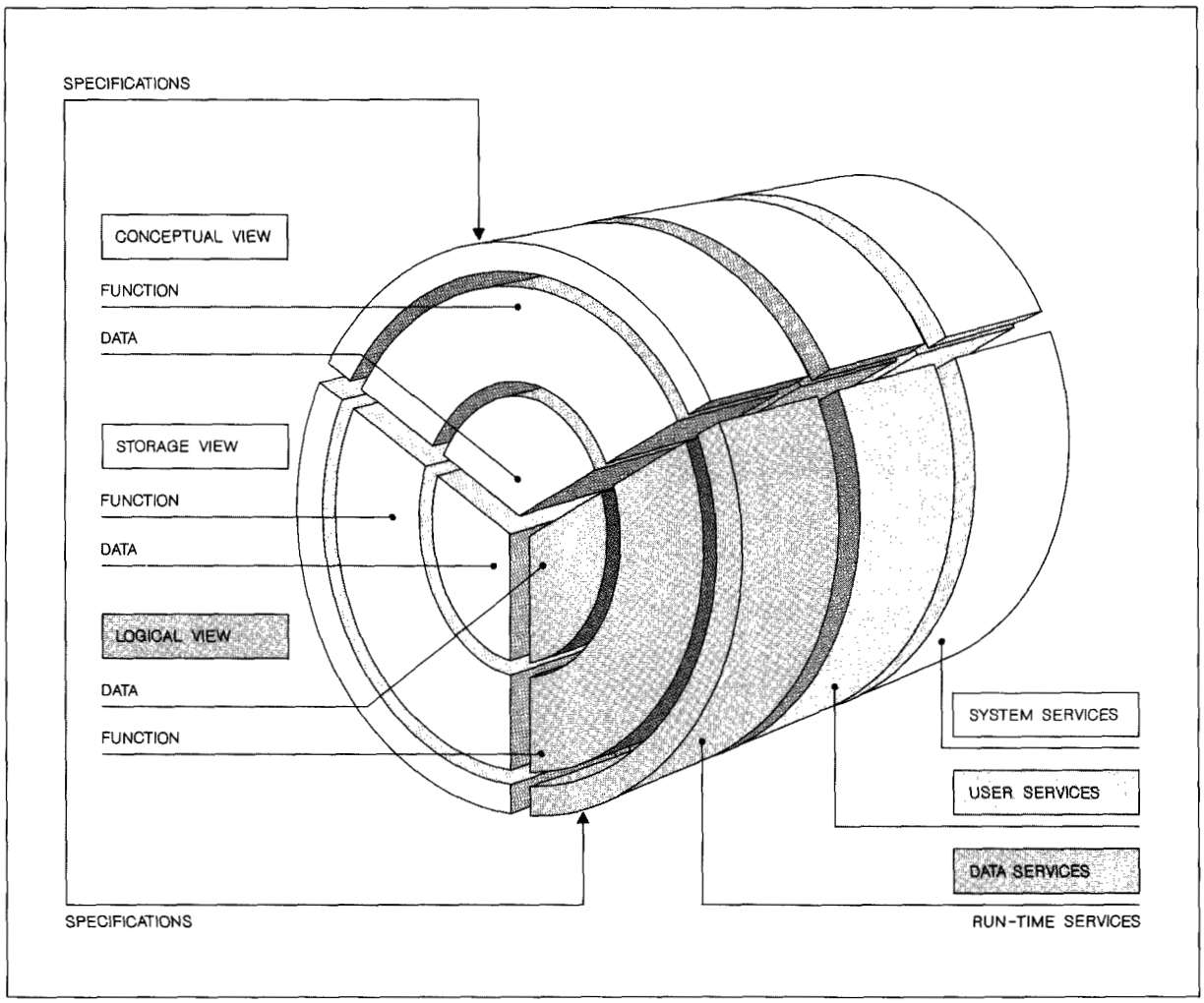
The concepts and facilities described in this paper are introduced in the manual, *Repository Manager/MVS: General Information*,[2] where the emphasis is on the RM/MVS product. The intent of this paper is to emphasize the basic technology of Repository Manager.

The architecture of Repository Manager

The Repository Manager architecture has two major domains: (1) specification and (2) run-time services. The specification domain encompasses the concepts supporting machine-readable specifications of tool structure and behavior, as well as end-user tools and program-callable functions for creating and maintaining these specifications. The role of run-time services is to enable execution of the specifications and to enforce global standards stated in the specifications.

Specification domain. RM manages specification by grouping assertions into related models of data and function. These models are further grouped into three categories called *views*:[3] (1) the conceptual view (CV), which is global, or common, across all tools and systems, (2) the storage view (SV), which models dependencies on system environments and services, and (3) the logical view (LV), which is specific to a tool. The views are depicted in Figure 1.

Figure 1 Architecture components



The conceptual view data model. Data are modeled in the conceptual view as entities, attributes, and relationships. These data representations are based on the work of Peter Chen.[4,5] Data value constraints, called *integrity policies*, are included in the model. Entities and relationships can be grouped together and modeled as *entity aggregations*. Entity aggregations are similar to the IBM Database/Data Communications (DB/DC) Data Dictionary "structure"[6] and the aggregation concepts of John and Diane Smith[7] and Dennis McLeod.[8,9] More abstract groups can be modeled as "objects." RM objects are similar to objects of OOPS (object-oriented programming system)[2,10] but with some important differences. RM objects can be abstractions for managing data in entity-relationship (ER) form and in files or other external forms. RM objects are managed with composite data locks, which persist across system restarts, whereas typical OOPS systems do not have such multiuser locks. The RM-managed data that are locked can reside in multiple database management system (DBMS) table rows in multiple tables.

All entities, relationships, entity aggregations, and objects are classified by "type," and are known to RM as instances of a specific type.

The conceptual view function model. Function is modeled in the conceptual view as policies on entities and relationships, and as methods for objects.

Four types of specifications, called *policies*, can be specified on constructs in the conceptual view. Integrity policies, introduced in the previous subsection, are part of the data model, whereas the other three types of policies are part of the function model. Security policies are rules for authorized access to entities, attributes, and relationships. Trigger policies specify execution of processes on the basis of the states of entity attributes and relationships. Derivation policies specify algorithms for creating entity attribute values. Policies are written as expressions in the IBM procedures language (REXX). Conceptual view policies are specified for enforcement when data are read from the repository or written to it.

The logical view should be regarded as a database of tool functional specifications.

In the function model there are object-type dependent operators called *methods*. The role of the method is to encapsulate the object. The name, parameters, and description of a method are globally specified. However, the detailed semantics of a method are determined by the object type that includes it.

The storage view data model. Data are modeled in the storage view as constructs that are dependent on the underlying system. For example, these constructs differ slightly between DATABASE 2™ (DB2™) in the Multiple Virtual Storage (MVS) operating system and Structured Query Language/Data System (SQL/DS) in the virtual machine (VM) operating system. They are concerned with tables and columns for storing instances of entities, attributes, and relationships. Performance-oriented constructs are also modeled, as for example, indexes on combinations of table columns.

The dependencies between the CV constructs and SV constructs are known and managed by RM. An example might be instances of the entity type PROGRAM stored in the DB2 table, Table Y.

The storage view function model. Function also is modeled in the storage view as constructs that are dependent on the underlying system. For example, transactions making up the implementation of a tool are functions that would be modeled in the storage view.[11]

