Data Integration
Using
Virtual Repositories

By W. Scott Kerr

Department of Computer Science,
University of Toronto, Canada

September 1999

A thesis submitted in conformity with the requirements for the degree of Master of Science, Graduate
Department of Computer Science, University of Toronto.

Copyright © 1999 Scott Kerr
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-45950-0
Abstract

Online information is available from a variety of sources. These include databases, transaction systems, the World Wide Web, repositories, and file systems. Increasingly, specialized applications offer their internal data for export as well. Other applications derive secondary data from these sources. Each source offers its own methods for accessing data, its own stream formats, and an internal model of its data. A problem often faced by enterprises is to create integrated, browsable representations of data from these sources. This is normally done by duplicating source information in a database or persistent repository.

This thesis introduces active virtual repositories (AVR's) as a means of data integration for read-only browsing. Analogous to virtual databases, AVR's do not duplicate the source information persistently. Thus, they do not have to deal with the synchronization issues between the sources and copies of the data. Instead, they simulate repositories by allowing retrieval of entities and navigation across their properties. AVR's forward browser requests to the source applications, which already store data or produce it on demand. Metadata models then map the sources' data representations into a common format for integration. AVR's then respond to browser requests by integrating data from several sources into browsable formats like HTML, or XML.

AVR's encode metadata about source applications in object-oriented schemata. These schemata are represented in the XML Metadata Interchange (XMI) format. The schemata coordinate navigation between the different sources and present them as a single information service. A separate metamodel is created for each source to be integrated. This embodies: (1) the source's subject domain and data model, (2) the methods and formats the source offers for data access and (3) the dependencies between sources that derive data from each other. A global schema provides the unified view of the schemata that encapsulate each tool. If the AVR stores computationally expensive data, it propagates updates to derived data through event-condition-action rules taken from active databases.

A prototype implementation of an AVR was built to show that these techniques are applicable to software engineering. The prototype seamlessly integrated a diverse array of independently developed software visualization and analysis tools without unnecessary data duplication.
For Ivan Kalea
Acknowledgements

I would like to thank my supervisor John Mylopoulos, my co-supervisor Kostas Kontogiannis, University of Toronto faculty members Igor Jurissica, Eric Yu, Alberto Mendelzon and Rutgers professor Avigdor Gal. I would like to thank the staff of University of Toronto’s Computer Science department (especially Kathy and Belinda). I also appreciated the support of all the staff at the IBM Centre for Advanced Studies at IBM Toronto Lab. Stephen Perelgut, Derek Rayside, Homyoun Dayani-Fard, Joanna McGrenere, Chantal Buttery, Bill O’Farrell, Jacob Slonim, Gabby Silberman and Karen Bennet were of great assistance. In addition, I must thank the systems administrators over the years I worked there. Collaborating closely with Ivan Kalas for a year provided much of the inspiration for the framework described here. Special thanks is due to Pat Finnigan who introduced me to Derek Rayside. I’m also grateful that he told me that OTI would be a fun place to continue the research we started at UofT and CAS. All the faculty and students involved with IBM-CSER project (the Software Bookshelf project) at CAS need to be thanked as well especially Howard Johnson, Hausi Mueller, Morven Gentleman, Ric Holt, Gary Farmaner and Susan Sim whom I had most interaction with. The students at the Knowledge Representation lab at University of Toronto also provided me with much discussion to hone the ideas in my thesis. Thank you Brian Nixon, Joseph Makuch, Niloo Hodjati, Daniel Gross and Brian Chambers. In addition, I appreciated the input of Iannis Toulakas, Martin Stanley and Brian Kramer for insight concerning the Telos data model (used in the early prototypes of this thesis). Working with the staff at Object Technology International Inc. also provided many refinements of the original ideas developed at CAS. Thanks especially to Dave Thomas, Peter Tanner, Kevin McGuire, Ivan Moore, Jim des Rivieres, Tim Ellison, Nick Edgar, Jeff McAffer, Tod Creasey, DJ Houghton, Mike Keith, Erich Gamma of OTI and Martin Nally of IBM TJ Watson Lab. I must also thank all the OTI staff with generously providing me with infrastructure and administrative support during my many nights in Ottawa working on the thesis. The administrative staff were especially supportive. I also would like to thank all those who provided me with emotional support during this time: my parents, my partner Robyn and my friends.

Finally, to those who helped me but are not mentioned here, my sincerest apologies for the oversight, and thank you!
# Table Of Contents

CHAPTER I: INTRODUCTION ................................................................................................................. 6

1.1 PROBLEM DESCRIPTION ........................................................................................................... 6
    1.1.1 Integrating Data for Browsing ......................................................................................... 6
    1.1.2 Motivating Example ....................................................................................................... 7
    1.1.3 Software Repositories and IDE’s .................................................................................... 8
    1.1.4 Change Management .................................................................................................... 9

1.2 RELATED WORK: TRADITIONAL CHANGE MANAGEMENT MECHANISMS ....................... 10
    1.2.1 Integrated Development Environments .......................................................................... 10
    1.2.2 Application-specific Change Propagation .................................................................... 11
    1.2.3 Middleware ................................................................................................................ 12
    1.2.4 Data Exchange Languages .......................................................................................... 12

1.3 THESIS CONTRIBUTIONS ...................................................................................................... 14
    1.3.1 Active Virtual Repositories .......................................................................................... 14
    1.3.2 Propagating Change in Active Metamodels using ECA Rules ..................................... 15
    1.3.3 Schema-driven Integration ............................................................................................ 17
    1.3.4 Algorithms Driven by Property Flags and Association Classes ................................ 19

1.4 OVERVIEW OF THESIS ....................................................................................................... 22

CHAPTER II: OVERVIEW OF APPROACH: PHASES OF DATA INTEGRATION .............................. 24

II.1 THE WRAPPING PHASE ........................................................................................................... 24
    II.1.1 Wrapping a tool ......................................................................................................... 24

II.2 EXECUTABLE EXTENSIONS .................................................................................................... 26
    II.2.1 Tool property routines ................................................................................................. 26
    II.2.2 Derived property routines .......................................................................................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>II.2.3 Action routines</td>
<td>30</td>
</tr>
<tr>
<td>II.3 THE INTEGRATION PHASE</td>
<td>30</td>
</tr>
<tr>
<td>II.4 THE PRESENTATION PHASE</td>
<td>32</td>
</tr>
<tr>
<td>II.5 PRINCIPLES OF TOOL METAMODELING</td>
<td>34</td>
</tr>
<tr>
<td>II.5.1 The Principle of Explicit Declarative Encoding</td>
<td>34</td>
</tr>
<tr>
<td>II.5.2 The Principle of Documentary Accuracy</td>
<td>35</td>
</tr>
<tr>
<td>II.5.3 The Principle of Modularity</td>
<td>35</td>
</tr>
<tr>
<td>II.5.4 The Principle of Posteriority</td>
<td>36</td>
</tr>
<tr>
<td>II.6 SUMMARY</td>
<td>37</td>
</tr>
<tr>
<td>CHAPTER III: INTEGRATING HETEROGENEOUS STRUCTURED DATA</td>
<td>38</td>
</tr>
<tr>
<td>III.1 DIFFERING DATA MODELS AND DOMAIN MODELS</td>
<td>38</td>
</tr>
<tr>
<td>III.1.1 Differences in Data Models</td>
<td>39</td>
</tr>
<tr>
<td>III.1.2 Differences in Domain Models</td>
<td>40</td>
</tr>
<tr>
<td>III.1.2.1 Tools Sharing the same Subject World</td>
<td>40</td>
</tr>
<tr>
<td>III.1.2.2 Tools Analyzing Different Subdomains of the same World</td>
<td>41</td>
</tr>
<tr>
<td>III.1.2.3 Tools Deriving secondary data</td>
<td>42</td>
</tr>
<tr>
<td>III.1.2.3.1 Tools Adding Secondary Data Properties to existing classes</td>
<td>43</td>
</tr>
<tr>
<td>III.1.2.3.2 Tools Adding new Secondary Data Classes</td>
<td>44</td>
</tr>
<tr>
<td>III.1.3 Differences in Document/Stream Formats and API's</td>
<td>46</td>
</tr>
<tr>
<td>III.2 POPULATING WRAPPER METAMODELS</td>
<td>47</td>
</tr>
<tr>
<td>III.2.1 Wrapper Schemata and Schema Extents</td>
<td>47</td>
</tr>
<tr>
<td>III.2.2 Example</td>
<td>48</td>
</tr>
<tr>
<td>III.2.3 Property flags</td>
<td>49</td>
</tr>
<tr>
<td>III.3 TECHNIQUES FOR DATA INTEGRATION</td>
<td>51</td>
</tr>
<tr>
<td>III.3.1 Integration-by-multiple-inheritance Example</td>
<td>51</td>
</tr>
<tr>
<td>III.3.2 Partially Ordering Schemata</td>
<td>54</td>
</tr>
</tbody>
</table>
V.1.1.4 The Application Layer ................................................................. 80
V.1.2 The Choice of Java ........................................................................... 81
V.1.3 XMI vs. Java ..................................................................................... 83
V.1.4 Routine Bindings in the AVR Framework Layer .............................. 85
V.1.5 Constructing Objects Representing Instance and Schema Classes ........ 86

V.2 ISSUES FACING AVR APPLICATION DEVELOPERS ..................... 87
V.2.1 URL’s and Semantic Keys ............................................................... 87
V.2.2 Reflection vs. Dictionaries ............................................................... 89
  V.2.2.1 Factory implementations .......................................................... 89
  V.2.2.2 Property routines ....................................................................... 90
V.2.3 Multiple inheritance, Object Fusion and Property Routines ............ 91
  V.2.3.1 AVR Semantic Checking and Java Reflection .............................. 92
  V.2.3.2 Action routines ......................................................................... 93

CHAPTER VI: CONCLUSION ....................................................................... 94

VI.1 SUMMARY ......................................................................................... 94
VI.2 FUTURE WORK .................................................................................... 95
**Typographical Conventions**

- *Italics* is used to indicate the following:

  1. Technical terms defined in the glossary. These are either terms introduced by this thesis (e.g. 'virtual repository') or a particular application of broader technical terms (e.g. 'information source'). If a term is used several times in a section or chapter, only the first usage will be italicized and bold. Only similar terms that are frequently used in this thesis are defined in the glossary to avoid confusion.

  2. Mathematical or logical variables in definitions and examples (e.g. ‘...the instance node $A$ is reachable IFF...’ or ‘...an $n$-ary association has $n$ association ends...’).

  3. Non-English terms (e.g. ‘ad hoc’).

- **Bold** is used (rarely) to indicate emphasis.

- **Fixed-width fonts** are used to indicate blocks of pseudocode, XML declarations, or references to these in the body of the text (e.g. ‘the schema class GlobalVariable declared in XMI above...’). All such references will be in a fixed-width font.

- **Large, Bold, San-serif** fonts of varying sizes are used chapters and sections.
Chapter I: Introduction

1.1 Problem Description

1.1.1 Integrating Data for Browsing

Enterprises access information from many kinds of source applications. These include databases, transaction systems, servers on the World Wide Web, repositories, file systems and other applications. Integrating data from independently developed information sources presents a range of problems for application developers. This integration is typically done by information services, which usually duplicate the sources in a database. When sources change, these services coordinate propagation of the changes to the duplicate copy of the data [Widom]. Here, differences in information sources’ data representations are commonly homogenized and reconciled under the relational data model [Chen] and a global schema. However, some integration frameworks federate pre-existing databases without duplicating them [Sheth/Larson]. Queries are delegated to subqueries in the member databases and the responses are integrated into a coherent view.

The data-duplication and query-delegation approaches outlined here both assume the presence of a database. A problem with these approaches is that the information sources may be files in the file system, or information filters that derive data from other information sources. An application itself can act as an information source. Information repositories allow client applications to request objects or documents represented in different stream formats. Example clients include web browsers.

Repositories often need to provide a browsing system for navigation between divergent information sources. They seldom need to support a complex query languages as databases do. This thesis
introduces an architecture for a metadata repository that avoids duplicating the sources persistently. In this framework, information can be integrated from tools with varying data models, domain models, data access methods and stream formats. This is done by encapsulating each tool in a common interface and creating a global schema to connect these interfaces into a single service.

This approach is an extension of Cooperative Information Services [Mylopoulos et al. 96] and the Software Bookshelf [Finnigan et al.]. It pursues the same goals as these papers:

“To be general, the framework must make as few assumptions as possible about the software systems being integrated. In particular, the framework should require as little as possible in modifications of the components being integrated. Finally, the framework needs to be open in the sense that component systems can be added and/or removed easily without drastically affecting the functionality or the effectiveness of the overall system.” [Mylopoulos et al. 96]

This thesis addresses the problem of providing browser-oriented data integration without making a priori assumptions about the information sources, filters or clients.

1.1.2 Motivating Example

In order to motivate this thesis, we take, as an example, a software development project. This project consists of:

1. A large corpus of legacy source code under version control/configuration management (VC/CM) or in the workspaces of individual developers.

2. Compiled images of the source in workspaces or shared builds.

3. Information automatically derived from the source or images. These include searchable indices for software artifacts, software metrics, batch testing reports, or quality assurance reports.

4. Related documents about software requirements, specification, or design.
5. Information from automated design tools that record design decisions or alternatives. These include graphical editors like those for UML [Rumbaugh et al.].

The challenge is to present a unified browsable service for this information. The legacy information in the project is a set of independently developed information sources and filters. This presents the same semantic interoperability problems as those introduced above: organizing and managing this changing, versioned, and derived information.

1.1.3 Software Repositories and IDE's

Software repositories provide persistence for language-independent version control/configuration management (VC/CM) systems (e.g. [Katz], [IBM]). These often service Computer-Assisted Software Engineering (CASE) tools such as Integrated Development Environments (IDE's) which are specialized for a single programming language or domain (e.g. Visual Age for Java [VAJ]). Even when IDE's offer a facility for tool integration (usually an application programming interface or API), it is difficult to integrate independently developed software tools because of the broad assumptions the API makes about the tools.

An IDE, if present in a development context, is where source edit operations occur. Thus, its repository of source code, or a developer's workspace behaves as an information source as defined above. Data derived from this source need to be synchronized with the source, the built image of a workspace, or other builds shared by a team of developers.

Software migration and program understanding strain the capabilities of most development environments. New developers joining a legacy software project need to understand features of the target software system quickly. Such projects often face imperatives to migrate to more current source languages, software or hardware architectures, or to assure compliance with some unanticipated
constraint. A variety of program understanding and software migration tools exist for many languages, and these tools can be used to assist in these challenges. They do so by deriving data from repositories or workspaces. However, these supplementary tools are difficult to integrate and synchronize with their sources.

These interoperability problems make browser-oriented data integration difficult. Developers must be able to navigate among the data produced by the different tools. The size and complexity of legacy software systems also present difficulties. However many of the difficulties stem from differences in domain models and data models of tools. Another problem is the fact that such data are stored in myriad formats, and media (if they are stored at all). They may also be stored on a variety of machine architectures. The data are often generated dynamically (e.g. by a parser) or stored in structured files, but rarely in relational databases. These are all problems of semantic interoperability – having tools understand data produced in different contexts by different tools.

l.1.4 Change Management

Another problem is maintaining semantic consistency between the different information sources. This consistency must be maintained when information sources change, and other data are derived from those sources. We consider the compilation units that a developer edits to be primary data, since they are not derived from any other data. Examples of derived secondary data include compiled object files or software metrics. Data derived from secondary data is still referred to as secondary data. In an integration service, policies and rules for propagating change between primary and secondary data must be specified. For example, software metrics generated from source files can be recalculated proactively and stored when those files change. Alternatively, metrics can be calculated reactively when a compilation unit is browsed, and discarded after a request is serviced. To allow context-
dependent browsing over source data requires that all metrics in a context be computed and stored before a query occurs. For example, "what procedures call more than 5 procedures?" is a context-dependent query. The context for the procedure calls might be a compilation unit, a subsystem (a collection of related compilation units), a component (a reusable collection of subsystems), the entire software system, or a product-line of related systems. Answering the request requires parsing all procedures in the context and caching the procedure-call properties before browsing-time.

Yet not all properties are context-dependent and need to be pre-populated. Some properties are context-independent and can be computed lazily. For instance, the query "what local variables are defined by a procedure?" only requires parsing that procedure.

The framework presented in this thesis allows the dependencies of secondary data on primary data to be modeled in the repository. It also facilitates the modeling of dependencies of secondary data calculated from other secondary data. The dependencies form causal chains with complex semantics that require modeling. In our framework, the models control updates to stored, derived data automatically. These models encode the dependencies between information sources and the filters that depend on them.

1.2 Related Work: Traditional Change Management Mechanisms

1.2.1 Integrated Development Environments

Integrated Development Environments are built for a particular source language, with a built-in assumptions about version control and configuration management. Change propagation mechanisms are proprietary, built into the IDE but not exposed to other applications through an API. During
compilation, the dependencies between the primary and secondary data are hidden in the implementation of the IDE. Primary data includes source code and secondary data include compiled object files, search indices, etc. An API may be offered to third party software tools so that they can be integrated as well. The API allows the tools to make requests for versioned entities, navigate among their properties or manipulate them. The API may notify registered tools that a compilation unit has been edited or that a build operation has taken place. Then tools can respond to these changes. However, such active notification is rare and limited in existing IDE API's.

1.2.2 Application-specific Change Propagation

Traditionally, a proprietary application that performs data integration for some enterprise is built over one or more DBMS's, using the data-duplication or query-delegation strategies outlined above. A relational schema for a database application cannot record the complex metadata dependencies needed to coordinate change propagation. "Metadata" are data that describe data. Therefore, with a proprietary solution, metadata dependencies often only appear in database application software, not with the schema in the database. That is, although they are metadata, they are encoded as source-code in the application program outside the database and non-declaratively – as application code. Conversely, declarative metadata are non-executable and can be browsed and managed easily alongside the data they describe. They are often used to indicate complex semantic relationships. This capability is required for a general change propagation framework.

Application-specific change propagation code produces a legacy code maintenance problem in its own right. This is because the control logic of change propagation is hidden in source code. So in order to inspect or change this, a developer must browse the code base and recompile the application. The explicit semantics for maintaining consistency among the sources and filtered data are in single-
purpose, proprietary solutions. There is no framework that can be re-used between applications. As more information sources are added, the cognitive overhead of maintaining the application scales intractably. What is required, instead, is a framework for change propagation that helps visualize metadata dependencies.

1.2.3 Middleware

Recently, systematic approaches have been developed in the burgeoning market of third-party middleware applications. These allow interoperability between legacy databases or transaction systems and user applications [Schreiber]. For example, they may allow a legacy transaction system to be accessible from the World-Wide-Web. Middleware often encodes dependencies for change propagation declaratively. Other approaches in research focus on integrating and modeling processes and data flow using process algebras [Bergstra/Klint]. This more maintainable approach records the complexities of the business logic of an enterprise. However, middleware solutions are offered as products for integrating popular legacy DBMS’s and transaction systems – to add value to those products. They do not allow integration of additional arbitrary legacy document formats, dynamically computed stream formats, or repositories. They do not offer a framework for integration of special-purpose tools.

1.2.4 Data Exchange Languages

The World Wide Web Consortium (W3C) has introduced the eXtensible Markup Language (XML) as a means of data interchange between different applications. In addition to providing a standard, easy to parse syntax, XML allows document authors to create Document Type Definitions (DTD’s). DTD’s are metadata declarations that describe the structure of XML documents in terms of lightweight
production rules. Documents can thus be validated and interpreted in terms of these DTD’s. These assist document format translation as well.

Recently, the Object Management Group (OMG) has chosen to use the XML Metadata Interchange (XMI) format to exchange data and metadata between tools [XMI]. XMI is based on an object-oriented data model. In addition to this, extensions to the basic object-oriented data model in the Unified Modeling Language (UML) are encoded in XMI to supply user-defined constraints in a schema. These can then be enforced on instances of that schema. It is anticipated that many tools will be incorporating XMI interpreters into their more popular products so that schema-driven translation of exchanged data can occur. A consortium including IBM, Xerox, Unisys and Oracle have already designated XMI as the primary standard of metadata exchange for their products. Schemata declared in XMI can generate XML DTD’s. These can be used to transform instance data conforming to one DTD into another DTD’s format. This is done using eXtensible Stylesheet Libraries (XSL’s). This can help integration of instance data for arbitrary tools accepting XML in different DTD’s.

However, there are a number of deficiencies to XMI as it is presently used. (1) More specialized tool vendors are not part of the consortium. (2) There is no integration of information sources – only a translation mechanism from one tool to another. (3) There is no plan to offer an integration API to an array of tools. (4) There is no control integration, coordinated change propagation mechanism or tracing of data dependencies of one tool on another.

The solutions outlined above include single-purpose applications, Middleware and XMI. They are all biased towards narrower problems of data integration than this thesis addresses.
1.3 Thesis Contributions

This thesis introduces a methodological framework for integrating data from independently developed information sources and filters. The framework also maintains consistency among changing, derived data. It does not use either of the mechanisms of data integration and change management currently in use:

1. DBMS-based integration systems (e.g. data-duplication/query-delegation) which are limited to integrating DBMS's. These systems encode metadata dependencies as single-purpose application code. They also include third-party middleware solutions that only apply to popular legacy DBMS's or transaction systems.

2. Proprietary solutions of domain-specific integration environments like IDE's. These are limited to particular platforms, languages, and VC/CM models. They are often built for a single purpose without interoperability in mind. They are therefore resistant to the integration of independently developed software tools. Furthermore, there is little provision for change-propagation from the IDE to the integrated external tools.

1.3.1 Active Virtual Repositories

This thesis introduces a framework for creating active virtual repositories (AVR's). AVR’s avoid duplication of data from sources and filters wherever possible. This is necessary because some stateful tools have arbitrarily scaling data stores (e.g. file systems or VC/CM systems for large, long-lived software development projects). In AVR’s instance data is translated from tools with their own persistence mechanisms or tools that dynamically generate data. Source and filter applications are encapsulated in a queryable framework interface. That makes them easier to integrate. Being able to retrieve data directly from integrated tools avoids having to store all data in the repository. It is sufficient to simulate a virtual browsable repository available over a network by HTTP requests [HTTP] or another TCP/IP request mechanism. AVR’s can adapt non-database information (and databases) into a browsable form. They can also instrument stateful tools like IDE’s, so an application
can offer the same framework interface as other encapsulated tools. By instrumenting legacy tools this way, the AVR framework can integrate legacy components with other applications in a network-centric environment.

migrate legacy tools to network-centric computing.

AVR's avoid data duplication by encapsulating each source application with a metadata-rich wrapper. The wrapper retrieves requested data lazily using the metadata to guide it. Virtual repositories facilitate read-only browsing since they use stateless tools (like parsers) or instrument stateful tools (like IDE's). They do not edit the contents of stateful tools, so they do not interfere with those tools normal operations or purposes.

1.3.2 Propagating Change in Active Metamodels using ECA Rules

The second contribution of this thesis is the concept of an active metamodel. This concept extends the notion of active metadata repositories [Mylopoulos et al. 96]. These combine Event-Condition-Action (ECA) rules from active databases with the rich data models used in conceptual modeling. This produces the metadata-driven coordination mechanism. In ECA rule processing, a monitor process observes an encapsulated tool for change in its state. When change occurs, the monitor notifies a rule-processing Coordinator by sending it a typed event object. The Coordinator retrieves a list of ECA rules that are listening for this class of event. Each ECA rule contains a condition statement referencing the event received. If the condition evaluates to true, the action part of the rule is fired, perhaps taking parameters from the event instance. Then the next rule in the list is evaluated against the event until the list of triggered rules is empty. The Coordinator then processes the next event sent by a monitor.
Active metamodels extend the functionality of active metadata repositories by introducing executable extensions into the otherwise purely declarative schemata. These extension routines can be used in two different ways:

1. Routines can be bound to the properties of the modeled classes. Property routines may take two forms:
   
   (a) Tool property routines initiate actions that gather information from an encapsulated tool and thereby populate or update (1) the associations linking instances of the schema classes or (2) the instance’s attributes (ie. primitive/enumerated values).
   
   (b) Derived property routines derive properties from other populated properties.

2. Routines can be bound to action parts of ECA rules. These are fired if a triggered rule’s condition evaluates to true. These action routines are used to propagate change among stored data derived from other data.

Applications are developed using this framework where each property or ECA action is bound to some executable routine.

The schemata’s purpose is to model the tools which are external to the repository. Data in modeled tools are translated from their raw form into XMI in order to respond to a browser request. This translation, from tool data to XMI, occurs in the property routines. Before encoding the routines, a developer must describe, in the declarative schema, the entities and properties that the tool generates. The information translated from the tool is said to instantiate the declarative classes in the schema for that tool. The declarative information (schema classes and ECA rules) is closely coupled with bindings to the executable routines in a single metamodelling environment. This facilitates the maintenance and evolution of complex metamodels. This is in contrast to database applications where there is no browsable, declarative schema of detailed metadata dependencies. Instead there is only (1) executable code written in a programming language which is decoupled from (2) a skeletal relational schema and
(3) persistent storage. The database, schema and application code then cannot be browsed together.

1.3.3 Schema-driven Integration

The AVR framework is as an application of the metadata-driven programming paradigm. Metadata-driven programming (or schema-driven programming) is an advance over data-driven programming. In data-driven programming, an application depends on data stored outside the application in resource files. The most common example is graphical user interface code, which defines menus with specific actions associated with each menu item. However, the labels in the menus come from resource files. These files configure the application, allowing it to be reused in different contexts without requiring recompilation. For instance, menus for users who speak different languages can be created simply by using different resource files.

Metadata-driven programming does more than supply instance data. An example is a programming language interpreter which can be given a language grammar at startup. Then the interpreter can parse files written in the language indicated by the grammar. It can then decide if they are syntactically correct (whether they conform to the grammar). To parse different languages does not require recompilation of the interpreter. Here the grammar is metadata about the files to parse, whereas the files are the data.

A similar example occurs in the design of XML [Connolly]. XML Document Type Definitions (DTD’s) are metadata that relate to the documents that syntactically conform to them. Applications are written to interpret DTD’s and to process documents that conform to those DTD’s. The applications do not assume a particular DTD. Instead, they are parameterized by a DTD. This shows the evolution of the World Wide Web community from single purpose HTML applications to more broadly applicable
metadata-driven applications.

In the AVR framework, the instance data conforms to this metadata in the schemata. Each instance datum has a declarative schema class with properties. The instance’s possible properties are limited by those in the schema class. More importantly, in this framework, the control-flow of many algorithms used is determined by metadata from the schema. This tells the algorithms how to treat the instance data – just as the interpreter is instructed by the grammar above. At key locations in the AVR algorithms, a schema class is queried about its properties. The code’s control flow is then determined by this schema information. The main purpose of the schema classes is to parameterize the executable code so it behaves differently for each schema. Then the same code can be applied (without recompilation) in contexts that are more diverse.

For example, a schema may be made up of declarative classes that extend other more generic classes. A schema class declares the properties its instances can have and what types of values they take. Declarative classes inherit properties from other classes and make them available to their instances. Thus an instance of one of these classes may be handled in a certain way by an algorithm because of:

1. Its particular instance property values,
2. The properties of the declarative class it instantiates, or
3. The superclass(es) which that class extends,
4. Some richer metadata feature of the data model. These could be property categories, property flags, or properties describing other properties.

Cases under category (1) make the application data-driven, whereas cases under the categories (2), (3) and (4) make the application metadata-driven (or schema-driven).
Suppose the entity `helloWorld()` is a declarative instance representing a particular procedure in a software system. `helloWorld()` declaratively instantiates the schema class `PascalProcedure` which specializes the schema class `Procedure`. This instance datum may be handled a certain way because:

1. Its `numberOfProceduresCalled` attribute has a value greater than 4. This is a data-driven reason.
2. It can have an attribute `numberOfProceduresCalled` since it is an instance of `PascalProcedure`. This is a schema-driven reason.
3. `PascalProcedure` declaratively subclasses `Procedure` and thereby inherits other properties. This is a schema-driven reason involving specialization.
4. The `numberOfProceduresCalled` attribute is flagged as not being persistently stored in the repository. Instead suppose, it is a `virtual` (i.e. dynamically computed, non-stored) property, because it is easily derived from another persistent attribute.

Metadata-driven programming fulfills some of the goals of component-based programming. Here components should be configurable so they can interoperate with other components in a wide variety of settings. Metadata provides a component with information about the data that it will process in a single context. Thus the component becomes more flexible and reusable in different contexts.

### 1.3.4 Algorithms Driven by Property Flags and Association Classes

XMI’s data model is used in the AVR framework to provide metadata about instances. It provides a rich way to describe the schema itself using flags attached to properties (i.e. associations and attributes). In object-oriented models, schema classes have labeled associations (i.e. n-ary relations) linking them to other classes. Attributes take values of primitives or enumerated types.

*Association classes* in XMI extend these data models by allowing associations to declare their own properties as if they were schema classes. XMI’s data model can thus describe relations that hold
between two associations, provide default values for properties, or add commentary about properties. Some of these properties-on-properties are built into the data model used by XMI (for example multiplicity or default value). Others are user-defined flags or other metadata that extend XMI.

The metadata in the repository models the tools. Thus the tools either have their own persistence or the ability to supply information on demand. These tools allow the virtual repository to avoid storing the browsable graph of nodes and links. Only computationally expensive parts of the graph are stored in the repository. Stored properties are flagged as such in the schema by the developer in charge of data integration (the data integrator). Other metadata guides the propagation of changes across the stored data. Property flags and association classes trace dependencies among properties derived from other properties. This occurs when tools derive data from other tools. The metadata guides retrieval from the tools, allowing less information to be stored. This makes the repository more virtual.

Figure 1 shows a UML diagram of part of a XMI schema.¹ In our software engineering example, if developers need to know which procedures a particular procedure calls, then only one compilation unit needs to be parsed and no state information needs to be stored in the schema extent. Therefore no change propagation or monitoring is necessary. In the diagram, the corresponding association end calls for the schema class Procedure is context-independent. Now suppose a static metric value [McCabe] is calculated from the control-flow graph and call-graph of the procedure. A fan-out metric counts the procedures called and global variables that are accessed/assigned by a procedure. That procedure’s metrics only require the parsing of that procedure alone. Thus the attribute fanOut is also context-independent. Properties like these can be discarded after a browser request since they are

¹ Diagrams in this thesis are UML diagrams unless otherwise indicated. In [Rumbaugh et al.] detailed descriptions of UML semantics and notation are provided. It is understood that UML diagrams can be converted into XMI syntax, since all graphical notations are isomorphic with XML syntax elements and attributes. Occasionally there are non-standard UML notations used in this thesis. These can easily be expressed in XMI using its built-in extensibility syntax.
inexpensive to recalculate. In the AVR framework, properties that are extracted from tools are flagged, as \texttt{active} whereas ones that are derived from other properties are flagged as \texttt{derived}. Properties that are stored persistently are flagged as \texttt{stored}, whereas ones that are discarded after a browser request are \texttt{virtual}.

\begin{figure}[h]
\centering
\includegraphics[width=0.6\textwidth]{procedure-calls-derived-metrics.png}
\caption{Procedure Calls and Derived Metrics}
\end{figure}

Now suppose the developers need to know the procedures a particular procedure is called by. This is the inverse of the \texttt{calls} property. They may also need to know how many procedures call that procedure (this is the fan-in metric). Derived properties like \texttt{calledBy} and \texttt{fanIn} can be calculated based on the updated, stored values at browse-time. For example, \texttt{calls} can be inverted to generate \texttt{calledBy}. In this case, the \texttt{calls} property needs to be stored in the \textit{schema extent} and changes monitored and propagated. This is because the corresponding association end \texttt{calledBy} and attribute \texttt{fanIn} are \textit{context-dependent}. Therefore, the timestamp of a source file should be monitored. So if the file changes, the procedure can be re-parsed and the corresponding instance’s stored properties updated. The property flags and association classes record whether a property is
stored or which properties it is derived from. Thus the metadata flags allow precise modeling of an association's state requirements, dependencies on other properties, change propagation and monitoring needs.