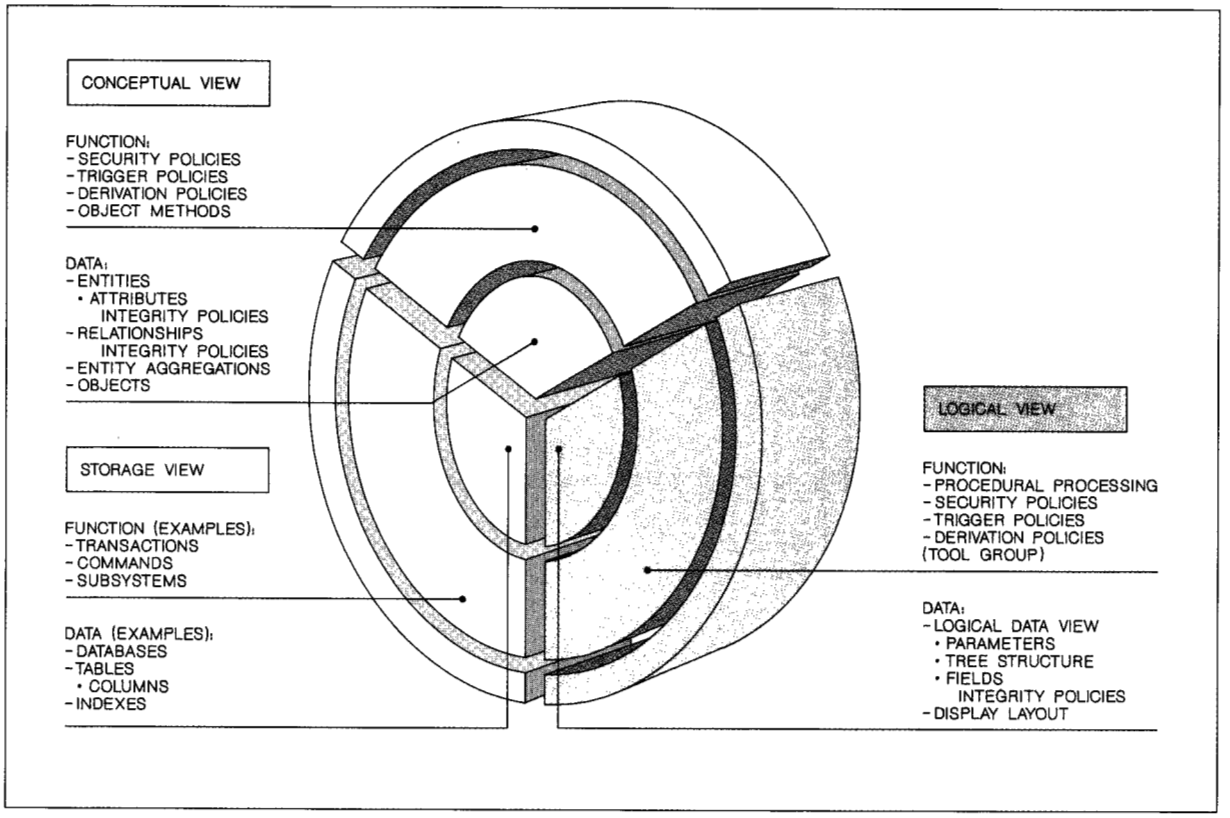
The logical view RM function. The logical view should be regarded as a database of tool functional specifications, and in the logical view, the basic construct is the RM function. The RM function is a package of specification, comprising the tool-specific data model and function model.

The logical view data model. The RM function data model is specified as a logical data view with its dependencies on the conceptual view, panels, and processing. The logical data view is composed of logical records, called *templates*, which are structures of fields. Template fields are used for parameters, views of entity attributes, views of Dialog Manager panel fields, and tool local storage. Templates are combined in hierarchic structures to view relationships, providing a general and powerful form of name "scoping." Integrity policies may be specified on a template field.

The logical view function model. Function is modeled in the logical view by RM function policies, which are nonprocedural, and by RM function procedural logic, which may be coded in a traditional programming language, such as C or COBOL. Security policies specify the authority by which a user can call the RM function or use a template field. Trigger policies specify template-field value-dependent conditions for automatically calling other RM functions. Derivation policies are algorithms for creating template-field values. Just as in the conceptual view, policies in the logical view are written as expressions in REXX. They are specified for enforcement at RM function initiation or termination, reading or writing RM-managed data, or reading from or writing to a display panel.

The tool group. The logical view includes the tool group construct. It is a packaging concept that allows specifications in the conceptual view, storage view, and logical view to be aggregated into a single group. Therefore, the tool group is the construct representing a complete tool system. The tool group is used to establish the scope of names during specification and to facilitate export and import of all the specifications for a tool. By utilizing the aggregation concept, the RM functions that support tool groups provide a methodology for tool installation.

Figure 2 RM function and data model



The elements making up the data and function models in the specification domain are depicted in Figure 2.

RM provides program-callable functions and interactive tools to perform query, update, and reporting on all specifications in the conceptual view, logical view, and storage view.

Run-time services domain. Run-time services are invoked by tools for access to RM-managed data (data services), dialog management (user services), and system facilities (system services).[2,12] As shown in Figure 1, run-time services can be regarded as being vertical slices through specifications. RM run-time services executes and enforces the data and function specifications in the conceptual view, storage view, and logical view.

Run-time services can be implemented in a number of ways, ranging from fully interpretive to fully compiled. The current implementation in RM/MVS is a semicompilation approach, where some functions are in executable form and other units require tool procedures to call run-time services.

Data services. Data services supports actions such as the reading and writing of entities, entity attributes, and relationships through the tool logical data view. Also provided are built-in functions that support the reading and writing of entities and relationships through template trees or groups of template trees, with a single call. Data services supports locking of entity aggregations on a long-term basis on behalf of a user, where the lock is persistent across system restarts. Entity aggregations can also be exported and imported between instances of RM. Commitment and restoration of RM-managed data are supported by internal use of SQL commit and rollback mechanisms. Data services enforces policies specified in the conceptual view and logical view for reading and writing RM-managed data. Enforcement includes checking the authority of users and tools to access RM-managed data, as well as executing derivation,

trigger, and integrity policies on the basis of the values of entity attributes and relationships.

User services. User services supports automatic mapping from template fields to Dialog Manager panel fields. Automatic mapping greatly improves the productivity for development of tools that use the IBM Dialog Manager (Interactive System Productivity Facility, or ISPF, on System/370). It also provides message management and diagnostic log services, with substitution of RM variables. User services enforces security, integrity, derivation, and trigger policies specified in the logical view for Dialog Manager I/O operations.

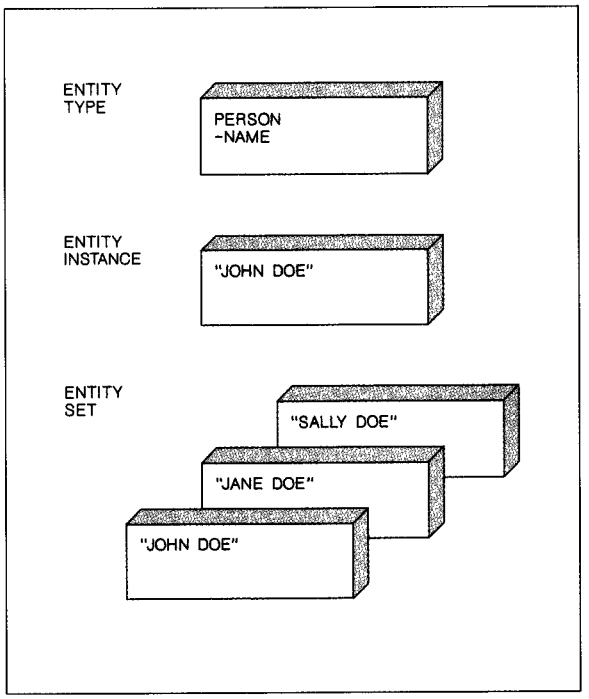
System services. System services provides such services as the open and close of RM function, call of integrated RM function, method RM function, and built-in RM function. It also supports dynamic binding of templates to entity sets and relationship sets, tracing and timing services, query of system-specific information such as the version and release numbers of the RM system, and system-specific support for services that may not be available in a particular tool development language. An example is the service that allows a REXX program to call a third-generation language program, passing parameters to it. System services enforces policies specified in the logical view for the invocation of RM functions of all types. This includes checking the authority of users and tools to execute an RM function, as well as executing derivation, trigger, and integrity policies on parameters.

Entity-relationship concepts in RM

Entities are representations of persons, places, things, events, and concepts in general.[4] Entities are often nouns. Entity attributes are representations of properties of entities and are often adjectives. Relationships are associations of entities and are often verbs or role descriptions. In RM, relationships can also be associations of relationships. Such relationships are often gerunds. In RM, entities and relationships may be categorized into higher-level groups, called *entity aggregations.*[7,8] These aggregations often are more abstract nouns or role descriptions. Persistent locks can be applied on those entities which are instances of entity aggregations. Extended composite object management is provided by utilizing entity aggregations and is explained in the next major section of this paper.

In RM, generalized constraints can be specified for entities, entity attributes, and relationships. These

Figure 3 Entity examples

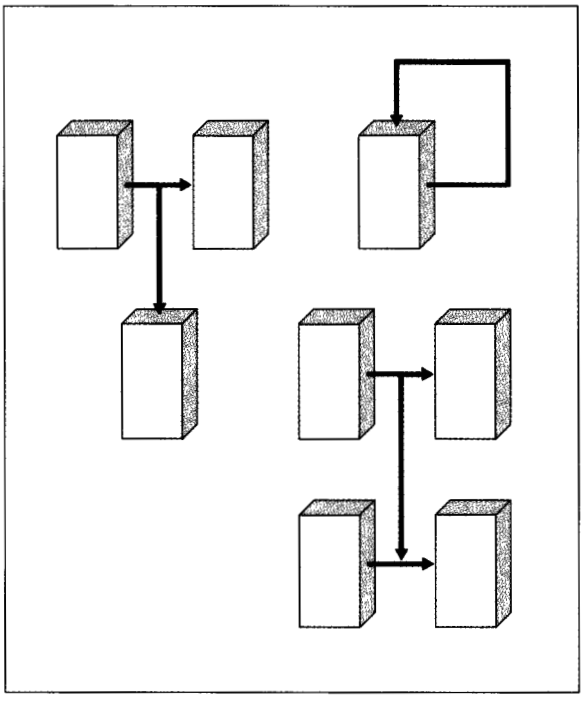


constraints are the RM policies, which were introduced earlier.

Entities and their attributes. Entities are instances of an entity type. The entity type defines characteristics common to all of the instances, including their attributes and policies. Groups of instances of an entity type are called an *entity set.* The set may be all instances of the type, or may be *ad hoc* groupings based on selection clauses or scoping specified in tool-specific logical data views. In the example in Figure 3, the entity set is those PERSONs whose last name is "Doe." Logical data views are explained later in the section on the RM function.

An entity type has a name and a description. It has a set of attributes, where one attribute, the "entity key," uniquely identifies the entity instance. RM supports "nonkeyed" entities, where data services automatically generates a surrogate key for each entity instance. It can be specified that the entity instances must be locked before they can be written. Trigger policies can be specified for the entity type, which cause RM functions to be scheduled for execution. The triggering conditions are checked at the time the entity is read or written.[13] These RM functions are

Figure 4 Relationships



executed when the tool issues an RM commit request, and RM restore processing will discard them. Security policies can be specified for read or write authority to the entities and attributes.

An entity attribute type has a name and description. It has default characteristics, such as data type and maximum length, which are used by generators of reports, panels, and tools, but are not necessarily the format of data stored in the RM-managed data store. The DBMS data format is specified in the storage view. Security, integrity, and derivation policies may be specified for entity attributes.

Relationships. In RM, relationships are "binary and directed." A binary relationship associates exactly two things, which can be entities or relationships. These relationships are depicted in Figure 4, where the boxes are entities, and the arrows are relationships. "Directed" means that each direction is given a name. Queries can be made using just the name in the right direction, and RM determines the characteristics of the relationship source and target. Relationship integrity is dynamically maintained: The existence of the source and target instances are automatically verified at relationship instantiation, and relationship instances are automatically deleted when their source or target instances are deleted.

A relationship has a description and two type names, one for each direction. One direction is called the *primary*, the other the *inverse*. No preference is given to one direction over the other. These terms are used to provide concreteness to the specification of the "source" and "target" of the primary direction (which are respectively the "target" and "source" of the inverse direction). The source and target may be entities or relationships. There are four kinds of semantic constraints on instances of the relationship type: cardinality, mandatory, controlling, and ordered set.

Cardinality—Cardinality semantics specifies that sources and targets can be related many-to-many, one-to-one, one-to-many, and many-to-one. For example, one-to-many allows one source instance to be related to multiple target instances via the subject relationship type, whereas any one of those targets can only be related to a single source instance. Specification of cardinality for one direction implies that the inverse direction takes on the inverse cardinality. This means that the inverse of one-to-many is many-to-one.

Mandatory—The mandatory semantic means that when a target instance is created, the relationship instance must also be created. This rule is enforced at RM commit time. Each direction can independently be mandatory for its target.

Controlling—The controlling semantic means that when the relationship instance is deleted, the target instance is also deleted. These automatic deletions are performed recursively, when relationship integrity is being maintained. Each direction can independently be controlling for its target.

Ordered set—The ordered set semantic means that the tools or users that instantiate the relationships can control the order in which instances are delivered at later read requests. Either the primary or inverse direction can be an ordered set, but they cannot both be ordered sets. The data services ER data manipulation language (DML) of RM allows the tool to manipulate the relationship order through use of the commands ADD BEFORE and ADD AFTER. Multiple relationship types may be defined with a common target, where some or all of the relationship types are independently ordered.

Security and integrity policies can be defined for instances of the relationship type. Security policies are enforced at read or write, where write is relationship instantiation or deletion. Relationships are not updated. Integrity policies are enforced at instantiation.

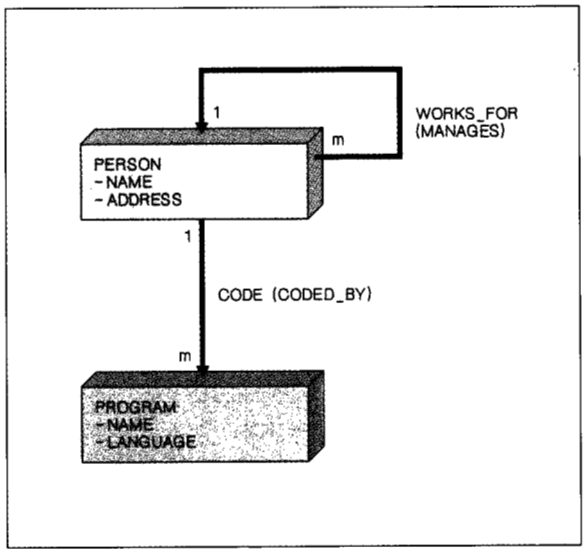
In RM, relationships do not have data attributes. Although there are cases where data attributes on relationships are useful—and we are doing work in this area—our focus has been on semantic control, where this is not vital. The function of relationship data attributes is modeled by dependent entities whose owner's source is the subject relationship. Dependent entities are explained in the next subsection.

The modeling of events or other concepts which require the association of more than two entities, commonly called *n-ary relationships*, is done by relationships between relationships, or by modeling the association itself by an entity. The reduction of all interdependencies to binary relationships allows specification of semantics at a fine level of granularity.

In the example shown in Figure 5, the entity PERSON is related to the entity PERSON by the relationship WORKS_FOR. The inverse of WORKS_FOR is MANAGES. One PERSON WORKS_FOR, at most, one PERSON, but one PERSON MANAGES potentially many PERSONs. The entity PERSON is also related to the entity PROGRAM by the relationship CODES. The inverse of CODES is CODED_BY. One PERSON CODES potentially many PROGRAMs, and one PROGRAM is CODED_BY at most one PERSON.

Dependent entities. Entities that occur in natural hierarchies can be designated as *dependent* entities. A dependent entity is similar to a *normal* entity but with the following differences. It is specified to be dependent on one, and only one, relationship type, for which it is the target. That relationship is called the *owning relationship*. The relationship source is called the *owner*. A dependent entity may be part of a chain, where the source of the owning relationship is itself a dependent entity. The dependent entity can occur in the chain only once; that is, it cannot be its own owner. The "backward chain" must eventually terminate in an owner which is not a dependent entity. For a dependent entity, instances of its entity key need only be unique in the context of the owner keys. Thus, dependent entities are a hierarchic form of name scoping. Access to the dependent entity

Figure 5 Two example relationships



instance can only be done in the context of the owner keys. Selection clauses in templates can refer to owner keys, so queries can be composed to discover the owners of a dependent entity instance. The owning relationship automatically has semantics of mandatory and controlling. It can have cardinality semantics of either one-to-one or one-to-many but can be neither many-to-one nor many-to-many. It can also be an ordered set, but only with ordering on the target (the dependent entity).

In the example shown in Figure 6, a PROGRAM may CONTAIN internal SUB_PROCEDUREs. The names of the SUB_PROCEDUREs are only unique in the context of the PROGRAM; therefore, they are modeled as dependent on CONTAIN. The graphic notation "D" at the upper left corner of the box for SUB_PROCEDURE indicates that it is a dependent entity.

Entity aggregations. Entity aggregation is a grouping concept, allowing entities of different types to be dealt with as a unit. An entity aggregation is hierarchic, with a root entity and a set of relationships arranged as branches in a tree structure. The hierarchy may be of any depth and any width.

An entity aggregation type has a name and description. It has a specified root entity type and optional relationship types with their positions in the hierarchy. RM functions can be specified to be executed before or after the export, import, lock, or unlock of

Figure 6 A dependent entity

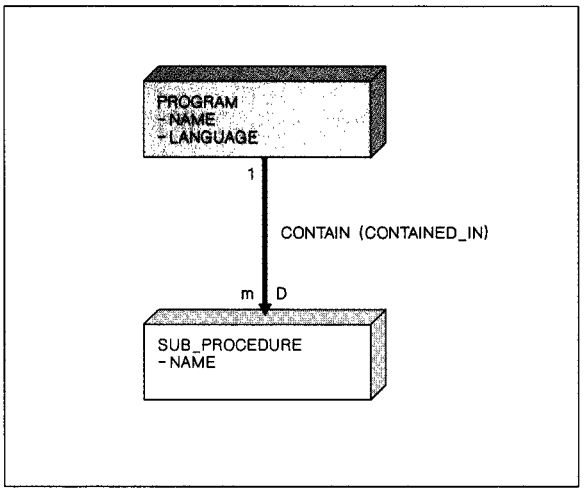
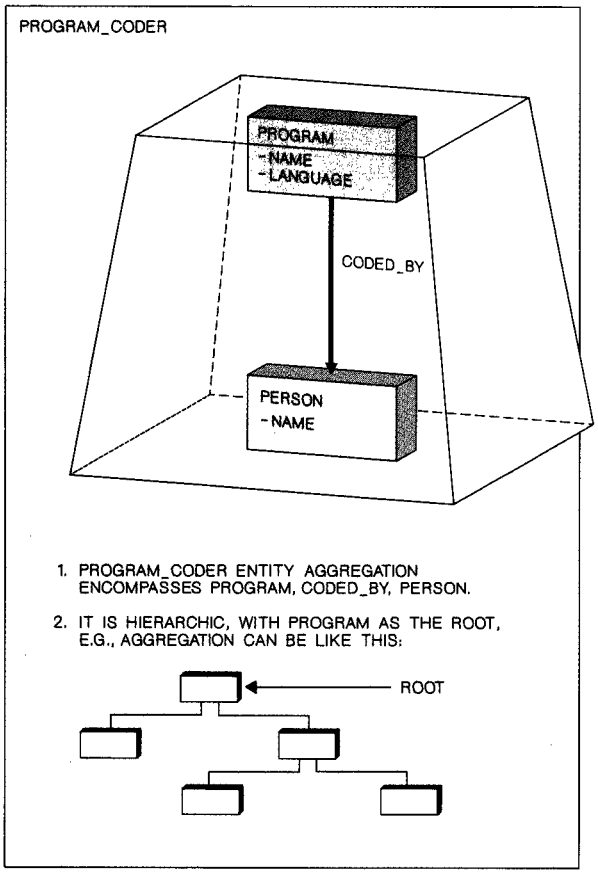