1.4 Overview of Thesis

The remainder of this thesis is organized as follows: Chapter II provides an overview of the approach used in the thesis. It describes the three phases of data integration: the wrapping phase (when tool wrappers are written), integration phase (when the mediator schema is written) and the presentation phase (when the integrated data is rendered at browse-time). It describes the different varieties of executable extensions that populate properties dynamically. Finally, it introduces the guiding principles that a data integrator should follow in constructing an application using the AVR framework.

Chapter III goes into further details about integrating independently developed information sources. Design issues faced in constructing schemata for the different wrapped tools are presented. The use of property flags to control metadata-driven algorithms in this framework is emphasized. The use of multiple inheritance to integrate the wrapper schema classes into the mediator schema is introduced. Then essential algorithms for maintaining semantic identity and populating the stored portions of the AVR are described.

Chapter IV introduces the change propagation framework for AVR's. It compares this framework to that of active databases with ECA rules, events, conditions and actions. Example XMI declarations of these are provided. Issues in synchronization, rule processing condition evaluation, termination and action execution semantics are covered.

Chapter V describes the layered architecture of an AVR implementation. This includes the invariant
layers that comprise the framework itself: the ECA rule Coordinator, XMI semantic checking, and framework semantic checking. The use of the Java programming language and Java reflection for wrapper schemata is also introduced. Important design decisions posed to data integrators by Java are also covered.

Chapter VI provides concluding remarks and describes future directions for research with AVR’s.

There is also a Glossary describing terms used throughout the thesis, and a References section which contains an enumeration of references to the literature.
Chapter II: Overview of Approach: Phases of Data Integration

This chapter describes the process involved in integrating the data generated by a set of independently developed tools.

II.1 The Wrapping Phase

The first goal in developing an AVR application is to gather data from the tools that need to be integrated. In order to do this, the tools need to be encapsulated in a framework interface. This way an integration mechanism does not need to know about the idiosyncrasies of each tool. It only needs to be instructed how to treat the framework interface. A developer called "the data integrator" is in charge of this process. The tool-encapsulation is done using separate metamodels for each information source and filter. These metamodels are comprised of schema classes – each with its own properties. These properties are bound to property routines that gather data from the tools. Other classes describe Event-Condition-Action rules for propagating changes in data between tools that derive data from other tools. These have action routines that propagate updates to stored data.

II.1.1 Wrapping a tool

Each tool is wrapped by a declarative schema with executable bindings to the services that the tool provides. Suppose a file system containing source code is wrapped as an information source. In addition, a parser is wrapped as an information filter that depends on this source to generate parse trees. Each wrapper has a schema of classes (Directory, SourceFile for the file-system wrapper or Procedure, GlobalVariable for the parser wrapper). These schemata are instantiated at
runtime for a particular corpus of source code. The instance graph can then be rendered in a selected stream format and navigated by a browser, written to disk or sent to another filter or client. Each tool has idiosyncrasies that are hidden by its wrapper. These idiosyncrasies include differing data models, domain models, representation formats, etc. The integration mechanism in the proposed approach is insulated from these idiosyncrasies.

These wrapper metamodels or wrappers explicitly encode the information sources' and filters' domain models. In addition to describing what data are used/computed by each tool, the schema also records information on the input/output stream formats of these tools, as well as information about the filters' behaviour and the data access services of the sources and filters. Figure 2 below illustrates the conceptual architecture of a wrapper being queried. It is not a UML diagram, but simply shows data flow.

![Diagram of Querying a Wrapper (Data Flow)](image)

**Figure 2 : Querying a Wrapper (Data Flow)**

These schemata are modularly designed so they can be reused easily in different scenarios where different sets of tools are integrated. Because the metamodels are modular, they can be populated independently. This way semantic errors in the declarative structure or the executable extensions can be detected during casual browsing of each wrapper, in isolation. Any semantic errors should be
detected before wrappers are integrated into a more complex model. As the reader may suspect, it is harder to debug a metamodel once wrapper integration has been achieved because data from each wrapper is intermingled with other wrappers' metamodel entities.

II.2 Executable Extensions

Executable routines are declared and are bound to schema classes. These executable routines communicate with a tool. They occur in two forms: \textit{property routines} or \textit{action routines}. A property routine is responsible for populating / updating a property of a schema class with instance data at runtime. A property routine takes two forms: tool property routines, and derived property routines.

II.2.1 Tool property routines

\textit{A Tool property routine} triggers a tool to gather information. The routine then translates these \textit{raw} data into XMI instance data. This normally happens as a result of a browser request for that instance datum. Example schema classes are shown in Figure 3. Suppose the schema containing these classes (and others) models the types of nodes in the parse tree generated by a source code parser. A schema class \texttt{SourceFile} has an association end \texttt{procedures}, which links it to instances of \texttt{Procedure}. Source files contain procedures, as indicated by the diamond at one end of the association. This means that this is a \textit{composite association}. The diamond indicates that strong containment or ownership semantics apply to the association. The end with the black diamond denotes the container class \texttt{SourceFile}. An instance can be connected to at most one container and if the container is copied, deleted or moved, so is the contained instance. Thus, the association’s composition semantics are maintained.
Suppose a new SourceFile instance is generated and displayed in HTML. A user selects its procedures property and may represent as a hyperlink. Once hyperlinks are activated the property routine associated with this property fires. The parse tree is generated (in the tool’s raw output stream format) and this tree is translated into an internal tree representation of XMI procedure instances. This tree is rendered in HTML in the browser. Since SourceFile is effectively populating its own properties, the property is flagged in the schema as <<active>>. Since these data are inexpensive to parse, they can be discarded after the query. Thus the procedure property is also flagged as <<virtual>> (the opposite of <<stored>>).

**11.2.2 Derived property routines**

*Derived property routines* populate a property by deriving data from other properties. These may be stored or computed dynamically. Figure 4 shows schema classes with derived properties. Here, the schema class Procedure has association ends globalVarsAccessed and localVarsAccessed which model the global and local variables accessed by a procedure. These properties link to GlobalVariable and LocalVariable respectively. Two properties are
aggregated (i.e. unioned) to populate the variablesAccessed property. The latter property contains all the variables a particular procedure accesses. Its value is an instance of Variable a superclass of GlobalVariable and LocalVariable. The aggregation can be done at browsetime. It only needs to consult the value of the other two association ends in the same instance of Procedure. This can be done lazily in the AVR, and the derived value need not be stored in the extent. The fact that variablesAccessed is a derived property is shown in the UML diagram by preceding it with a "\(^{\prime}\)". The AVR UML diagrams that show association classes also flag these properties as \(<\langle\text{derived}\rangle\rangle\). In addition, the fact that the class Variable's name is italicized in the diagram means that it is an abstract declarative class. That is, it cannot be directly instantiated. Only its subclasses can.

![UML Diagram](image)

**Figure 4: Procedures Accessing Variables**

Now suppose a developer needs to know what procedures access a particular global variable. This is represented by the association end accessedBy in the schema class GlobalVariable that takes Procedure as its value. Such a model is illustrated in Figure 5. The model in Figure 5 is a context-
dependent association end, since it only makes sense in some context. The developer may only need to know what procedures in a particular subsystem access the particular global variable. Only the procedures in the subsystem need to be parsed which, however may take longer than a browser request allows. Thus, in order to prevent browser time-outs, their globalVarsAccessed links need to be pre-stored so that global variable access links can be readily available. By looking up the values of globalVarsAccessed that reference a particular GlobalVariable, the value of accessedBy for that variable can thereby be populated. This amounts to inverting the property globalVarsAccessed to populate accessedBy. A schema-driven algorithm in the AVR does this. This requires that a context needs to be populated and stored. Thus, this kind of association end requires that the repository not be purely virtual.

![Diagram](image)

**Figure 5: Procedures Accessing Global Variables**

Schema-driven algorithms perform many of the property derivations in the AVR. This is discussed in later chapters. Property flags parameterize these algorithms, providing metadata such as which populated properties should be aggregated or inverted. Property routines, by contrast, gather data from
tools. Another type of routine used in the AVR is outlined in the next section.

### 2.2.3 Action routines

*Action routines* execute the action in the Event-Condition-Action (ECA) rules. These rules consist of (1) a schema class representing a detected *event*, (2) a *condition* that is tested on the event, and (3) an *action* class which describes the action to be performed if the condition evaluates to *true*. The action routine is thus bound to the action class. The purpose of ECA rules in the AVR framework is to propagate updates of stored, derived data. These updates are propagated across a directed graph of information filters that form a chain of dependencies. For example, a monitor could poll the file system for changes to source files. When a polling *event* occurs, the new and old timestamps are compared for equality (the *condition*). If the file has changed, the parser is fired on the source file to generate a new parse tree (the *action*).

The tool routines, derived property routines and action routines make the wrapper schemata active. Once the sources and filters are wrapped by the data integrator, they all present a common framework interface that makes integration easier.

### 2.3 The Integration Phase

The second phase is the integration phase. Each of the tools' wrapper schemata is integrated into a global schema called the *mediator schema* or *mediator*. This is done by the data integrator. To do this she must find semantic connections between classes and properties in the wrapper schemata.

The terms "wrapper" and "mediator" come from Gio Wiederhold's work on data integration [Wiederhold]. The mediator schema tends to be more purely declarative than the wrappers: The
executable bindings in wrappers access services in the tools directly, whereas the mediator reconciles
idiosyncrasies of the gathered data into a common schema without communicating directly with the
tools. A client’s requests for information from the mediator are delegated to the tool wrappers and thus
the tools. This is illustrated in the data-flow diagram in Figure 6.

Figure 6: Integration Overview (Data Flow)

When the virtual repository starts at runtime, wrapper and mediator schemata are registered.
Registration includes both declarative information and executable routines for gathering instance data.
The declarative portion comes from XMI. The executable portion is a library of routines that map to
the association ends, attributes, and ECA rule actions. Once this registration is complete, a small
amount of seed instance data is added indicating the locations of the tools. These data also indicate
where users can begin browsing. Once the stored properties are populated, the AVR is ready to service
browser queries.
II.4 The Presentation Phase

The presentation phase is different from the first two phases. It occurs during the browsing of an integrated metamodel in a virtual repository. As such, it is fully automated by the metadata authored in the first two phases. The schema classes that are integrated in the mediator include enough metadata to generate a browsable output stream for them. Extensible Stylesheet libraries (XSL’s) can be used to transform the direct XMI model into tailored HTML or XML for other DTD’s. Therefore, no application code need be written to specially render XML data in a particular DTD. The HTML/XML-generating routines in the AVR do not assume any particular wrapper schema. They are instead parameterized by schemata for the currently available tools. They use these to guide the output-stream rendering process.

Here is an example of how data from a tool is represented in a browser for navigation. Suppose, an instance representing a procedure has a web page generated for it dynamically from the tool data. The page has hyperlinks that represent instances of association ends, pointing to other instances in the graph. Attributes with primitive values are also shown, but not as links. A browser-request mechanism is constructed so that specific CGI query strings in URL’s can be mapped to particular instances. The AVR is fitted with a schema-driven HTTP [HTTP] request processor. A client browser can send an HTTP request to trigger generation of the output stream representing a particular instance. The AVR maps the CGI query string to the instance requested. It then populates the instance’s graph neighbourhood. This consists of property links to other instances. Then the AVR generates the HTML/XML stream to render the neighbourhood around the instance. Links to adjacent nodes are shown as hyperlinks that each contain an HTTP/CGI-reference to a single neighbouring instance.

The output stream is generated after data about a node is gathered from several tools and possibly the
stored schema extent as well. Each instance node has properties which have been gathered from several tools' data. A user can thus navigate properties derived from different tools. However, all data from all tools is reachable in the same browsable graph. Suppose the repository integrates source code, parsed information and design documentation. A user can then navigate parse trees, call graphs, control- and data-flow graphs, and view source or external documentation—all in a single navigable graph. The multiple views of an instance are presented as links in a browser.

Moreover, the graph need not only be browsed in a single stream format. Certain links can trigger special-purpose browsers or tools with their own user interfaces. These tools would be registered with the repository by having their own wrapper schemata. The wrapper schema for a tool with its own user interface, indicates what classes and properties offer these special-purpose views. They also indicate how to supply information to the tool to trigger these views. Thus, links to these special-purpose views can be inserted in the HTML/XML pages so they can be reachable from the main graph.

Entirely new tools can also be built directly over the AVR. For example, special-purpose client-side Java applets [Gosling et al.] can be written to render views that integrate properties gathered from several tools. The executables for an applet are downloaded from the AVR to the client through an HTTP request. These can extend generic web browsers to handle the different schema classes in a more customized way. They may be metadata-driven or not.

Suppose some tools identify where procedure declarations/calls and variable declarations/fetches/stores occur in the source code. Other tools provide call graphs, and data-flow information derived from parsers. Therefore, it is possible to build a tool that renders source code in HTML with hyperlinks. Here, procedure calls or global variable fetches/stores can link back to the respective declarations of the procedure or global. Alternatively, clicking on one of these items shows a pop-up menu listing all
integrated properties and views of the procedure or variable available in the virtual repository (including a link back to the declaration).

The pop-up menu, which is dynamically computed for a node in the source code, is called a multi-headed link. This and the various presentation strategies above were introduced in a framework called the Software Bookshelf [Finnigan et al.], of which this thesis is a direct descendant. There, the strategies did not involve XMI, wrappers/mediators, ECA update rules, attribute/action routine bindings as in this thesis. The output was restricted to (1) HTML generated CGI-scripts running in a Web server or (2) viewers that were applets. However, the concepts of dynamic generation of web-browsable streams from tool data, driven by declarative schemata were fully introduced in that paper.

One of the main motivations for this thesis is to show how the schemata's metadata can be stored elegantly and declaratively with a minimum of executable code and storage requirements for example data. This complex metadata graph (not just the instance graph) can also be browsed from the same repository for its own debugging and maintenance by the data integrator.

This outline of the wrapping, integration and presentation phases gives an overview of what the data integrator needs to do to create an AVR. This provides the background for the guiding principles that the data integrator should keep in mind when creating an AVR.

11.5 Principles of Tool Metamodeling

11.5.1 The Principle of Explicit Declarative Encoding

The data integrator closely examines each information source and filter that must be integrated. She explicitly encodes, using a declarative language, any relevant semantic information used or assumed
by each tool. The benefit here is twofold. First, the semantic consistency of the metamodel can be enforced in declarative modeling language. This is semantically checked in a modeling language interpreter. The interpreter takes a lexically and syntactically correct document structure, resolves references in the text, and enforces the semantic rules particular to the language chosen.

Second, the data integrator avoids the sprawling effect of incremental, *ad hoc* integration solutions. In proprietary applications, much of the semantic metadata such as tool data models and data dependencies between tools are invisible. They are hidden in routines with no declarative power. Thus these metadata are inaccessible to global browsing and control. By following the principle above, the result is an explicit, browsable, centralized representation of the assumptions and data dependencies of the tools.

**11.5.2 The Principle of Documentary Accuracy**

Incorrect or misguided assumptions implicit in a particular tool must also be included in that tool's wrapper schema. If a tool divides the subject world into entities and properties which the integrator disagrees with, then these entities and properties must still be encoded *verbatim* in the wrapper. The reconciliation of disparities in different tools' pictures of the world is postponed until after the wrapping phase. Once the tools are modeled, a data integrator can recast the tools' wrapper schemata in the mediator schema during the integration phase. The tool can also be used in different integration settings (i.e. in different mediators for different clients). The integrator should not bias the wrapper *a priori* to the needs of a particular integration setting.

**11.5.3 The Principle of Modularity**

By following this principle, the integrator assures that each tool's wrapper schema is to be considered a
It encapsulates the services and data the tool offers—abstracting away its idiosyncrasies. When a tool does not depend on other tools, its wrapper should be created as if the other tools did not exist. This allows the integrator to add or remove tools from the system by adding or removing the corresponding wrappers. If the wrapper schemata are viewed as directed graphs, then reference arcs between schemata should only exist when there are explicit data dependencies between the corresponding tools.

The dependencies are discussed in detail in on page 54. However, other than for these cases, each metamodel should be isolated whenever possible. Thus prior to the integration phase, the schema graphs for tools that do not depend on each other would be disjoint connected components.

### 11.5.4 The Principle of Posteriority

This framework is designed to allow data integration of different application sources. However, the proposed approach is not bound to a specific application domain. The XMI data model has no built-in world semantics to base a domain model on. It also does not supply any primitive concepts that are supposedly general to any subject world (e.g. colour, space or time primitives). The principle of posteriority states that wrapper and mediator metamodels should be created out of the available domain models already explicit or implicit in the available tools or their documentation.

This is a bottom-up strategy. A more classical top-down approach to creating global metamodels in Artificial Intelligence is to design an *a priori* domain model based upon the musings of the integrator and then try and force the tools to provide instance data that conforms to it. For example, one might try to produce a domain model which would generalize the characteristics of both object-oriented and procedural programming languages—assuming refinement of this general schema could encompass
specific languages later. Such an approach puts the data integrator at too elevated a position with respect to the software tools being modeled and it prejudices the mediator schema.

II.6 Summary

In order to produce accurate modular wrappers for available tools, the data integrator should model each tool as in isolation from any other information source. During the wrapping phase, tool routines and derived data routines are coded to supply data to the mediator. Action routines handle updates to stored, derived data. During the integration phase, the mediator is written which combines the wrapped tools into a global, browsable view. The integrator can then compensate for the tools' inconsistencies a posteriori in the mediator, while maintaining documentary accuracy in the wrappers. This way, the wrappers can then be reused as modular components in other mediators. This can be done without biasing them to the integration needs of the first mediator. During the presentation phase, the graph can now be rendered and navigated in a variety of output formats.
Chapter III: Integrating Heterogeneous Structured Data

Tools providing data about similar subject worlds may have differing data models, domain models and I/O stream formats. These differences need to be reconciled by the data integrator. This chapter provides a more detailed view of the intricacies of integrating wrapper schemata. It also discusses in detail how wrappers are populated. This occurs either lazily during browsing or aggressively for context-dependent queries.

III.1 Differing Data Models and Domain Models

Because software systems are lexically, syntactically and semantically defined, they can be described in structured models – classifying the software artifacts and their relationships according to a schema. A variety of software tools exist to analyze this structure and produce derived data from it. Integrating these tools may become a difficult process because each tool structures and emits its data differently. The problem faced is thus the integration of heterogeneous structured data. This is a harder problem than federating a set of databases. In this context, each database has a different schema, transaction/query services, and data representation formats. However, the range of possible semantics and formats in software tools’ output is much broader. Moreover, most tools do not declare metadata about their data (like a database’s schema) and may not have their own query services. Finally, the data they store, in most cases, is not organized into tables, indices and views.

The problem here also differs from the integration of unstructured or semi-structured data. Unstructured or semi-structured information sources include the manually authored documents on the World Wide Web, and natural language documents. Natural language design documents for a software project must be integrated as well. However, they are the exception in a world of structured, syntactic,
tool-derived facts about software. If they were the norm, they would lend themselves to schema-less modeling [Chawathe et al.].

The obvious differences between structured data sources are in the lexical format each tool provides and requires. Wrappers parse these streams and construct an instance graph to answer a request. However, the mediator’s picture of a software project must also reconcile two important structural differences in the tool’s data: disparities in data models and disparities in domain models.

III.1.1 Differences in Data Models

This section presents some of the major data models presented in the research literature to date. Variations of these models may be presented by different tools during the wrapping phase. They may be undocumented and implicit in the tools’ stream formats. But the data integrator must still document them in the wrappers:

- A typed, relational model of entities and properties without inheritance. These models are used in relational DBMS’s. A schema declared in this data model could be an Entity-Relation schema [Chen]. These models are also used in software tools such as stand-alone parsers. These may have explicitly declared schemata or not.

- An object-oriented data model includes single or multiple inheritance of properties between entity classes. Depending on the system, there may be extra metadata associated with properties (e.g. multiplicity, default values, or inverse properties). UML [Rumbaugh et al.], XMI [XMI], and RDF [RDF] are variants of this model, including such extra metadata.

- A data model allowing multiple inheritance of properties, multiple levels of instantiation (e.g. metaclasses in addition to classes and instances), multiple instantiation, or categories of properties. Conceptual modeling languages such as Telos [Koubarakis et al.] include these features among many others.

- A typeless, schema-less data model where there are only instances. There are no classes or schema-level properties, only instance-rules and instance-to-instance associations. As mentioned above, these are appropriate for integrating unstructured or semi-structured
data sources. The TSIMMIS [Chawathe et al.] system is an example of this.

During the wrapping phase, a data integrator converts a tool's data model to XMI's object-oriented data model. The issues faced only concern each stand-alone wrapper.

### III.1.2 Differences in Domain Models

Separate from differences in data models are differences in domain models. Here “domain model” applies specifically to how the source classifies entities in a particular subject world within the expressive limits of the data model. It is constrained by that data model’s declarative power.

Differences in domain models are reconciled during the integration phase. How this is done is specific to each integration setting, since it depends on the other tools being integrated.

#### III.1.2.1 Tools Sharing the same Subject World

Two independently developed sources may offer information about the same subject world but have different pictures of this world. Their domain models classify the same set of concrete entities according to separate criteria. This is illustrated in Figure 7. The parser wrapper in the diagram has already been introduced. Suppose another tool organizes testing activities and classifies software artifacts as *Correct*, *Incorrect* or *Untested*. Classes with italicized names like these indicate abstract classes. Abstract classes cannot be directly instantiated in a declarative model. Because the two wrappers' pictures of the are orthogonal and complementary, a data integrator can integrate both information sources. Thus the same entities would be instances of schema classes in the two hierarchies simultaneously.
III.1.2.2 Tools Analyzing Different Subdomains of the same World

Other sources analyze two or more related subdomains of the same subject world. They each place emphasis on different regions of that world. These concepts are illustrated in Figure 8. Here classes in the hierarchies do not necessarily describe the same set of entities (as they do above). The hierarchies may in fact be disjoint. The figure contains the same parser wrapper as described earlier. There is also a scanner tool, which extracts include-statements in that language. Another scanner extracts program comments. Integrating these facts requires building associations between disjoint but related hierarchies. Each hierarchy is thus represented by the XMI schema of the wrapper for each tool. The purpose of the mediator schema is to make the nodes in each hierarchy and their instances reachable from the others. Classes like FileWithIncludes and SyntacticSourceFile form the bridge
in the mediator schema by being multiply subclassed into `SourceFile`. They multiply inherit properties that link classes within a schema. This makes all parts of the schemata reachable for a browsing user. This is discussed further on page 51.

**Figure 8: Associating Classes from Different Wrappers**

In the figure above, there are no dependencies between the wrappers. In the next section, interdependencies between schemata for derived data are discussed.

### III.1.2.3 Tools Deriving secondary data

A *filter* takes data generated from a primary source and generates data from them. Suppose the parser's data are used to generate static complexity metrics from a parse tree [McCabe]. Another tool generates
a set of substrings from a set of identifier names found in the source code [Anquetile/Lethbridge].

When wrappers are written for the primary tool and the filters, the filters' wrapper schemata extend the primary tool's wrapper schema. They do so by either subclassing classes in the primary tool's wrapper or by referencing those classes as property values. The relationships between the wrapper schemata for the source and filter can be associated for browsing in these two different ways.

III.1.2.3.1 Tools Adding Secondary Data Properties to existing classes

The filter's schema may extend the primary tool's schema by adding properties to its classes (by subclassing them). These properties record the additional information that the filter adds to the primary tool's data. The Procedure class from the parser wrapper has properties describing the static control-flow characteristics of a parsed procedure. Most of these are not shown in the figures. From these, the metrics wrapper generates static complexity metrics about procedures using declarative specification. In such a way, the filter wrapper schema can extend the Procedure class. This is implemented in a class ProcedureWithMetrics by adding new attributes for the already computed metrics. These metrics are denoted as integers and floating point numbers. Some of these metrics are shown in Figure 9 where fanIn and fanOut are derived from properties found in the SyntacticWrapper class. The specification strings avoid having dedicated property routines for the metrics.
III.1.2.3.2 Tools Adding new Secondary Data Classes

The primary source's schema is extended by adding a new class hierarchy to it. These new derived classes may point back to classes in the primary source. Continuing the example in Figure 9, another tool computes all of the significant substrings found in a set of identifiers (e.g. names of files, variables and procedures) [Anquetile/Lethbridge]. This is shown in Figure 10. These properties are extracted using tools, so they have their own property routines. The procedure name substrings are not simply attribute values to be added to the Procedure schema class. Each substring is associated with a set of procedures, variables, files that contain that substring. Several procedures and variables may contain
the same substring. Thus, Substring is a schema class with properties pointing back to procedures and variables that (weakly) contain it. These properties are containedInProcedureName and containedInGlobalName.

As in Figure 9, Figure 10 shows the source’s classes being subclassed adding new properties. Classes like Procedure, and Variable are subclassed adding a containsSubstring property pointing to their substrings. Therefore, the Substring class in the filter’s schema is reachable from the primary tool’s schema. Therefore, the instance graph forms a single, browsable connected component.

![Diagram](image)

**Figure 10 : Deriving New Classes from other Classes**

The differences in domain models discussed above are normally encountered by the data integrator in combination. Tools that derive data may share common assumptions with the tools they depend on, or
The data integrator resolves any domain model differences in the mediator.

### III.1.3 Differences in Document/Stream Formats and API's

A third difference between tools is the data format that the tool supplies. Native tool streams are converted to XMI by the wrappers. A primary source’s data may be sent to a secondary filter in a raw form. For example the file system is a wrapped information source and the parser is another source that requires a raw source stream from the file system.

For example, the metrics generator in Figure 9 summarizes control flow information from the parser, while the substring analyzer in Figure 10 requires a list of identifier and file names from the parser (and file system). The AVR must be able to generate customized streams to pipe to arbitrary wrapped tools. This will require a stream generator that takes data in XMI and converts it to the native format that the filter expects (e.g. RSF [Muller/Klashinsky], TA [Holt], or RDF [RDF]). Since each of these formats is a variant on an object-oriented data model or entity-relation data model, the mapping in each generator from XMI should not be difficult.

However, some wrapped tools do not supply their raw data in a I/O stream. For example, some tools will offer an API for exporting data. An application written using the API can thus inspect the tool’s state. The wrapper in this case, communicates directly with the API. When a filter has an API, its wrapper communicates directly with the source tool’s wrapper in memory to obtain the data it needs. This does not involve the use of an I/O stream such as a filter pipe. Instead, it involves the AVR implementation’s internal representation XMI of the source tool’s data. Information communicated between tools can then be converted to the format required by the filter’s API.
III.2 Populating Wrapper Metamodels

III.2.1 Wrapper Schemata and Schema Extents

In order to perform semantic integration of the wrappers’ instance data, those wrappers must first be populated with instance data. Property routines populate links between instances (instances of associations) or they populate attribute values with primitives. For each property that has been flagged as <<active>>, there is a binding to a property routine.

Most wrappers extract data from tools lazily, when triggered by browser queries. For example, suppose a developer browses integrated data about source code using a lazy wrapper. During browsing a developer may cause the invocation of the property routines that populate the neighbourhood in the graph. For example, suppose the developer browses a procedure to find links to procedures it calls. The virtual wrapper stores only schema classes with property routines that are lazy and forgetful. So the instance data is discarded at the end of each request. A property whose instance links are discarded after a request is a virtual property. Otherwise, it is a stored property. The links that instantiate stored properties are stored in a materialized schema extent — along with the sources and destinations of those links. With virtual properties, as the source code changes, new changes are freshly parsed rendered and forgotten. Therefore, no updates need to be propagated.

On the other hand, in other wrappers there may be stored data derived from primary data. This needs to be updated as the source code changes. The stored instances and links would otherwise go out of date. For example, suppose a developer needs to know what procedures call a particular procedure in some context. In such a scenario, the parse tree must be stored in the schema extent to answer this context-dependent property query. If a source file changes, then the file should be re-parsed and the
stored call-graph should be updated. Then the property routines are re-invoked to update the stored parse tree. This must occur before the user’s request since it is a context-dependent property. The whole context must be pre-populated in the schema extent. For any significant software system with large volumes of data exchanged between tools, populating the context may take too long to do at request-time.

However, pre-population is only necessary because, the context-dependent property is not readily supplied by an instrumented IDE. Most IDE’s have a workspace with indices to store context-dependent information. Wrapping such tools can be done in a way so that there is only metadata in the wrappers. Thus, the *schema extent* of the wrapper would be negligible — it would only contain the *seed instance data* that users could use to start navigation from. Seed instance data usually describes the top-most container. Here it would be the IDE itself which contains software projects (which contain source code, etc.)

**III.2.2 Example**

We return to the software engineering domain example with tools other than an instrumented IDE. In this scenario, the parser parses a single compilation unit and outputs the parse tree in a raw ASCII stream. The property routine translates the stream representing the parse tree into its own schema class instances, which can be used to answer a browser request and then be stored persistently or discarded. If a developer wants to know what procedures `helloWorld()` calls, then a property routine is fired for the `helloWorld()` instance and calls is populated to fulfill the browser’s request. The routine finds the nodes and arcs in the raw parse tree and creates new instances or finds stored instances corresponding to the called procedures. These nodes become the values of `helloWorld()`’s calls. **In Figure 11 the value is printf(string).** Note that in UML,
an instance of a class has an underlined name ending in a colon, followed by the name of the class it instantiates (here, the 

\textbf{Figure 11: Two Instances and a Link}

When the \texttt{printf(string)} instance is created, its own properties are not populated immediately. When the property routine creates \texttt{printf(string)}, it does not recursively fire the \texttt{calls} routine on \texttt{printf(string)} to populate its \texttt{calls} property. Otherwise the entire transitive call-graph fanning out from \texttt{helloWorld()} would be generated when the first instance was browsed.