Figure 7 An entity aggregation



entity aggregations. An example could be: For the TOOLSPECIFICATION aggregation type, after successful import, execute the TOOL INSTALL NOTIFICATION function. Entity aggregations can overlap, in that more than one entity aggregation type can contain the same entity or relationship type.

Data services provides a callable RM function to do persistent lock management on entity instances which constitute an entity aggregation. Lock levels supported are no-update, update, add, and delete. No-update, also known as *stable read*, can concurrently be held by multiple users, but they must all be requesters of the no-update lock level. This is a nonexclusive lock, but it cannot be shared with holders of the higher-level locks (update, add, delete). The higher-level locks are exclusive; there cannot be any other user holding any lock on the instance. At lock request, data services checks that the requester has the authority to operate on the underlying entity instances at the requested level. These entity aggregation locks are enforced by data services when any user or tool attempts an operation on the underlying entities. Data services entity aggregation lock management cooperates with system services object method routing such that run-time reduction of object method call overhead is possible. See the next section for an explanation of object control.

In the example shown in Figure 7, PROGRAM_CODER is an entity aggregation type, which is used for showing which person codes a given program.

RM object concepts

My experience has been that it is relatively easy to explain RM entity-relationship (ER) concepts and establish some level of understanding with most audiences. It is less straightforward with RM object services. In this section, I will attempt to establish the motivation for RM object services, and at the same time introduce the concepts and facilities.

The object concept in RM provides for a common set of management facilities for RM-managed data that represent real-world items with a complex structure that often have information stored in nonhomogeneous media. For example, complex work products that are shared by multiple users are good candidates for management as RM objects. The object can be composite in nature. That is, it can span many database tables and media and can be very large. The object can have its data encapsulated by methods, which are object-type dependent functions.

Figure 8 Methods, inheritance, and bodies

RM OBJECT TYPE	METHOD	METHOD IMPLEMENTATION	BODY LOCATION
DOCUMENT	STORE RETRIEVE GENERATEDOCUMENT	SDOC GDOC GEND	ER, GML, AND OPTICAL (MIXED BODY)
DESIGNDOC SUPERTYPE: DOCUMENT	UNIQUE: GETMESSAGE TRACKREVIEWS GENERATEDOCUMENT INHERITED: STORE RETRIEVE	UNIQUE: GM37 TR7 GN17 INHERITED: SDOC GDOC	ER, GML, AND OPTICAL (MIXED BODY)

The object can be a subtype or a supertype of other objects, where the subtype inherits the methods of the supertype. Object method support is the basis for providing a single interface to multiple products performing the same function. The relationships of an object to entities, other relationships, and other objects are managed through use of data services relationship management. The object is the means by which RM provides consistent authorization support for composite data of mixed media. The object is the basis for managing versions of composite data. The object provides common attributes that tools can query and update to keep track of data movement between the host and the programmer workstations.

Composite data. If the data of an object are totally in entities and relationships, the object is said to have an "internal body." If its data are totally in some form not in entities and relationships, it is said to have an "external body." An example is a document composed of BookMaster source files, in a partitioned data set library. A mixture is said to have a "mixed body." See Figure 8 for examples of mixed media object data bodies.

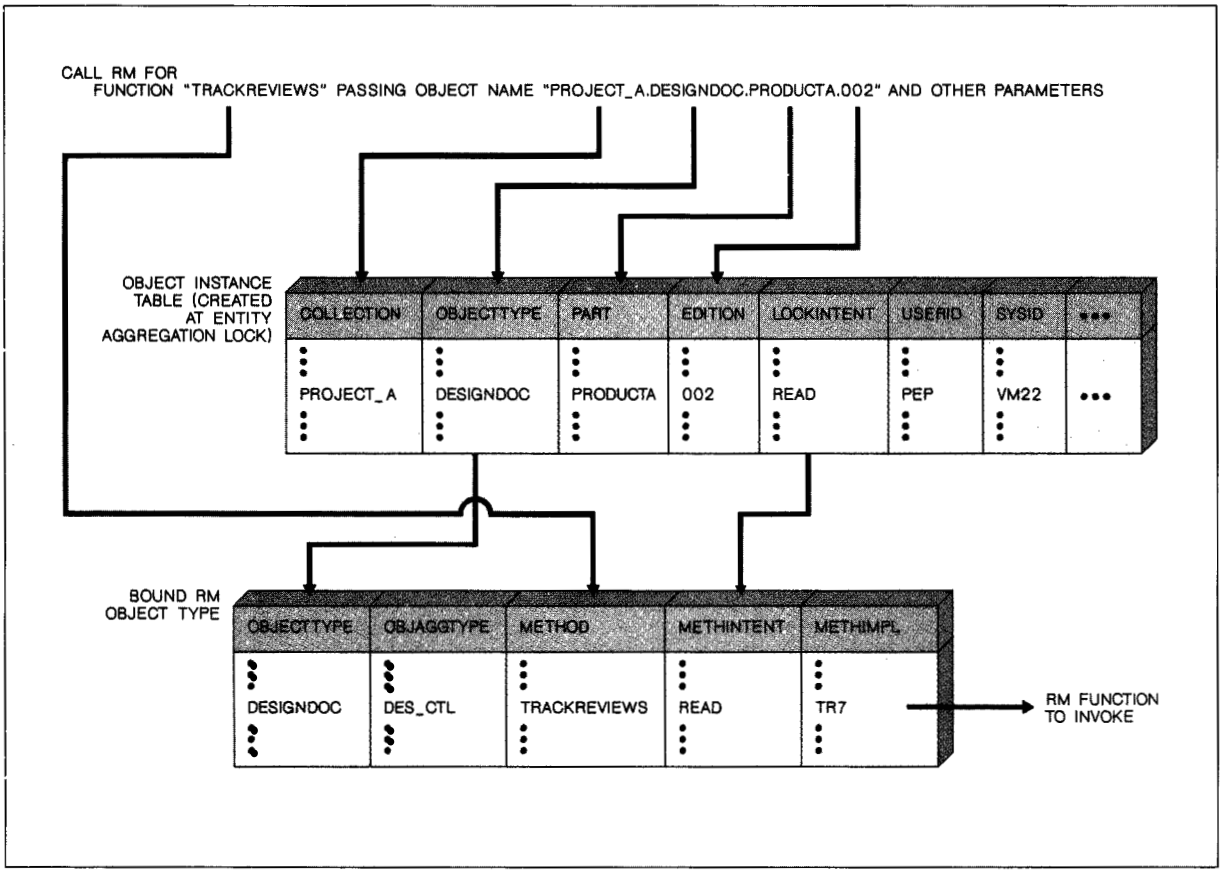
In all cases, the control information about an object is in entities and relationships. The conceptual view for the object control information is shown later in Figure 10.

Type-dependent encapsulation. The definition of an object (the "object type") can include a list of names of "object methods." An object method is an RM function; therefore, it has a globally unique name, description, and input and output parameters with policies. Every RM function which is an object method has at least the object name as an input parameter.[14] Regardless of which object type includes an object method, that method always has the same set of parameters. A method can have a different implementation for each object type where it is used. The method semantics are dependent on the method implementations (MIs), but, to be most useful, the MIs for a particular method should provide similar semantics. As an (admittedly extreme) example, it would be confusing for one implementation of the create method to do instantiation and another to erase files. See Figure 9 for an example of scheduling an MI for a method call. The object can be a subtype or a supertype of other objects, where the subtype inherits the methods of the supertype. See relationship SUPERTYPE in Figure 10. Also see Figure 8 for an example of method inheritance. Object support is the basis for providing a single interface to multiple products performing the same function.