III.2.3 Property flags

Consider a scenario where developers want to know what procedures call \texttt{helloWorld()}. This is the \texttt{calledBy} property in the schema class \texttt{Procedure} as shown in Figure 1 (on page 21). This property requires the precomputation of the call-graph of some context. Pairs of association ends in a binary association are the \textit{inverses} of each other. For example, \texttt{calledBy} in \texttt{Procedure} is the inverse of \texttt{calls} and \texttt{accessedBy} in \texttt{GlobalVariable} is the inverse of \texttt{globalVarsAccessed} in \texttt{Procedure} (as in Figure 5 on page 29). The AVR stores \texttt{calls} and \texttt{globalsVarsAccessed} and computes the \texttt{calledBy} and \texttt{accessedBy} links by inverting the stored values. Thus, it does not need to store \texttt{calledBy} and \texttt{accessedBy} so they are flagged as \texttt{<<virtual>>}. Some association ends \textit{aggregate} other ends. For example, \texttt{variablesAccessed} in \texttt{Procedure} \textit{aggregates} \texttt{globalsVarsAccessed} and
localVarsAccessed (as in Figure 4 on page 28). If globalsVarsAccessed and
localVarsAccessed are stored, variablesAccessed can be derived from them. Therefore,
the latter are flagged as <<derived>> and <<virtual>> as well. In the AVR framework, the
integrator flags certain properties as <<active>>, <<derived>>, <<stored>>,
<<virtual>>, in associations that provide additional metadata about how each end is handled. This
is illustrated in Figure 1 (on page 21). For example, the links instantiating calls and
globalsVarsAccessed need to be pre-computed for a context and stored. Thus the data integrator
flags calls and globalsVarsAccessed as <<active>> and <<stored>>.

The data integrator also declares which properties a derived property depends on in its specification
string. Then a property can look up its inverse, or perform an aggregation operation. For example, if a
developer needs to find out which procedures call printf(string), then the AVR finds which of
calls links have printf(string) as a destination. The sources of those links (in this case
helloWorld()) are then the destination of calledBy for printf(string). This
specification string is illustrated in Figure 1 (on page 21). The reverse lookup is what is meant by
“inversion” of properties. The data integrator encodes this and encodes the fact that
localVarsAccessed and globalsVarsAccessed are aggregated to form the property
variablesAccessed. Similarly, the specification string is placed close to the association line in a
UML diagram illustrated in Figure 4 (on page 28).

Once the wrappers are populated (either lazily or aggressively) their instance data needs to be
combined in an integrated view by the mediator. This is discussed next.
III.3 Techniques for Data Integration

The main technique used for data integration of wrapper schemata is multiple inheritance of schema classes between the wrappers. This is illustrated in Figure 8 (on page 42). In Figure 8, classes in the mediator schema inherit properties from corresponding superclasses in the wrapper schemata. The data integrator finds corresponding classes in different wrapper schemata manually. Since property routines are bound to each property, they are multiply inherited as well into the mediator schema's classes. The mechanism for this will be discussed in the implementation chapter below. Similarly, the update ECA rules for the wrapper classes are also inherited.

III.3.1 Integration-by-multiple-inheritance Example

Suppose there are three tools that have wrapper schemata. A subset of each of these schemata and a mediator schema are shown below in Figure 12. This figure also illustrates how schemata reference each other using properties and how the mediator inherits schema classes.
The first tool is the parser as described above. The second tool is the local file system itself. Its wrapper identifies directories, make-files, source code files and compiled object files for the same language as the parser. Only SourceFile is shown. The third tool is a scanner which extracts comments associated with program statements. A comment is associated with a statement if the comment immediately precedes the statement. A statement is identified by (1) the source file in which it occurs.

Figure 12: References and Inheritance between Schemata
and (2) its line number and (3) its character offset. The association end comment in the class Statement is a domain-independent end (so it does not need to be stored). Because of this, the ScannerWrapper schema need not maintain an extent and is thus called a virtual wrapper. Its instances and properties can be computed at browse-time, purely lazily. It can also acquire its seed instance data (sourcefiles) from the extent of the file system wrapper.

Notice that the class SourceFile occurs as an association end type (for the definedIn end). This is an example of a dependency between two schemata. This is shown in Figure 12 by the arcs traversing schema boundaries. The inter-schema dependency exists because SourceFile is defined in another schema wrapping a different tool.

Some wrapper schema classes will have no correspondents in other wrappers – that is, they cannot be merged with other classes in the mediator. These are simply transferred into the mediator schema as members (this is occurs with Comment above in Figure 12). When instances of classes are browsed, the queries for property values are delegated to each wrapper superclass from which each property was inherited. Then this property query triggers the bound property routine to determine the property’s value. For example, if a user browses an instance of CommentedProcedure, the inherited properties definedIn and comment make the instance graphs of the three wrappers reachable. This can be seen in Figure 12 above, since these connect comment procedures, source files and comments in the mediator.

In Figure 12, CommentedProcedure multiply inherits from Statement and Procedure. It must thus associate instances of Procedure and Statement that correspond to the same item in the world. Suppose the Procedure instance helloWorld() is declared on line 5 of SourceFile “helloWorld.psc”. Its declaration begins at character offset 0 on that line. The
line number and offset are used to associate a statement instance in the ScannerWrapper schema. So by mapping values for properties definedIn, lineNum and offset, corresponding instances in different schemata can be looked up and fused instances created in the mediator schema.

III.3.2 Partially Ordering Schemata

Unless one tool computes its data from another tool's data, each wrapper metamodel (including its instances) should be a disjoint set of connected elements (a connected component). This is because independent tools operate and are modeled autonomously. However, the four schemata in Figure 12 (on page 52) are not independent, so their extents will not be either. The subclass arcs and association arcs that cross schema boundaries in Figure 12 are aggregated into dependency arcs in the Figure 13 below. Such graphs give a high level view of the dependencies between schemata.

![Figure 13: Schema Dependency Graph](image)
In Figure 13 the partial ordering of the schemata is shown. There are no cyclical dependencies, but it is not totally (linearly) ordered either. It is based on data dependency arcs. A dependency arc forms indicating schema $A$ depends on schema $B$ when:

1. A class in schema $A$ subclasses a class in schema $B$,
2. A class in schema $A$ has a association end that has a class in $B$ as a destination or
3. A class in schema $A$ has a property, which is derived from a property declared by a class in schema $B$.
4. An event class in schema $B$ may trigger an ECA rule declared in schema $A$ that is listening for this type of event (this is a special case of 2).

A partial ordering is possible when the subclass arcs, association arcs or property derivations and event triggers do not form cycles between schemata. For tools that have been developed independently, dependencies are rarely mutual. It is possible for inter-schema association arcs to form cycles (breaking their partial order). However, this only occurs where tools are built mutually assuming each other's existence. Cycles cannot occur with subclassing alone. Similarly, no set of derived properties can form a dependency cycle. There must be a tool property ending the dependency chain. Cycles may form when chains of different kinds of arcs form. Events trigger rules with actions that, when fired, produce other events. These can potentially form cycles. It is left to the data integrator to prevent infinite event loops at runtime. However, this is the same burden imposed on software developers by executable languages. When two schemata do mutually depend on each other, they are collapsed into a single node in a schema dependency graph. Then the partial order can be restored if it is important for visualization purposes.
The discussion above has introduced some of the basic mechanisms for wrapping and integrating a set of tools. The next sections introduce some lower level issues in populating schemata with instance data.

### III.4 Population and Semantic Identity

#### III.4.1 Seed Instance Data and Stored Data

When the AVR starts, wrapper and mediator schemata are registered. These include XMI declarations and executable routines. The location of a small collection of initial seed instance data is also registered in XMI. In the example, these data represent directories containing source code files in the file system wrapper in Figure 12 on page 52. From the directories, the source files are enumerated, and from these the parsed contents of those files (procedures, variables, etc.) populate the repository’s extent for stored properties. When property routines create new instances, several issues are introduced.

#### III.4.2 Semantic Keys

When it is necessary to store a property, no instance can be populated twice. Two distinct property routines may create two links that point to the same instance. The second routine does not know that the first has created an instance. In order to prevent duplication, the data integrator must associate a semantically significant key with the identity of each node. Once this has been achieved, a lookup operation on the key can be performed before a duplicate node is created. So when a parser parses a source file, it can use the name of the procedure as part of the semantic key for the Procedure instance. This was done with “**helloWorld()**” above. Then if other procedures in other files call this procedure, the virtual repository can resolve all references (calls) to the same node in the graph.
When these links are created, they are constructed out of information supplied the tool. Keys must contain enough of these semantics so every property routine can generate a semantic key for the same entity in the world. Generating a key is based on (1) a given subject world, and (2) a wrapped tool which generates instance data about that world. Thus, two instances with the same semantic identity, but created by different routines, must have the same key. This allows the AVR to determine semantic identity and thus avoid duplication. This is illustrated in the next section.

Several tools may have different name spaces describing the same entities. Thus each wrapper and each mediator has its own name spaces and lookup tables for semantic keys. A mapping function may be necessary to translate a name in one tool to the corresponding name for the same entity in another tool. This could be done declaratively using regular expressions or executably using call-outs. For example, the Software Bookshelf [Finnigan et al.] includes a name translation service. This mapping function is especially important in the mediator. The mediator must also associate and fuse the instances in different schemata that multiply inherit properties.

III.4.3 The Lazy Creation Algorithm in a Property Routine

Suppose a property routine generates new instances with semantically significant keys. When a new instance is to be created, the necessary data to create a key is available to the routine just before the instance’s creation. This requires that the key can be generated from a raw instance data stream from the tool. An identifier can then be used as a key in a lookup to see if the desired instance has already been created. If the lookup returns nothing, a new instance is created, added to the lookup dictionary and is added to the link being populated. Otherwise the pre-existing instance is returned by the lookup and the link is populated. In our example, the routine for the `calls` property of the class
Procedure would be expressed as follows:

```c
void calls(Procedure caller) {
  1     Create an empty list of instances that will be the links for "calls".
  2     Get a list of strings from the wrapped tool, where each string is the name of a procedure, which is called by caller.
  3     For each string returned {
      3.1    Generate the key from the string (possibly transforming the string if necessary).
      3.2    Do a lookup on the key to see if a Procedure instance with the same key has been created.
      3.3    If it already exists, add that instance to the list of links.
      3.4    Otherwise create a new instance for the procedure and add that to the new link.
  }
  4     Assign the list of links to the caller's "calls" association end.
  5     Do any necessary semantic checks imposed by the schema (e.g. type-safety, multiplicity, default values if the list is empty, etc.), and recover from any semantic errors that are reported.
}
```

III.5 Traversing and Inverting the Instance Graph

III.5.1 Mediator Schemata and Graph Traversal

Suppose there is more than one tool with virtual, context-dependent properties. The data integrator wants to make these available by inverting stored, context-independent properties. This requires a context-wide precomputation of the integrated instance graph. Some tools depend on other tools, so the mediator schema pipes data to the filter tools as in Figure 12 page 51. For example, to parse all source files in a directory tree, the parser’s wrapper must be told by the file system’s wrapper, what source files exist. The mediator relays this information from one wrapper to another as needed.
In order to generate a single instance graph from several tools, the mediator performs a control integration role as well as a data integration role. A mechanism traverses the graph keeping track of the current instance whose properties are being populated. This way, during the precomputation of an instance graph, the traversal mechanism populates the next instances to traverse by generating property links.

### III.5.2 The Walker

The data integrator flags all the properties that are context-dependent. These are derived from stored properties. Stored properties require generation and traversal of the instance graph. An executing process called a *walker* performs this function. The walker is essentially an iterator that traverses an instance graph as a depth-first tree traversal. It does this while generating the nodes in the immediate graph neighbourhood of the current node. It starts with a set of pre-specified root instance nodes. These are the *seed data*. In addition to generating the links that form the tree, the walker generates all the stored, non-tree links like *calls*. Since a procedure call may be recursive, it may form a cycle, so it cannot be a composite property and cannot be used for traversal. Otherwise, the walker would enter an infinite loop if a node’s call graph formed a cycle.

The walker generates and stores the *<stored>*> links of each node (like *calls*), and proceeds to perform the same operation recursively on the node’s children. It descends a tree as it generates an extent graph. The walker uses a subset of context-independent properties that are flagged as composite to descend the tree.

The following properties are all composite: a directory containing a subdirectory or source file, a source file defining a global variable or a procedure, a procedure defining a nested procedure or a local
variable. These are illustrated in Figure 14 below.

Figure 14: Composite Associations for Graph Decomposition

III.5.2.1 Recording Visited Nodes

During the traversal, the walker has no record of nodes visited, although lookup tables store nodes by key. The traversal is depth-first so the walker only needs to know:

1. The current instance node.
2. Which properties should be stored.
3. Which nodes it has already populated and stored (to prevent duplication).
4. Which association ends in the schema are composite — that is, which ones determine the links to recurse on.

The walker does not need to record which links it has already recursed on for the current node (ie. the path back to the root). This is because the recursive depth-first algorithm lets the runtime call-stack perform this role. The algorithm walk shown below satisfies the list above. Note that the algorithm
checks the aggregation flag of the association end to see if it is composite.

```java
void walk(Instance instance) {
    Store instance if it is not already stored (looking it up first using its semantic key).
    For each of the properties in instance: property {
        If property is "stored" {
            2.1.1 Calculate property's value using its property routine and store it.
        }
        If property is an association end (ie. not an attribute) and its "aggregation" is "composite" {
            2.2.1 walk(value)
        }
    }
}
```

Remember this algorithm uses a semantic key on each node to tell if a matching instance already exists (as introduced on page 56). When a link is calculated from the property routine, an algorithm similar to the one on page 57 is used. Since it is a depth-first traversal, only the path back to the root needs to be maintained. If a minimum spanning tree had been used, a data structure would be needed to record the entire graph first and then to record the subgraph representing each tree. All this would consume much more memory than a cursor, a call-stack maintaining the path to the current tree's root and a lazy system of property population.

By flagging certain properties as composite, the data integrator controls how the walker recursively descends the instance tree. Thus, the metadata-driven algorithm avoids algorithms with greater memory requirements like minimum spanning tree.
III.6 Summary

This chapter has discussed the differences in tools to be integrated. These include differences in data models, domain models and stream formats or API’s. Techniques for populating schemata and integrating gathered data were also introduced. Now the data integrator must maintain the semantic consistency of the derived data which is stored and which depends on other changing tools. Change must be propagated from sources to the filters that depend on them.
Chapter IV: Change Propagation

IV.1 Introduction

This chapter introduces the mechanism for change propagation in the AVR framework. Once a portion of the extent is populated, it needs to be updated as the information sources change. Once stored properties are updated, then changes need to be propagated from the stored properties to derived properties. Tool wrappers populate stored properties. Therefore, the AVR has the tools at hand to initiate necessary actions in the wake of a change in any monitored source. The goal is to coordinate the actions in the tools when sources change, and integrate the results into the schema extent.

The mechanism for this coordination is based on CoopWARE (Cooperation With Active Relationships Enforcement) [Mylopoulos et al. 96]. Like the AVR framework, CoopWARE employs a declarative conceptual modeling language. In CoopWARE's case, it is Telos [Mylopoulos et al. 90]. CoopWARE facilitates the integration of heterogeneous, distributed information services. These are sources, filters and clients that may provide other services. It uses the notion of Event-Condition-Action (ECA) rules from active databases. ECA rules can handle information propagation across unreliable networks. As shown in [Gal], the flexibility gained by using ECA rules in distributed systems is that they allow multiple paths for an information system to receive data. They also facilitate activating consistency maintenance operations. However, in the AVR framework, ECA-rules are used more because of the heterogeneity than the distribution of information sources. ECA rules are based on event-notification across distinct threads of execution. This is an alternative to if-then-else statements for conditional control-flow. Therefore, for CoopWARE and the AVR framework, ECA rules are useful since, in both cases, change propagation is asynchronous.
### IV.2 Event-Condition-Action Rules

Wrapper schemata contain monitors in separate execution threads. These detect changes in wrapped tools. Monitors communicate with the AVR's Coordinator—the change propagation rule processor. When a monitor detects change in a tool, it notifies the Coordinator with an event. Notification consists of the insertion of an event instance in the Coordinator's event queue. This inserted event has properties determined by its class, which describe the change detected in the tool.