Object relationships. The object instance is represented by the instance of the object edition entity type. The relationships of an object to entities, relationships, and other objects are defined as relationships from the object edition (Figure 10). These relationships are instantiated through use of the normal data services ER DML.

Object access authorization. RM object services provides consistent authorization support for composite data. Every object type must have an entity aggregation type defined for it to be used for locking and method access purposes. Such an entity aggregation always has object edition as the root entity. Object lock-level requirements are defined for each object method. The levels are those lock states supported

Figure 9 Method implementation routing



for entity aggregation locks. If the object is locked at the level required for the method (or at a higher level) for the requesting user, the method may be called. Otherwise the object method call is rejected.

Version control. The object provides the base for common version control services (Figure 10). Object instances with the same collection, type, and part name can have different object edition key values, thus being different "versions" of the "same thing." Object editions, and collections of object editions, can be made into different versions relative to each other by use of the relationships COLLECTION BASED ON COLLECTION and EDITION BASED ON EDITION. This is sometimes referred to as "heritage versioning," since if the tools properly maintain instances of the ... BASED ON ... relationships, it is possible to query where the object "came from."

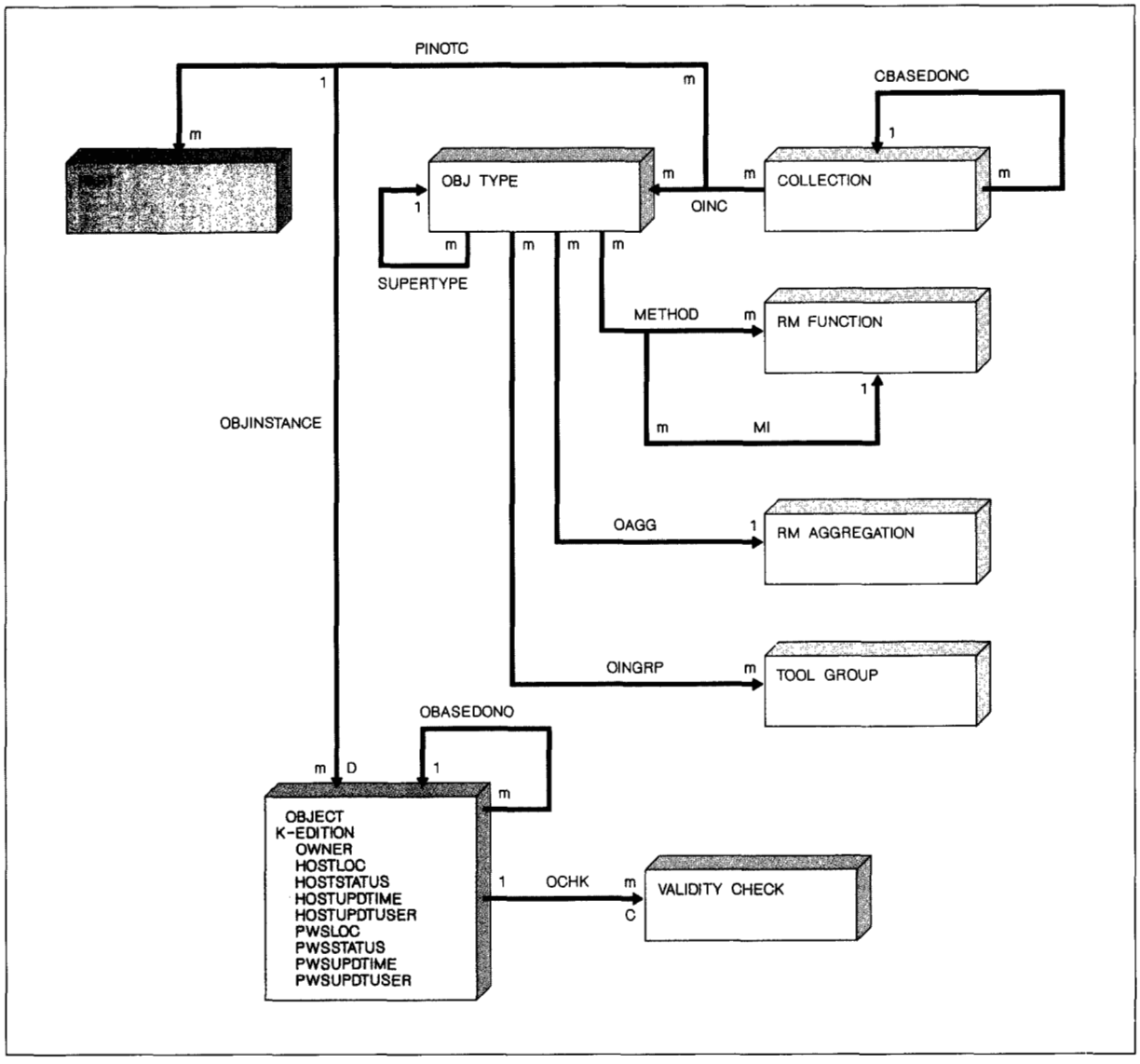
Object data location. The object is the basis for data movement between the host repository and the host-connected workstation. Method implementations do the actual data movement, but the common attributes in the object edition can be used to hold information such as identifying which workstation and which file at that workstation holds a copy of the object data body.

The RM function

It is important to understand the RM function construct because it is the basic unit of tool functional specification. It is similar to the Ada® package specification, in that it is intended to be used as an interface description, which describes the outside of a black box.[10,15] RM functions are "understood" by the RM system. Tools are made up of a network of RM functions.

Four types of RM function. An RM function has a name and description. It is classified into one of the following four types.

Figure 10 Object control, conceptual view (CV)



Open/close. An open/close RM function is just a package of templates. The RM function is opened; the templates are used to access RM-managed data and services; and then the RM function is closed. The RM function has no procedural logic, but it can have any type of logical view policy. For example, security policies prevent unauthorized users and tools from opening an RM function.

Integrated. The integrated RM function, in contrast, does have procedural logic. It need not be associated with any object type, and no object-type dependent implementations can be specified. It can be called by tools (other RM functions) and policies, with parameters passed through its parameter template fields. It can have any type of logical view policy. For example, any violation of integrity policies on input parameters will cause the RM function call request to be rejected. Its procedural logic is executed at the time it is called and can be coded in third-generation language or REXX programs. It accesses ER data through its own tailored view, which is specified by

templates. See the following subsection on the logical data view for more detail about templates. The integrated RM function can have optional dialog display panels, with automatic mapping between panel I/O fields and template fields.

Method. The method RM function can be called, but it has no procedural logic. It is the interface from a tool to the MI (method implementation). Its parameter template must have at least the fields for the object name. It can have any type of logical view policy. This type of RM function can be called, with object lock control enforcement and with routing to the object-type dependent implementation (MI). The example of method-to-MI routing is shown in Figure 9.

Integrated and method RM functions can be called by conceptual view policies, logical view policies, procedural logic, and end-user command.

Method implementation. The MI RM function has procedural logic, but it cannot be called directly by tools. It can only be invoked via method-MI routing. It can have any type of logical view policy. For example, an input parameter could be synthesized by a derivation policy which includes the calling of an integrated RM function that solicits additional input from the end user. In most respects, the MI RM function is the same as an integrated RM function, except that its parameter template must have fields for the object name.

RM function logical data view. The logical data view in an RM function is comprised of templates. The template is a group of fields; it is a logical record. A template field is a view of one or more of the following: entity attribute, parameter, interactive panel field, and local storage. The fields in a template may view a subset of the attributes of an entity. The template-field data type, precision, and length can differ from the ER data stored in the DBMS. If they differ, the field value is converted at run time.

Templates can be arranged in trees, where a template tree is a hierarchic view of the entity-relationship network of the conceptual view. A branch of the template tree maps to a relationship in the conceptual view. A template mapping to a dependent entity has a field for each owner's key attribute.

Retrieval selection clauses can be specified on a template. For the example in Figure 3, a template for reading PERSONs based on last name would have a selection clause similar to PERSON.LASTNM = LLAST, where LLAST is assigned the value of the last name, "Doe."

A template can be arrayed so that a large set of entity and relationship instances can be read or written in a single operation. A template array is a table in main storage; it can have any number of rows, and retrievals can be under the control of a selection clause.

A field in a template can be synthesized by derivation policies, so the fields in a template can be a superset of the attributes of an entity.

Logical view policies are subordinate to conceptual view policies, in that conceptual view policies are always enforced before ER data are changed.

ER data reads and writes can be issued on a template, or the data access can be done in fewer calls by using RM built-in functions. Built-in functions operate on template trees or groups of trees in a single operation. The RM logical data view provides a simple but powerful form of data access. The run-time syntax is very simple, but the semantics in the specification are comprehensive. It provides a form of hierarchic name scoping on data which can be overlapping sets. It supports a means of nonprocedural processing that is driven by the occurrence of events and changes in the data state. Defined events are ER read, ER write, RM function initiation, RM function termination, display read, and display write.

Display specification. Interactive display panel specifications are held in RM as part of the RM function specification. For example, included are logical field display coordinates and default highlight control. The display and the logical data view are related, but are separate components of the RM function specification.[16]

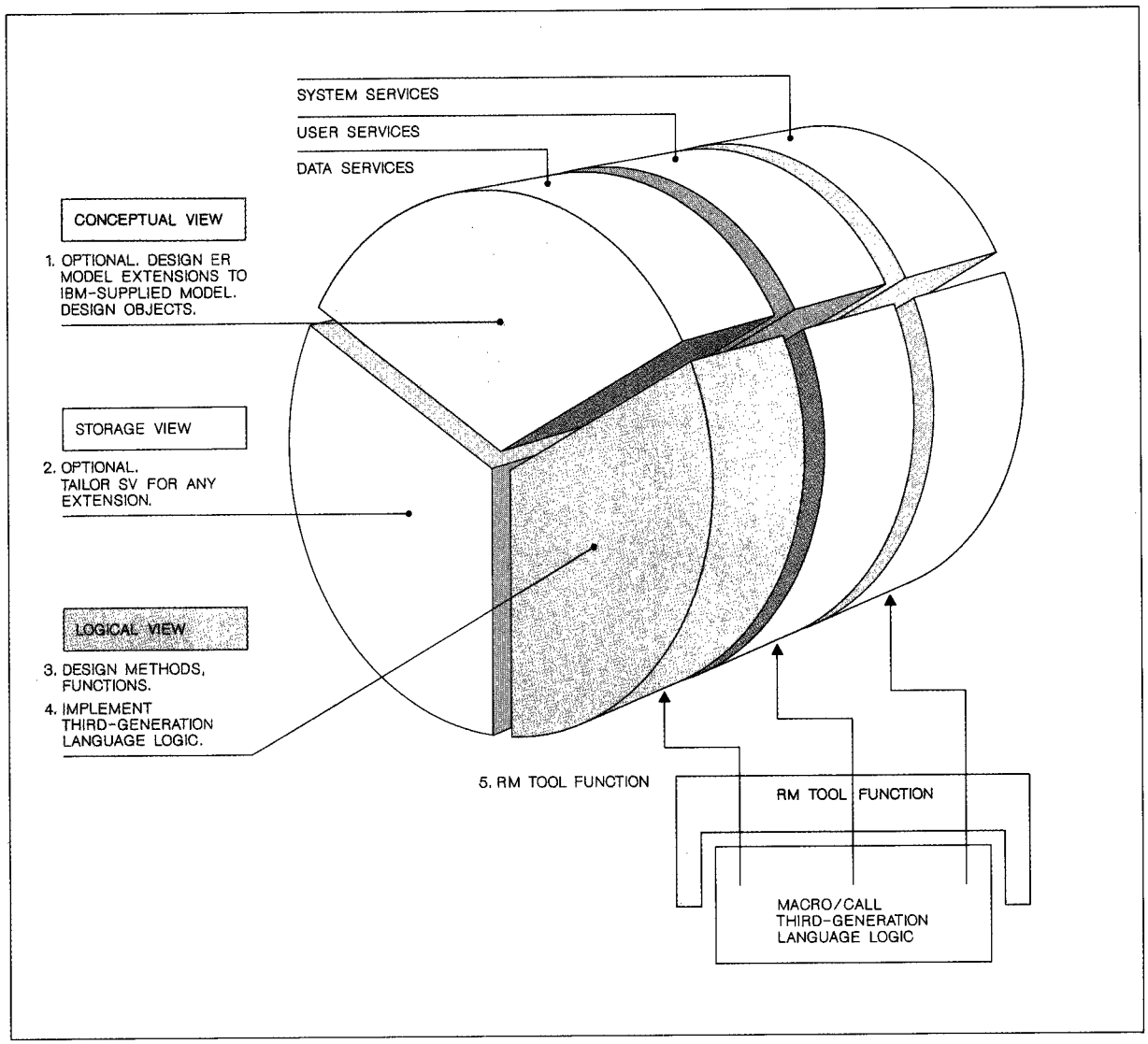
RM tool development and execution method

The intent of this section is to give an intuitive understanding of what is involved in designing and implementing a tool, and to provide an integrated view of the concepts and facilities described so far.

A tool designer will usually perform the following steps (illustrated in Figure 11):

1. Define a conceptual view—This step is optional, in that the existing conceptual view may be com-

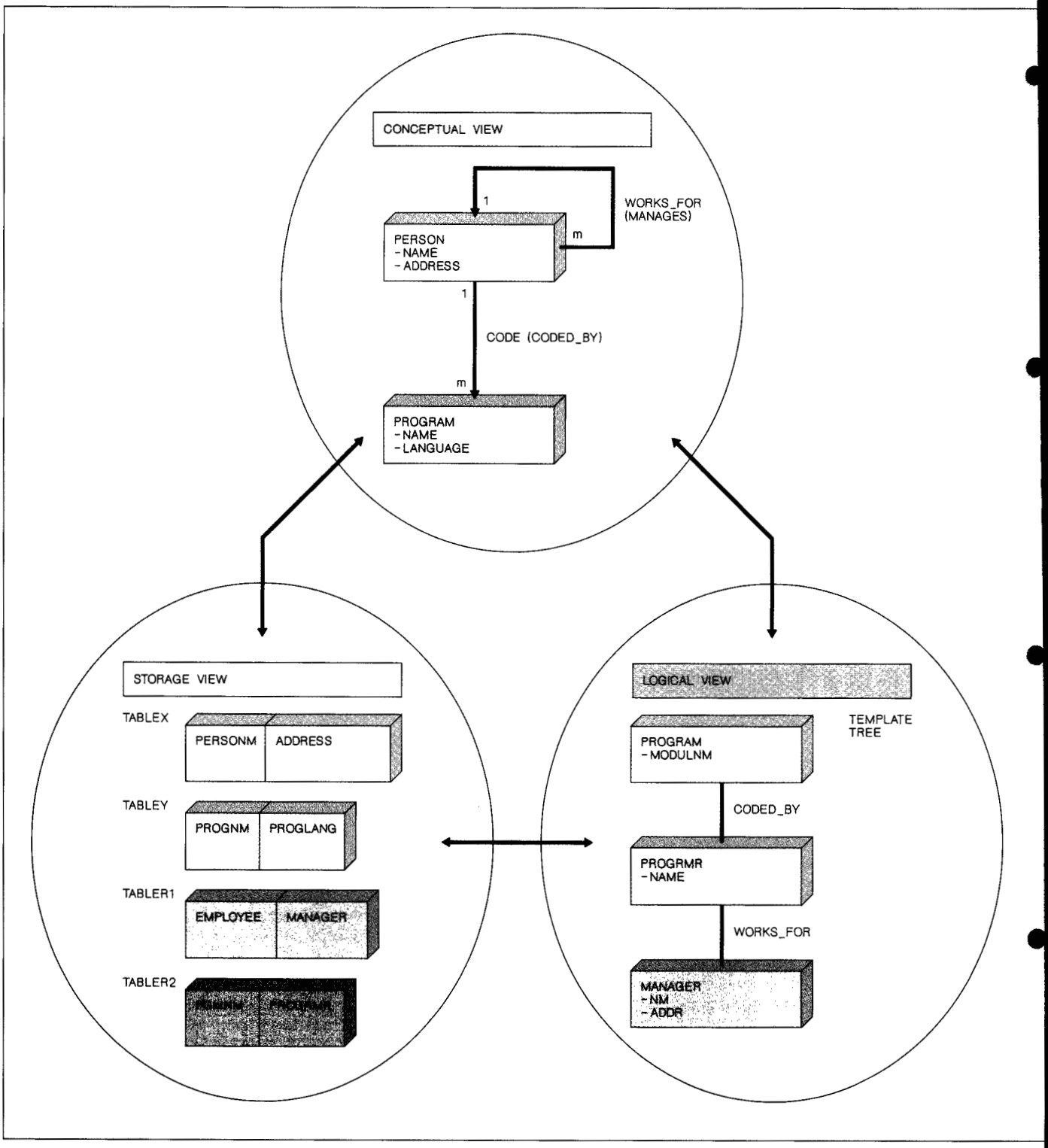
Figure 11 Tool development steps



plete enough for the purposes of the new tool. This step is "data modeling," that is, design of the ER model. Usually, design consists of extensions to the IBM-supplied ER model. New object types and their object methods can be defined.

2. Define tailored storage view for the conceptual view—This step is optional, since it obviously is only necessary if the conceptual view was changed. Also, a default storage view will be provided by RM. This default is suitable for prototyping the tool, and the conceptual view changes.
3. Define one or more RM functions in the logical view—A mapping of the logical view to the storage view may be done, depending on the RM implementation and the user's needs. The RM function can be a simple data view (open/close), an integrated RM function, a method, or a method implementation.
4. For each RM function, write one or more programs—RM provides productivity facilities for the PL/AS, PL/X-86, PL/I, COBOL, C, and REXX programming languages,[17] including generation of source code, such as the procedure parameter list,

Figure 12 CV, SV, LV mapping example



template declaration structures, return code constants, and invocation macros.