The ECA rule classes form a dependency graph that can be used to visualize update dependencies in the AVR. ECA rules are classes stored with each schema. The instances of events, rules and actions may also be stored, or discarded. When stored, these instances model the sequence of events and actions in the history of the AVR. They form a log allowing rollback of AVR transactions. If the modeled dependencies were not declarative, but were merely source code instructions, they would be harder to browse and visualize. However, since they are declarative, the consequences of changes to the ECA rules are easier for data integrators to assess.

The Coordinator is an ECA rule processing engine which

1. Receives typed events from monitors that are registered with it. Their schema class indicates their type.

2. When the Coordinator is receives an event, it searches its list of ECA rules for ones \textit{listening} for the type of event received. These rules are then considered \textit{triggered}.

3. For each triggered rule, the rule's condition is evaluated, based on property values of the event received.

4. If the condition evaluates to \texttt{true}, the Coordinator fires the rule's action and proceeds to evaluate the next triggered rule in the list until the list is empty.

ECA rules in active databases, CoopWARE and AVR's provide a uniform encoding of change
dependencies. The encoding is uniform since the rules are declarative, strongly typed statements that are centrally interpreted by the Coordinator. In contrast, modifying such rules in a traditional application requires inspection of much of its source code.

ECA rules in the AVR framework require a constraint language. XMI lets different constraint languages be used. For instance, the contents of a `<specification>`...`</specification>` element in XMI is an arbitrary string – not XML. These can then be read off and interpreted at appropriate times during rule processing. This allows ECA rules to offer more complex, semantically based rule enforcement.

ECA rules can also specify the remedial actions that respond to violations of constraints. This response need not be the simple rollback of a transaction but could be a full range of actions that avoid rollback. Suppose a transaction composed of several events, rule evaluations and fired actions. Rules can compensate for the failure of actions in the same transaction and avoid rollback of the entire transaction. For instance, a new instance insertion action may fail because it violates some constraint. Suppose this is due to the presence of a previous instance with a matching property. An ECA rule listening for the failure event can fire an action to delete the offending instance and retry inserting the new instance.

### IV.2.1 Example

Here is a more detailed example of AVR change propagation. Suppose an event indicates that a directory storing source files has changed. A monitor has been polling the `timestamp` property of a corresponding `Directory` instance periodically. The `timestamp` property routine handles the actual communication with the file system. Suppose a directory’s timestamp changes when a file has
been added, deleted or modified in that directory. (However, we admit this is not generally true for file systems). The data integrator’s ECA rules declare that, when a timestamp changes, several possible actions fire in the AVR:

1. When a source file is added to a directory, a corresponding `SourceFile` instance is added to the repository. Then the parser wrapper parses the file and new `<<stored>>` properties for the `SourceFile` are stored. To do this, the AVR launches the *walker* (described on page 59) on the new source file to coordinate the population of the `<<stored>>` properties. This allows all integrated tools to participate in a centrally orchestrated population and update process.

2. When a source file is modified, its corresponding `SourceFile` instance is removed from the extent. Also, its *composite property* children (instances of `Procedure`, `GlobalVariable`, `LocalVariable` declared in the file) should be recursively removed as well. Then an empty `SourceFile` instance is re-inserted. Then the walker is launched over the re-inserted instance as in (1.) above. This then repopulates its properties.

3. Any deleted source files and their composite children should be recursively removed from the repository’s schema extent.

Now we describe the parts of an ECA rule in more detail.

**IV.2.2 Event**

An event class in an ECA rule indicates which type of event the rule is listening for. In the example, an ECA rule listens for an event that signals change in any directory timestamp. In general, an event instance is generated immediately after an *insert*, *modify*, or *delete* operation on the extent. Here, an event is generated when a monitored `Directory`’s *timestamp* property is modified. The monitor in the file system wrapper polls the file system periodically for change. This requires that the monitor retain a copy of the timestamps for each directory it monitors. Then it can compare the current value to the stored value. When the system starts for the first time, no wrapper extents are populated. Therefore, the walker must traverse the transitive closure of composite property links. Stored
properties are then populated before the Coordinator can begin listening for events.

Each event class also has its own timestamp attribute indicating when a change was detected. This also indicates when the event instance is generated. This way, events can be prioritized chronologically, regardless of the order in which they arrive at the Coordinator. Of course, only events accumulated in a queue can be prioritized. They must accumulate in the queue in order to be reordered.

When an event reaches the front of the queue, the Coordinator searches for rule classes that are listening to an event of this type. For a rule to be listening for an event, it must have an event property with a value corresponding to the event’s class. When a listening rule is triggered, its condition property will be tested. For each triggered rule, a new rule instance is created. The rule instance’s event property links to the event instance that triggered it. The properties of the event instance are evaluated in the rule class’s condition, and the outcome of the condition evaluation (i.e. true or false) is recorded in one of the rule instance’s attributes.

The event is instantiated throughout the evaluation of all triggered ECA rules and the resulting actions. Event instances also never change their properties after they are generated. After the last fired action commits, the event may be discarded or retained. Retaining the event instance and triggered rule instances may allow rollback or other compensatory actions later.

The event class shown below specializes a more general event class (Event). Event defines the timestamp instance attribute. In Figure 15, the class ChangeEvent_Directory_timestamp inherits that attribute from its superclass Event. ChangeEvent_Directory_timestamp also tells the monitor for the file system’s wrapper schema that it should watch the timestamp property in Directory. When the timestamp property of the Directory instance changes, a new
instance of ChangeEvent_Directory_timestamp should therefore be generated and sent to
the Coordinator as a notification.

The specification string "DETECT_CHANGE_IN Directory.timestamp" in the event
class says, "the monitored change is always in the timestamp property of the monitored
Directory instance". This property also tells the monitor that "all instances of the property
Directory.timestamp are to be monitored for new insertions".

**IV.2.3 Condition**

The rule's condition is evaluated when a rule is triggered and it is the next rule to be processed among
currently triggered rules. Here, a sufficiently expressive language must be chosen for encoding the
condition. The requirements of the language chosen are as follows: The condition statement
must be able to reference (1) the properties of the Event instance and (2) any global data in the virtual repository. For example, suppose a query language can express simple constraints on XMI instances and their properties. Now if the query language allows these expressions to be composed with Boolean operators and equality, this would certainly be sufficient. Global data could be queried using that language and properties could be referenced from the Event instance as needed.

Figure 16 below shows the declaration of an ECA rule that is triggered by instances of the event class ChangeEvent_Directory_timestamp:

![Diagram of ECA Rule for Directory Timestamp Changes](image)

The ECA rule above is interpreted as:
"When the timestamp of the directory changes, compare the set of files currently in the directory to the set of files in the old, stored instance of the Directory. If the set has changed because of addition, modification or deletion of files, then fire the Action_Incrementally_Rewalk_Directory action."

The action performs the necessary actions described on page 66. Note in the condition's specification that, for any two SourceFile's to be considered equal, both their name and their timestamp properties must be equal. Therefore, if a file were modified, its instance would fail on equality when compared to an old instance. Thus, sets of files would also fail on set equality if one set contained a file that was a modification of a file with a matching semantic key in the other set. Similarly, set difference considers the timestamps of the files.

An alternative way to encode a condition is to make it a call-out to the executable implementation language. This would be similar to an executable extension in a property routine or action routine. Executable conditions provide an efficient means of evaluating the typed events and global data. Its return value indicates whether the condition evaluates to true or false. Executable conditions can compute complex numerical data efficiently. They may also involve references to external tool data in its unwrapped raw form. The disadvantage of executable conditions is that they are non-declarative and are therefore difficult to browse along with the dependency graph of ECA rules and wrapped/integrated schemata.

**4.2.4 Action**

Every rule's action property declares an action class, which is instantiated if it is fired. The properties of the Event instance can be bound as arguments to a fired action routine. An action may trigger other events, or it may request that constraints be checked after it commits. These enforce the consistency of the repository extent. When an action commits it may return a result. This result and the
results of the constraint evaluations may notify the Coordinator of a new event. Such an event can
determine future actions or decide if rollback of the action is necessary.

Actions may be bound to action routines or may contain declarative specification strings. Here is an
example string:

"DELETE(RETRIEVE(rule.event.monitoredInstance.key).files -
     rule.event.monitoredInstance.files);
WALK(rule.event.monitoredInstance.files -
     RETRIEVE(rule.event.monitoredInstance.key).files"

The way this action is fired is as follows: The specification uses path query statements and several
upper-case keywords. A path query traces a path of property values by chaining the property names,
separated by periods. Path queries define a traversal of the instance graph that terminates in an
instance, a primitive or a set of instances/primitives. Here both path queries terminate with a set of
SourceFile instances to perform operations on. The first property is “rule” which refers to the
ECA rule that fired the action. “event” refers to the event property in of the rule. The rule is an
instance of ECARule_Directory_timestamp_change. The event property of this rule is an
instance of ChangeEvent_Directory_timestamp as in Figure 16 on page 69.

“monitoredInstance” refers to the monitoredInstance property in the event class. This is
a Directory instance, as in Figure 15 on page 68. Now “key” and “files” refer to the key and
files properties of Directory (also in Figure 15). As in the condition statement of
ECARule_Directory_timestamp_change, “RETRIEVE” retrieves the stored instance by its
key (in this case the old Directory instance). The action specifies the following AVR transactions:

1. Delete the SourceFile instances corresponding to deleted and modified files. The list
   of SourceFile’s to delete is the set-subtraction of (1) the deleted files and old copies
   of the modified files from (2) the unchanged files and new copies of the modified files.
   Because equality for two files includes the timestamp, the set subtraction will consider old
   and new copies of modified files as distinct set elements.
2. Populate the properties of the new files and modified files using the walker for the mediator schema. The list of files to traverse is the set-subtraction of the unmodified files from the newly added files and the new copies of the modified files.

The specification string is defined in the action class. So the fired action instance contains free variables, which are bound to it when it is instantiated. In general, actions will contain free variables, which parameterize their operations at runtime. They are either computed from the event class’s properties, or are globally visible.

The other way to encode actions is to use strings that map to action routines. A reflective, interpreted executable language is desirable for this kind of action invocation. This allows mappings between strings in XMI elements to routines in the executable language. For example, the string could map to a method in the Java programming language [Gosling et al.]. Java allows mappings between strings and routines to be invoked at runtime. This and other suitable features of Java are discussed on page 81.

Note that in the example, the ECA rule forms a bridge that activates information filters when a source changes. This is done through the walker and mediator schema. Thus, the file system schema notifies the parser’s schema when files change. For this to happen, the parser’s wrapper must be listening for those changes. Thus when the wrapper is registered with the AVR, it registers the rules listening for an ECARule_Directory_timestamp_change. All inter-tool dependencies and propagations of change are modeled this way in the AVR framework. When the file is re-parsed, the walker retraverses the files — reactivating tools as necessary to repopulate all <<stored>> properties. In this case, it reactivates the parser storing the indicated properties.
IV.3 Rule Priority and Conflict Resolution

When an event triggers more than one rule, a policy of rule priority decides the order in which rules are evaluated. The problem has been investigated by numerous researchers within the area of artificial intelligence and expert systems. The conflict resolution policy thus controls the order in which rules' actions are fired. When a rule is triggered and its corresponding action is fired, it may generate more events. These are only processed after all of the triggered rules are evaluated against the first event. Since no two events are generated in the same clock cycle, the events can be put in a queue that is ordered over timestamps. However, when multiple rules are contending for the same event they must be prioritized as well. Here are four possible policies:

1. One policy is to not prioritize at all. Here, triggered rules are evaluated against an event non-deterministically. This demands that the ECA rules produce no race conditions or deadlock.

2. Another policy is to order the ECA rule classes totally or partially.

3. Pair-wise relative priorities must be capable of ordering any set of triggered rules (totally or partially) by comparing rules in pairs.

4. Numeric priority values can be assigned individually to rule classes. However, these are difficult to maintain as new rules may be added later.

The AVR framework does not dictate the rule priority policy. It simply provides the appropriate interface to be implemented by wrapper/mediator schema designers: Given (1) an event instance and (2) a set of triggered rules, the policy can decide which rule to evaluate against the event next.

IV.4 Dependency graphs

A detailed inheritance graph is shown in Figure 12 (on page 52). In addition, an aggregated rendering of this schema dependency graph is shown in Figure 13 (on page 54). In Figure 12, source files are
updated, and then need to be re-parsed by a parser, but not the scanner. The scanner extracts the program comments and associates them with statements. The parser’s wrapper provides context-dependent properties (Procedure.calledBy and GlobalVariable.accessedBy among others). The scanner wrapper’s properties are all context-independent. The portions of this dependency graph that are stored are maintained by incremental re-traversals fired by ECA rules like ECARule_Directory_timestamp_change in Figure 16 on page 21. This follows the algorithm on page 61, using the mediator schema as shown in Figure 12 on page 52.

IV.5 Termination of Rule Processing

Termination of rule processing is not guaranteed in the AVR framework. It is similar to a programming language in this respect. It is also similar to detecting deadlock in distributed systems. However, a schema-level restriction can encourage confidence in termination: No causal cycles can exist in any ECA rule dependency graph. When existing rules are modified or new rules are added to the Coordinator’s ECA schema, they must form partially ordered chain of dependencies. The dependency graph resulting from any change to the ECA schema is traversed quickly using a breadth-first search to guarantee that it is a directed, acyclic graph (DAG). Properties derived from properties defined in another schema form dependencies that this algorithm can trace. Actions declare the events that result from them. The tracing consists of examining the property specification strings used throughout this thesis. Since schema-level changes are not performed frequently, the space and time costs of this framework-level (ie. non-XMI) semantic check are not considered exorbitant.

Another heuristic that can be applied is to suspend all actions in a queue until currently triggered rules have been processed. Suppose a set of triggered rules is pending evaluation against the same event. Then the currently evaluated rule fires an action. This action cannot delete or modify any instances that
the event references, nor properties required to evaluate the triggered, pending ECA rules' conditions.

Otherwise, spurious results may occur when the other rules evaluate their condition statements. Nor can any other non-ECA transaction change these referenced instances.

Therefore, the action is put in a queue of pending actions. These are fired when the set of rules is finished being processed for a single event. During processing of the action queue, event processing stops, since the action may produce more events. Therefore, the Coordinator alternates between processing a queue of rules on an event and processing pending actions generated by rule evaluation.
Chapter V: Implementation Details

V.1 Overview of Implementation

V.1.1 A Layered Architecture

The AVR implementation consists of several components. The core is made of the declarative wrappers and their executable extensions. These are XMI schema classes, their stored extents (if any), and the property routines and action routines. In addition to the core, there is an HTTP service, which accepts HTTP requests. It maps the requests’ CGI query strings to (1) the instances of schema classes stored in the schema extents or (2) invocations of routines to populate these instances. They can also be mapped to schema classes (including ECA rules, event and actions) for schema-level browsing.