5. Execute the tool RM function—The third-generation language logic accesses RM-managed data and services via the logical data view in its RM function. These accesses are under the control of the logical view policies in the RM function and the global conceptual view policies. The policies and data semantics are enforced by RM run-time services.

Repository Manager in relational DBMS

RM uses the services of the IBM SAA DBMS, through its implementations in DB2 and SQL/DS. Interactive dialogs are provided for database specification and for mapping the conceptual view to the storage view. SQL DDL (data description language) is generated from the RM-managed data, including automatic generation of useful indexes. Static SQL application source code is generated to improve performance of common execution paths. Dynamic SQL is used where appropriate. RM provides extensive instrumentation for system and tool diagnostics and tuning.

A simplified example illustrating the roles of the three parts of the specification domain is shown in Figure 12, with a DB2 storage view.

Repository Manager within SAA

The main concepts and facilities in the Repository Manager/MVS Version 1 Release 1 product are depicted in Figure 13. This is a superset of the SAA repository CPI, which is shown in Figure 14. Some specific non-CPI facilities available are reliability, availability, and serviceability (RAS) services for tracing and logging, interactive dialogs for specification maintenance and prototyping, and utility tools for product installation and customization.

SAA repository CPI. The functions that ultimately will be supported by the Repository Manager portion of the SAA CPI (repository CPI) are shown in Figure 14. The first level of the CPI only supports the syntax of ER data manipulation language (DML). The specification domain is an extended two-schema architecture, which includes the conceptual view and logical view, but not the storage view, which is specific to System/370.

The CPI conceptual view includes entity and relationship with integrity, derivation, and trigger policy types. Security policy capability is not included, pending definition of SAA security. The CPI includes aggregation, object with method, and supertype for method inheritance.

The CPI logical view includes RM function, with logical data view and integrity, derivation, and trigger policies. The RM function types of open/close, integrated, method, and method implementation are supported. Logical view security policies have been excluded. Tool group (to group definitions of objects, entities, relationships, aggregations, and RM functions) is included.

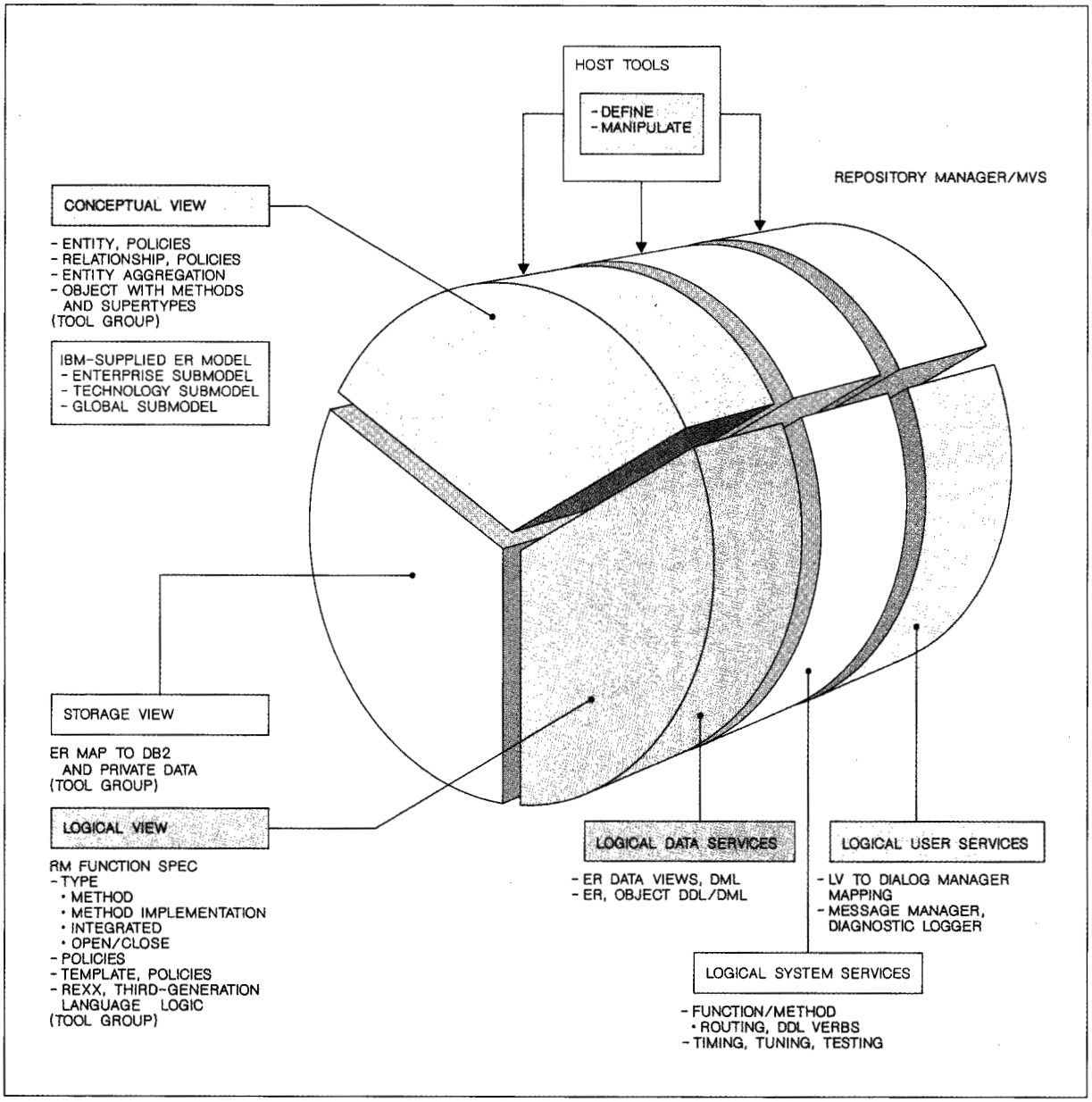
A Repository Manager data load facility is part of the RM product.

Run-time services in the repository CPI are data services and a subset of system services but not user services. Data services includes ER logical data view management and data access (DML), including built-in functions, ER and object specification (DDL), and object instance access (DML). System services included are the open/close RM function, call RM function with method call routing to object-type dependent implementations, and DDL verbs for all RM function types. Also included are bind, unbind, and system information query. Not included in the CPI are system services for timing and diagnostic tracing.

RM implementation

RM is implemented in RM means that important elements of RM/MVS are specified in RM-managed data via a conceptual view of RM itself. Examples of system elements modeled in RM-managed data are tool groups, entities,[18] relationships, entity aggregations, objects, RM functions, system control blocks, buffers and data areas, system commands, system return codes, and system messages. Some benefits observed for this approach have been: Interactive maintenance tools are quickly implemented and easily maintained; documentation is to a large extent