The core is a layered interpreter. That is, model interpretation is separated into a layered series of components. Each layer depends on the layer immediately beneath it. The bottom three layers are written purely in the Java programming language [Gosling et al.]. The fourth layer is a combination of Java wrappers and native tools. Java objects are used to represent the various data structures in the AVR framework. A schema class, like Procedure is parsed from XMI and represented as a Java object. A declarative XMI instance, like helloWorld(), is also represented as a Java object. Thus the word “instance”, in this thesis, has only referred to declarative instances, whereas “objects” will henceforth indicate Java programming language objects. Java and declarative instantiation are kept strictly orthogonal in the AVR implementation. This is illustrated in Figure 17 below. Schema class objects Java-instantiate a single Java class XMISchemaClass. The Java class offers only accessor/assignment (“get”/“set”) methods. Declarative instance objects all implement a Java interface
XMInstance and Java-instantiate arbitrary Java classes written by the data integrator. These allow arbitrary behaviour and roles for instance objects in various settings. Java instantiation is shown by the descending vertical arrow in the figure. At the same time, an instance object declaratively instantiates a schema class object. This is shown by the left-to-right arrow in the figure. This is discussed later in greater detail.

The layered architecture of the AVR framework is shown below in Figure 18. Each layer will be discussed in a subsequent section – starting at the lowest layer.
V.1.1.1 The XML Layer

The base layer of the architecture is an implementation of an XML parser and XML generator. The interpreter consists of a library of Java classes modeling the generic XML syntactic structure. It can parse XML streams into Java objects. It can also output streams of XML elements and DTD’s that marshal Java objects in memory. Correctness checking at this level consists of two parts. (1) It checks that input streams consist of well-formed XML. For instance, it checks that pairs of tags match to form XML elements. (2) It checks for validity against received DTD’s. Once parsed, an XML document forms a DOM (Document Object Model) tree, which can be edited, or otherwise manipulated. The DOM model is represented through a Java API which affords the accessor and assignment (“get” and “set”) methods that correspond to those specified in DOM [DOM]. Several off-the-shelf XML parsers generators exist (the most notable being IBM’s XML4J [XML4J]). Note that this layer’s purpose is to parse/generate the purely declarative portion of the AVR framework: schema classes and instances. It does not deal at all with the executable bindings. It is generic and not specialized to interpret any specific DTD of XML (e.g. XMI).
The XMI Layer

The XMI layer enforces XMI semantics over the XML parse trees acquired from the XML Layer below. This layer consists of Java classes that do one of two things. (1) They take the DOM tree parsed from an XML input stream and interpret it as a tree of XMI schema classes or declarative instances. This normally occurs when a data integrator is registering a new schema with the AVR. (2) They take a set of schema classes/instances and transform it into a DOM tree for output from the AVR. This normally occurs in response to a browser request. In addition to structuring data according to schema classes and properties, the XMI Layer introduces the property flags and specification strings necessary to derive properties from others. At this layer XMI semantics are enforced. These include, type safety of schema classes and properties, multiplicity of properties, default values, etc. It does not enforce semantics specific to the AVR, like stored/virtual property flags. Like the first layer, the XMI Layer is an interpreter of purely declarative statements.

The AVR Framework Layer

The third layer is the AVR framework engine. It is responsible for interpreting semantics specific to the framework. It interprets property flags and specification strings from the schema classes defined by the data integrator. For example, it stores properties that are flagged as <<stored>>. The AVR Framework Layer also performs aggregation and inversion of properties according to their specification strings. These are declared in the schema classes and association classes. This layer contains the walker for traversing and storing the schema extent. It also calls the appropriate virtual property routines to virtual populate properties lazily when they are queried. It also contains the ECA engine described in the previous section. Rule registration, event queuing, rule triggering, rule queuing, condition evaluation and action firing are all coordinated at this layer. Thus this layer is not purely...
declarative. It is the meeting point between the declarative XMI and executable routines bound to XMI schema classes and ECA rules. The bridge between Java routine stubs and native (non-java) executables occurs at the application layer. This form of integration only occurs with non-Java tools that use direct API’s instead of streams for communication.

In order to evaluate conditions in ECA rules, the AVR Layer requires that the data integrator supply a condition language interpreter. This must be implemented in Java or, if it consists of native code, it must have a Java API so it can be called to interpret specification and condition strings. If no interpreter is supplied, the default condition language is assumed. This is the same path-query language used in specification strings and used in ECA rule examples in the previous section. For conditions, the data integrator may otherwise write non-declarative call-outs to Java routines. However, since they are specific to certain ECA rules, they are considered a part of individual schemata. Schema-specific executable code exists at the Application Layer above.

Although the stored schema extents conform to the XMI data model, they are not stored as XML. XML is too verbose to offer an efficient storage mechanism. Instead, a more compact, proprietary format is used. Instances and schema classes are stored as BLOB’s (binary large objects). The instances’ properties are not spread across relational database tables, as in object-relational repository systems. This is because the cost of joining these tables to reconstitute an object is not tractable for browsing purposes. For communication between AVR’s, this compact format may also be used for rapid transfer of data.

V.1.1.4 The Application Layer

The bottom three layers constitute the AVR framework implementation itself and are not written by the data integrator. The fourth (top-most) layer is the Application Layer. At this layer, a data integrator
writes a set of Java classes to accompany each schema. These provide the property routines and action routines for the declarative XMI instances. These routines are Java methods. Thus, the Application Layer contains the wrappers encapsulating information sources and filters. The wrappers handle direct communication with the tools. They also present a uniform framework interface to the mediator so it can integrate them. This interface abstracts away idiosyncrasies of the tools. The mediator is mostly declarative, containing few property routines. Any property routines it has do not communicate directly with tools. When present, these routines combine properties from different tools to derive new properties. This layer is a mixture of Java classes and native tools written in other languages. The non-Java code is due to native source applications that are wrapped. These applications can also execute on other machines than the one hosting the AVR. They are normally independently developed legacy tools. However, they must be wrapped in Java classes so their raw data can be converted to XMI and made accessible to their wrapper. The wrapper also contains Java monitors that detect change in the tools. The coordinator receives event notification from the monitors and launches the walker over the stored data requiring updates. The walker then communicates with the mediator to do its traversal. The mediator forwards the walker’s property requests to the wrappers. Then the wrappers gather data from the (potentially native) tools. Piping instance data from sources to filters, from filters to other filters, and finally to clients also occurs at this layer and may occur during traversal. Since the communication between schemata may be proprietary and native, this is appropriate.

V.1.2 The Choice of Java

Java was chosen as the most suitable implementation language for the AVR framework. This language in expresses the framework interface presented by the wrapper schemata. The mediator schema, the walker and the ECA engine use this interface. Java is a high-level, strongly typed, object-oriented,
interpreted, network-centric, and platform-independent language. Once java source code is compiled into executable bytecode, then the bytecode can be loaded into an interpreter implementing a Java virtual machine (JVM). Java interpreters can run, in principle, on arbitrary machine architectures. These range from embedded systems on small, hand-held devices to mainframes. In addition to this, the same Java bytecode can be loaded over a network and run on various machine architectures without requiring recompilation. This mobile code is ideal for downloading client applets into a generic web browser with a Java VM, or writing mobile agent applications.

Java provides a set of platform-independent bytecode libraries with intuitive API’s that afford numerous services. They include libraries for graphical user-interface composition, input/output, network communication, remote method invocation, internationalization, CORBA [CORBA], and many other popular facilities. Third party tool vendors also supply many libraries — each with a high-level API. These may communicate with specialized hardware sensors for communication or data gathering from sensors.

Java was chosen since the AVR framework requires that data integrators write their own applications wrapping information sources and filters. Java was the best choice since it is easy to learn, being a high-level language. The libraries listed above provide API’s for many low-level functions — easing the burden of the wrapping phase. Remote wrappers can be written as network applications easily. Thus data can be gathered, integrated and change propagated over networks easily.

Java is also useful if a data integrator also writes client-side applications to render specific data types. Symmetric code can be written to run on both the client and AVR for marshaling/unmarshaling, interpreting the XMI data streams and generating new declarative instances to pass across the network. For example, if a client-side application allows editing of portions of the XMI instance graph, then
these edits can be sent back to the AVR and interpreted by the same set of Java classes specific to the schema in use.

Finally, the use of dynamic binding in Java reflection is key for the AVR framework. This allows the declarative strings in XMI to be mapped to (1) Java classes to instantiate for declarative instances and (2) methods to invoke to gather properties and perform ECA actions. This would not be possible if C++ had been chosen, although other languages such as Smalltalk and Tcl have this capability.

V.1.3 XMI vs. Java

The interpreter in the XMI Layer consists of several parts. The first is an API, which describes a subset of the XMI data model. The API consists of a small number of Java interfaces. These allow a set of XMI schema class objects and declarative instance objects to be directly manipulated by the AVR. Declarative instantiation, specialization and property arcs are represented as references between Java objects. Java references are equivalent to pointers, but are represented more abstractly in the syntax.

There is an important issue when implementing a repository with a declarative, object-oriented data model. This issue also occurs with richer data models such Telos [Koubarakis 96]. Object-oriented executable languages instantiate and specialize programming language classes. A designer may be tempted to use programming language instantiation and specialization to represent declarative instantiation and specialization. This choice may cause problems as executable language semantics are determined by the needs of the language’s compilers and runtime environment. The declarative language’s semantics only concern pure modeling issues. Thus, the executable language’s compiler design is determined by different issues than declarative modeling.

Java has two levels of instantiation: a class level and an object level. It has single instantiation: an
object can only be an instance of one Java class. It also has single inheritance of behaviour and fields.

In order to implement the declarative XMI data model in Java, a designer should not distort or compromise the model because of limitations of the executing language. In the AVR implementation, XMI inheritance and instantiation are orthogonal to Java inheritance and instantiation as illustrated in Figure 17 on page 21. This is achieved by representing both schema classes and declarative instances as Java objects. In addition, instances reference the schema class object they declaratively instantiate. Schema class objects reference the instance objects in their extent and the superclasses they declaratively specialize. Inheritance is interpreted at runtime by collecting the properties of superclasses and making these properties available to declarative subclasses.

A data integrator uses Java classes with accessor ("get") methods to represent instances with properties. The return types of method signatures in Java do not change over inheritance hierarchies. This prevents the narrowing of the return types of accessor methods with the same name. A method in Java represents a contract to invokers of that method. Once made public, this contract cannot be altered without producing semantic errors in the Java compiler. For example, suppose that the Java class Student has a getCourses() method returning a Course[] array (indicating the courses the student is taking). The corresponding method in the subclass GraduateStudent actually returns a GraduateCourse[] array but must return a Course[] in its method signature. This limits the modeling capability of Java, but fulfills the notion of a contract.

However, such contract should not exist among schema classes. Properties of schema classes can be narrowed as they are redeclared lower in schema class hierarchies. For example, suppose XMI schema class GraduateStudent has a property courses that takes GraduateCourse as its value. Schema class NightSchoolStudent declares courses to take NightCourse as its value. GraduateNightStudent is a subclass of both GraduateStudent and
NightSchoolStudent and declares courses to take GraduateNightCourse as its value. When NightSchoolStudent is added to the schema, an XMI interpreter checks that the value of its courses is both a subclass of GraduateCourse and a subclass of NightCourse (which it is). If GraduateNightStudent's courses property took Course as its value, then GraduateNightStudent would be rejected by the XMI interpreter, since Course is a superclass, not a subclass of GraduateCourse and NightCourse. These semantics cannot be expressed using Java method signatures. However it makes sense to be able to narrow the type of a property when that property is redeclared in subclasses. In the design of the AVR implementation, Java and XMI instantiation and specialization were made orthogonal. This avoids the limitations that Java would place on the modeling capabilities of the AVR.

V.1.4 Routine Bindings in the AVR Framework Layer

The AVR Framework Layer uses the API in the XMI Layer below it to perform the necessary operations for an AVR interpreter. The AVR Framework Layer interprets the various property flags, specification strings, association classes, and ECA rules. It enforces AVR-specific constraints (not enforced by generic XMI) on any user-defined schemata and schema extents.

However, this layer also provides a framework for binding properties and action classes (in ECA rules) to Java methods. These are the bindings to executable routines. The binding generally uses a feature of the Java language called Java Reflection [Reflection]. This facility allows the dynamic loading and instantiation of Java classes given only strings that represent the name of the class. These strings come from the XMI schema classes and properties. A mapping is also produced between a property name and a Java method name. In order to populate a property with instance data at runtime, the method can be invoked by reflection. The method then communicates with a wrapped tool, that provides data in the
tool's stream format. This is then converted into XMI instances in the Java method. These instances then become the value of the specific property. Java reflection provides the bridge between declarative XMI and executable extension in the AVR framework.

V.1.5 Constructing Objects Representing Instance and Schema Classes

Java objects representing XMI instances and schema classes are constructed as required by the AVR's layered architecture. The objects representing schema classes can be constructed from a single Java class representing generic schema classes. This class implements the DOM routines from the XML Layer. This is defined as a set of Java interfaces by W3C [DOM]. The Java class also implements a set of accessor/assignment methods from interfaces defined for XMI schema classes at the XMI Layer. Finally, the Java class implements AVR framework-specific methods. These methods are schema-invariant so they correspond to the schema class semantics at the AVR Framework Layer. The Java class does not implement any schema-specific methods so there are no methods corresponding to the Application Layer.

The schema-specific executable bindings occur in the Java classes representing declarative instances. When a property is populated, it is the instance object's methods that are invoked for each property. The Java classes for XMI instances are thus schema-specific, so they add methods beyond the XMI requirements. Thus they are written by the data integrator. However, like the Java class representing schema classes, the instance Java classes also implement the accessor/assignment methods of the DOM API for the XML Layer, the XMI API for the XMI Layer, the framework-specific methods for the AVR Framework Layer. In order to conform to these interfaces, a base Java class for instances can be extended by wrapper schema implementers. This class implements the interfaces for the bottom
three layers.

When an XMI instance stream is being parsed, or instances are being otherwise created, the construction of appropriate Java objects is forwarded to a factory. A factory is a programming idiom implemented as an object. Its responsibility is to create other objects. The factory design pattern is described in [Gamma et al.]. There is a factory supplied by the data integrator for each wrapper schema. The kinds of instance objects to create are indicated by a string passed to the factory. The string is acquired from the wrapper schema’s class which the new instance declaratively instantiates. It is important for the AVR framework to not monopolize object construction, but instead, to forward this responsibility to each schema’s factory. This way the property and action routines specific to the instance’s schema class can be directly associated with the instance object. Note that during a walker’s traversal, the factory also creates new instances.

V.2 Issues Facing AVR Application Developers

V.2.1 URL’s and Semantic Keys

When a browser request for a particular instance datum arrives at the AVR, the URL in the HTTP request will contain a CGI query string identifying the instance. The AVR must then retrieve or create this instance and generate a stream representing it for the browser. First, the CGI query string must be parsed to identify the instance desired. These query strings are normally composed of pairs of argument names and values. The pairs are separated by ampersands (“&”) and the names and values are separated by an “=” (e.g. argument0=value0&argument1=value1&argument2=value2&...). For an AVR this
string should contain:

1. The schema class to be instantiated,

2. A description of a containment path from a root seed instance datum (e.g. a source file directory) to the contained instance (e.g. a local variable in a nested procedure in an unnested procedure in a file in that directory).

The path in (2.) may be a single string that is interpreted by the mediator schema (e.g. “C:/MyDir/MyFile.MyProc$MyNestedProc.myLocalVar”). It could also be a sequence of argument/value pairs for each of the parent types. In the case where the instance is not stored persistently (i.e. it is virtual), the AVR must produce the instance corresponding to this string. However, in order to do this, the AVR may also have to create some of that instance’s containers. For example, if the instance of the nested Procedure that contains myLocalVar is not already populated, it will need to be for the query’s duration. This rule carries up to the root instance in the containment hierarchy. In this way, virtual schemata afford random access to their instances despite being populated lazily and forgetfully. It is the responsibility of data integrators to make their virtual schemata respond to queries in this way – providing this random access. On the other hand, a non-virtual schema is only required to initiate the walker’s traversal to prepopulate the schema’s stored extent.