Figure 13 Repository Manager/MVS V1R1

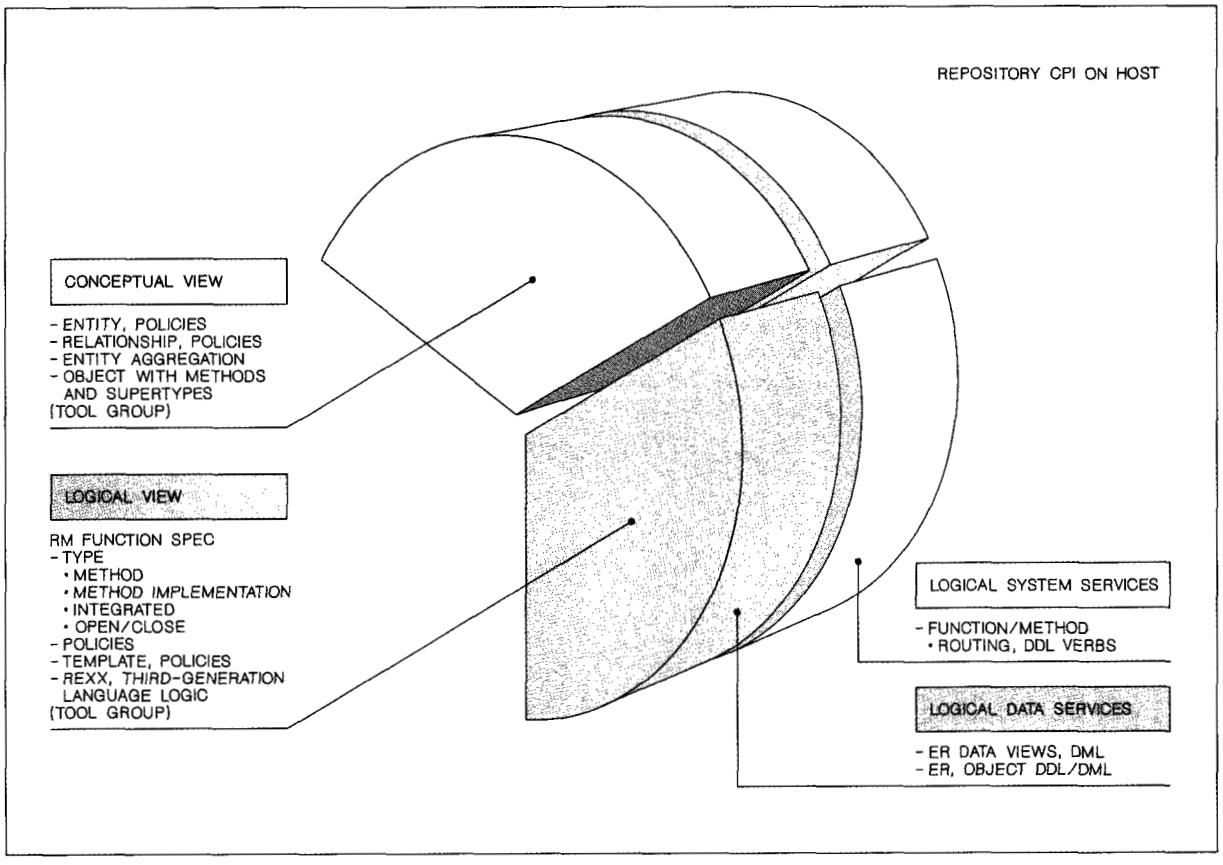


automatically generated; and source code for constants and data structure declarations are automatically generated.

Since RM function specifications are held in RM-managed data, it is straightforward to generate source code fragments in the language the tool developer chooses. Such support has been implemented for PL/AS, PL/X, PL/I, COBOL, and C.

A Repository Manager data load facility (RM loader) is part of the RM product. It populates the RM-managed data stores from files created by programs or users outside RM. An example of this is the Dic-

Figure 14 Repository CPI structure, strategy



tionary Model Transformer (DMT) product offering. DMT reads data from the IBM DB/DC Data Dictionary and writes it to a file, which can be read by the RM loader.

RM provides integrated support for the Query Management Facility (QMF™) for reports and queries on ER data.

Future directions

The repository CPI will be extended in functional scope as well as in support of other host and cooperative environments. We continue to work on extending the conceptual view data modeling constructs to be semantically richer, while not neglecting opportunities for fully utilizing the underlying operating systems and DBMS by extending the storage view architecture. The concepts of entity, entity aggregation, and object will continue to converge, with eventual integration with the general concepts of entity generalization.[7]

Further research needs to be done on structural aspects of the RM architecture, such as peer-connected Repository Managers and hierarchically connected Repository Managers, which are aware of each other and cooperate in solving the problems of distributed semantic management, with acceptable performance.

Summary

Repository Manager is a system for managing specifications. In the IBM SAA strategy, it is also a system to support the development and execution of software engineering tools for application development and other strategic requirements. It enables tools to be specified through an extended three-schema architecture which models data and function. It trans-

forms these specifications into systems of tools and executes the tools while enforcing corporate standards. Repository CPI is an additional component in the IBM Systems Application Architecture Common Programming Interface. Repository Manager/MVS is implemented using its own technology.

Acknowledgments

Significant and fundamental early contributions were made to the architecture by Claude Miller, Christopher Wood, Vern Watts, Carlos Goti, and Jerome Fox. In the application of Repository Manager technology to the IBM internal software engineering strategy, William Beregi and Gene Hoffnagle were key contributors. Peter Hein shared a vision of what is possible and provided an early demonstration of the value of RM with the MIRAGE system.[19] Very early on, Raymond Berman implemented the first production RM tool (OCTOPUS), and it is still in daily use.[20] The product and technology efforts may not have survived without the vision and unflagging support of IBM management, especially Donald Hyde, who started the research effort, Robert Tabory, who was instrumental in obtaining funding at critical times, and Norman Pass, who kept the effort alive and took it into the product development organization. William Hallahan and Claudia Gardner-Treiber were the key people with the experience and perspective on how to apply the advanced technology prototypes to the application development problems of IBM customers. Many more people have been collaborators and supporters over the years. My thanks to all these people.

Many suggestions by Fran Beason, Karen Roberts, and the patiently anonymous referees were crucial in making this paper as readable as it is. Any errors in fact or style are solely mine.

Cited references and notes

1. *Systems Application Architecture: AD/Cycle Concepts*, GC26-4531-0, IBM Corporation (1989); available through IBM branch offices.
2. *Repository Manager/MVS: General Information*, GC26-4608-0, IBM Corporation (1989); available through IBM branch offices.
3. The term *view* is somewhat limiting, in that some people might interpret it as meaning only data as opposed to both data and function, but it is used for primarily historical reasons.
4. P. P. S. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Transactions on Database Systems* **1,** No. 1, 9–36 (March 1976).
5. E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems* **4,** No. 4, 397–434 (December 1979).
6. *OS/VS DB/DC Data Dictionary Administration and Customization Guide*, SH20-9174, IBM Corporation (1979, 1984, 1986); available through IBM branch offices.
7. J. M. Smith and D. C. P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Transactions on Database Systems* **2,** No. 2, 105–133 (June 1977).
8. D. McLeod, "A Semantic Data Base Model and Its Associated Structured User Interface," *MIT/LCS/TR-214*, Massachusetts Institute of Technology, Cambridge, MA (August 1978).
9. M. Hammer and D. McLeod, *SDM: A Semantic Data Model*, USC TR 80-3, University of Southern California, Los Angeles (February 1980).
10. G. Booch, *Software Engineering with Ada*, The Benjamin/Cummings Publishing Co., Menlo Park, CA (1983).
11. This is not done in the RM Release 1 product, but the architecture enables it to be done in the future, for example, for performance or control reasons.
12. G. F. Hoffnagle and W. E. Beregi, "Automating the Software Development Process," *IBM Systems Journal* **24,** No. 2, 102–120 (1985). RM is described in this paper as "common tool services."
13. Policies for read are separately specified and enforced from policies for write.
14. Actually, it is four parameters for the four parts of the object name: collection, object type, part, and object edition.
15. *Common APSE Interface Set (CAIS), Proposed Military Standard, Version 1.3*, Report AD-A134825/9, Office of the Secretary of Defense, Ada Joint Program Office, Washington, DC (August 1984).
16. The display specification acts as a storage view for the interactive RM function. Perhaps we should recast the architecture to make that explicit.
17. PL/AS and PL/X-86 are IBM product development languages for the System/370 and the Personal System/2®.
18. J. M. Fox, J. C. Goti, C. R. Miller, and J. M. Sagawa, "Implementing a Self-Defining Entity/Relationship Model to Hold Conceptual View Information," *Proceedings of the Second International Conference on Entity-Relationship Approach to Information Modeling and Analysis*, ER Institute (October 1981), pp. 569–581.
19. MIRAGE (MInispec and Repository based Application GEnerator) was the advanced technology prototype for DevelopMate™. It was a fully functional tool for Yourdon-DeMarco-based requirements and application analysis and prototype generation.
20. OCTOPUS (Old Code TO Properly Understood Software) is an interactive tool to allow queries on the component structure of products with multiple versions under concurrent development. It allows unlimited bidirectional and recursive queries of a "bill of material," including macros, modules, and program symbol usage.

James M. Sagawa *IBM Programming Systems, Santa Teresa Laboratory, P.O. Box 49023, San Jose, California 95161-9023.*

Mr. Sagawa received a B.S. in electrical engineering from the California Institute of Technology in 1963. He joined IBM at the Burbank branch office where from 1963 to 1969 he was a systems

engineer specializing in engineering and scientific applications, operating systems, time sharing, and graphics. As a member of the CADAM® implementation team at the Lockheed California Company, he wrote the graphic attention handler and various performance prediction tools. From 1969 to 1971 he was on assignment as a consultant to IBM United Kingdom for marketing and implementing engineering graphics, scientific applications, and operating systems. In 1971 and 1972 he was on assignment at the IBM World Trade Manufacturing Industry Marketing Center in Munich, Germany, where he provided guidance to European country-level marketing for engineering and scientific applications. From 1972 to 1977 he was a member of the Information Management System development team in Palo Alto, California, and served as chief programmer for IMS/VS 1.0.1. In 1977 he helped start the Sundance advanced technology project, which evolved into the Repository Manager (RM). As architect and chief designer, he led the effort to apply RM (in its implementation as Common Tool Services) to the development and execution of IBM internal system development tools as the base for the Software Engineering Support Facility (SESF) architecture. Later, he led the effort to apply it to the development and execution of application development tools as the base for AD/Cycle™. Mr. Sagawa is a past member of the SESF Architecture Review Board and ADE System Design Council. He currently is a member of the Santa Teresa Laboratory Technical Review Council and the AD/Cycle System Architecture Board.

Reprint Order No. G321-5394.