When the response stream for a browser is being generated, an instance object must generate a URL for each of its non-primitive property values. These represent all the adjacent instance nodes in the graph. Such a URL could be generated from the semantic key discussed earlier (in the section on page 56). This way, when a URL arrives at the AVR, its corresponding semantic key can be regenerated, and a lookup performed against the dictionary for the indicated schema class. If there is a schema extent for this class, then an instance should have already been populated by the walker. Otherwise the schema class is virtual and a new instance needs to be created for the duration of the query. A Java
constructor should be available so that the semantic key can be used as an argument to directly
generate the instance. The data integrator must meet these requirements.

V.2.2 Reflection vs. Dictionaries

V.2.2.1 Factory implementations

Each schema supplies a factory for producing new Java objects for declarative instances. This factory
must be implemented by the data integrator, and this can be done in a number of ways. The most
general way is to construct an instance of the indicated class by Java reflection. There is a default
instance factory supplied with the AVR framework that constructs instances this way (since it is the
most general). The factory implements an InstanceFactory interface containing a method
newInstance(String schemaClassName, String semanticKey). The first
argument names the schema class to instantiate. This is mapped to a corresponding Java class indicated
in the schema class’s declaration. Several schema classes may map to one Java class. The second
argument is the semantic key for the instance. When the key is parsed from a URL’s CGI query string,
the instance can be looked up in the extent (if any). If the lookup fails, a new instance object is
constructed from the factory. In order for the default factory to be used, all Java classes representing
instances must implement a public constructor. This accepts a schema class name and a semantic
key as arguments. The factory performs a lookup in the schema to find the Java class to construct by
reflection since they need not correspond one-to-one. Then the constructor is passed the schema class
name and semantic key and invoked reflectively.

Java reflection facilities such as reflective constructor invocation are not implemented efficiently on
Java virtual machines (JVM’s). Suppose many objects are to be created as in the traversal of a walker
over a large extent. Here it is recommended that objects be created by other means in the factory. An alternative method is to have the factory contain a prototype instance for each Java class to be instantiated. The merit of using a prototype-based factory is mainly its efficiency. This prototype can be placed in a lookup table keyed on the schema class string passed to the factory. Thus, the correct Java class is found for the schema class string. The prototype is acquired during lookup. Then it is cloned by invoking a polymorphic instance method. The cloned instance is also passed the semantic key so it can represent the instance indicated by the factory's `newInstance(String, String)` second argument. The factory can then return the clone. It is important that each prototype be capable of producing clones initialized in the correct state that corresponds to the semantic key. Normally this state is only enough to reproduce the same key on demand from the new instance object. Properties pointing to parents in the containment hierarchy will also be initialized. However, most other properties will be either unpopulated or only populated with default values. The invocation of instance methods is much faster on most JVM's than is reflective invocation of constructors. The latter are non-polymorphic class-level methods.

V.2.2.2 Property routines

The data integrator faces similar issues in the invocation of property routines as in object construction. Property routines can be invoked two different ways. Suppose an AVR is generating an output stream rendering an instance. It needs to iterate over the properties of the schema class and invoke the corresponding property routines on the instance object. Similarly, suppose a walker is generating the stored property values during a traversal. It iterates over the schema class's list of property names flagged as `<stored>`. The schema class supplies the property name as a string and the corresponding links or primitive values need to be returned. Instance objects always implement a Java
instance method selectProperty(String). This returns the property value for the given property name. The property method string passed to selectProperty(String) is supplied by the schema class. The invocation of the property method in the body of selectProperty is done by reflective method invocation in the default implementation. This implementation is supplied by an abstract superclass in the framework. Alternatively, the method can be overridden to contain a series of if...then...else statements that try to match the string. For instance:

```java
public Object selectProperty(String methodName) {
    if (propertyName.equals("calls")) {
        return calls();
    } else if (propertyName.equals("globalVarsAccessed")) {
        return globalVarsAccessed();
    } else if ...  // other cases
    return super.selectProperty(propertyName);
}
```

Even for large numbers of property method names in a schema class, this is likely to be much faster than the reflective approach on most JVM's. Properties are populated in bulk by a walker. In addition, properties are populated frequently during browser requests. So the selection of property routines for invocation can become a performance bottleneck.

### V.2.3 Multiple inheritance, Object Fusion and Property Routines

In cases where properties are multiply inherited from several wrapper schemata into a single instance object, multiple inheritance is achieved by delegation. Separate corresponding instance objects are created – one in each wrapper. Once this is achieved, the instance object inheriting the properties references these objects. Mediator instance objects that multiply inherit properties may have their own Java class. When such a mediator instance object is queried for a property, its selectProperty method is invoked. This can select which delegated object to forward the property request to. When
two delegate objects can supply the same property, the selectProperty method decides which of these is used. It can also combine the results from both delegate property methods. The delegates are thus "fused" into the mediator object. This makes all their properties, property routines and the underlying tools accessible through the mediator. Mediators register which wrapper schema classes correspond to mediator schema classes. Therefore, when the factory constructs a new instance of a wrapper instance, a corresponding mediator object can be immediately created to coordinate property delegation. This is where the name translation service introduced by the Software Bookshelf [Finnigan et al.] may be useful. Semantic keys may not be identical between wrappers or between wrappers and the mediator.

V.2.3.1 AVR Semantic Checking and Java Reflection

Reflection, despite its poor performance on most JVM's can be put to good use in checking a newly loaded wrapper schema. If the default factory is to be used, this is known by the AVR system at schema-registration time. Thus, all Java classes representing instances can be retrieved from the schema's XMI declaration. The system can then force these classes to be loaded into the virtual machine and it can check for the presence of a public constructor taking two strings. The factory invokes this feature as needed. If this reflective check fails, then the appropriate message is sent to the user interface and the data integrator is alerted of the failure. It is better for this failure to occur at schema registration time than during the normal operations of the AVR (e.g. during a walker's traversal).

Similar checks are possible for property routines. Whether or not property routines are looked up by an if...then...else selector or by reflection, the XMI declaration of the schema indicates what method name is used for each property. It is assumed that there will be a property routine for each property.
name. At schema registration time, the property method names are retrieved from the XMI declaration. Then each Java class for each instance type is searched for its property routines by reflection. Failures to name methods properly can then be checked at registration time rather than later, where consequences may be dire. Again, these checks are more assuring if properties routine invocations are selected by pure reflection. Otherwise name mismatches in the if...then...else selectors may not be detected.

V.2.3.2 Action routines

Action routines are the easiest to implement, as they subclass a Java interface Action which supplies an instance method fire(). As there can be only one such method on an action, no lookup is required (checking is done by the Java compiler). All parameters for the firing of an action are contained in the action object's state. Thus the fire() method is polymorphic and does not take any Java parameters.
Chapter VI : Conclusion

VI.1 Summary

Active Virtual Repositories offer a network-centric alternative to traditional data integration mechanisms. AVR’s are superior to these traditional strategies where data is not already stored in databases nor need it be duplicated in databases. AVR’s are also more appropriate when users wish to examine the integrated data through browsers. This is done by navigating hyperlinks, as opposed to accessing the data through query languages. The schema-driven algorithms used throughout this thesis require a richer data model for encoding metadata than traditional Object-Oriented modeling provides. Specifically, property flags, association classes and property specifications are necessary to drive the control-flow of the AVR algorithms. The walker offers a general mechanism for populating and updating the stored extent of instance data through the integrated mediator schema. Stored data is used to answer context-dependent property queries. This is done without complete duplication of the wrapped tool data in a database with inverted indices.

XMI is declaratively rich, and it is an emerging standard for data interchange. Semantically based interoperability and data integration require the rich descriptive capabilities of languages like XMI. The differences between the data and domain models of information sources and filters, and the dependencies between these tools can thus be expressed in XMI. Rendering this information declaratively has the advantage that a data integrator can browse and debug the wrapper and mediator schemata along with the browsable instance data. The persistent storage and queryability of wrapped tools is exploited so that more of the AVR can be virtual, forwarding to the respective tools. The fact that XMI schemata can generate XML DTD’s allows the transformation (though XSL’s) of this
integrated data for arbitrary tools accepting XML.

This thesis attempts to overcome some of the deficiencies of various XML data/metadata interchange standards communities. The deficiencies of XML and, to a lesser degree XMI, include a lack of support for integrating arbitrary, non-database software tools with proprietary stream formats, domain models and data models. The adoption of XMI is strategic in that it is becoming the acknowledged de facto standard data exchange format between software vendors intending their data to be integrated. Since a framework is provided for the direct instrumentation of tools, the tools can also provide data for integration within the larger interchange schemes being standardized. Moreover, AVR’s can act as lightweight migration strategies to make legacy software tools network-centric.

**VI.2 Future Work**

Plans for future work with the AVR framework include the integration of this system with tools that model design intentions. Modeling frameworks like $i^*$ [Yu] capture the rationale of designers of software systems, information systems, office or manufacturing processes and other worlds. Typically, software modeling frameworks (like UML [Rumbaugh et al.], SADT, etc.) allow the encoding of software structure and function, (the “what” and “how” of software). However, they do not capture the rationale driving design choices (the “why”). Integrating software analysis by automatic tools (as outlined in the AVR framework) with high-level design information from intentional models as in $i^*$ can enrich a developer’s picture of a software system.

In addition, the use of association classes can allow the elaboration of the design rationale of wrapper and mediator schemata themselves. Developing wrappers in XMI and Java is non-trivial. Capturing the rationale of a model eases the burden of a new user browsing a complex data integration system. So in
addition to modeling world semantics, a hybrid AVR-\textit{i*} system can allow a greater level of self-documentation in active XMI models.

Ongoing research at the University of Toronto is exploring the idea of capturing rationale in software architecture. Software architecture is the high-level aggregate visualization of software implementations and designs. The connections between different large-scale software components is typically visualized in terms of functional characteristics (the tasks they perform). However they are rarely visualized in terms of the strategic dependencies and vulnerabilities these components entail. Strategic dependencies are those that treat the dependers and dependees as actors in a larger process. The rationales of software architects can be encoded in terms of these dependencies and thereby captured. By integrating software tools that aggregate data about systems under study with rationale information, again a richer picture of software architecture emerges. The AVR framework allows this changing, higher-level derived information to be integrated with design rationale.
Glossary

Note that glossary terms are either those introduced in this thesis, or are common terms that have been given particular meanings in this thesis. Only terms, which are similar and used throughout the thesis, are defined here to avoid confusion.

- **Abstract class** — a schema class that cannot be instantiated. Only subclasses of the abstract class (if not abstract themselves) can be instantiated.

- **Action** — the part of an *ECA rule* that is active since it contains an executable extension (its *action routine*). It performs some transaction on the repository or provides some service in a cooperative information system. Actions are used in AVR’s to perform change propagation.

- **Action routine** — the executable extension for the action part of an *ECA rule*. An action is fired when the condition of the rule evaluates to true.

- **Active metamodel** — a metamodel (or schema) with executable extensions that assist in populating entity properties and encoding the *ECA rule actions* executed during change propagation.

- **Aggregated properties** — derived properties that are derived by taking the union of other association ends (or sum of other attributes). For example, a people property may be computed by taking the union of the values of men and women properties.

- **Association** — in XMI or UML, an *n*-ary relation consisting of *n* association ends. Association ends are distinct from attributes in that they connect *n* schema classes (not primitives). Associations are instantiated by links.

- **Association classes** — in XMI, UML, an entity that adds extra metadata to an association. In AVR’s this can drive metadata-driven algorithms about associations.

- **Association end or end** — in XMI, or UML, one of the *n* ends of an *n*-ary association (relation). It is one of the two types of properties discussed in this thesis.

- **Attribute** — in XMI or UML a property with a primitive value type. Not to be confused with an XML attribute.

- **Class** — see schema class.

- **Composite (or composition) association ends (or composite properties)** — properties that indicate strong containment. When the container (the owner the link) is copied, moved or deleted, the contained instances (the values of such links) are copied, moved or deleted as well. These properties can thus be used to decompose a schema extent graph into a tree. These are used in the walker’s tree traversal algorithm to populate the stored property links of this tree. This way it also populates the schema extent.

- **Condition** — the part of an *ECA rule* that evaluates whether the action portion should be fired. If the condition evaluates to true, the action is fired. The condition must be encoded in the appropriate declarative language. Alternatively, it may call a routine that evaluates to a Boolean.
- **Condition evaluation** — the evaluation, by the Coordinator, of a condition in a triggered ECA rule.

- **Context-dependent property queries** — A query for a property value in an AVR that requires the pre-computation of a predetermined context of instance data. Such queries can not be answered through lazy evaluation. In an AVR re-documenting a software system, the context may be a compilation unit, a subsystem, component, the entire software system or a product line. When the context is the entire contents of the AVR, the query is domain-dependent in the traditional sense (e.g. “Who is not John?”). Most queried properties will be virtual and will be calculated by inverting a stored, context-independent property, which has been stored in the schema extent.

- **Context-independent property queries** — A query for a property value in an AVR that does not require the pre-computation of a predetermined context of instance data. For example “What is John’s name?” as opposed to “Who is not John?” which is context-dependent. This property’s links are often stored in the schema extent so they can be inverted to populate virtual, context-dependent properties.

- **Coordinator** — Each AVR has a single Coordinator which processes ECA rules. When an event is received from a monitor, the Coordinator finds all rules listening for this class of event. Then it marks these rules as triggered. Triggered rules’ conditions are evaluated against the bound variables in the event received (or globally visible/computable data). When a condition evaluates to true the ECA rule’s action is fired.

- **Data dependencies (or dependency)** — Dependencies that information filters (secondary information sources) have on primary information sources (or other filters). These dependencies exist because a filter derives data from the source(s) it depends on. In an AVR, a derived property in one schema class may derive its data (and thus depend) on the property of another class(es) in a different schema. It thus can be seen as a dependency between the two schemata.

- **Declarative** — applied to classes, or instances, this is to distinguish the classes or instances from those in an executable object-oriented programming language. Declarative classes or instances are created in declarative languages. Thus, one can refer to declarative instantiation between declarative instances and classes. This does not imply an object-oriented implementation.

- **Declarative languages** — formal languages that do not consist of sequences of executed actions. Instead, they describe semantically formal structures. For example, XMI is a declarative language. Non-declarative languages include procedural and object-oriented programming languages.

- **Dependency** — see data dependency.

- **Derived property** — a property whose links are populated by derived property routines — that is, they are populated from other properties.

- **Derived property routines** — property routines that derive property values from other properties.

- **End** — see Association end.

- **Event-Condition-Action (ECA) Rules** — Declarative rules coordinating change propagation in active databases or AVR’s. They consist of events, which trigger the rules in a Coordinator. Rule conditions are evaluated based on the data presented in the event (or global data). If the condition
evaluates to true the rule's action is fired.

- Event processing – when an event triggers a set of ECA rules in the Coordinator, event processing begins. When the last rule's condition is evaluated, event processing stops. Event processing does not depend on when or whether actions are fired. It only concerns condition evaluation for triggered rules.

- Executable extensions – non-declarative executable routines that extend the declarative portion of an active metamodel in an AVR.

- Extends – a schema class extends another if it is a subclass of it. A schema $B$ extends schema $A$ if $B$ contains a subclass of a class in $A$ or if $B$ contains a class with an association end that references a class in $A$.

- Extent – see schema extent.

- Filter – see information filter.

- Fire an action – an ECA rule's action is fired when its condition evaluates to true during rule processing by the Coordinator.

- Forgetful – the virtual portion of the AVR is forgetful in that it does not retain the instances necessary to answer a browser query after that query has been answered.

- Information filter, (or filter) – see Secondary information source.

- Information source – An application generating information for an information service. They can be primary information sources, secondary information sources, (or information filters).

- Instance – an entity that instantiates a schema class. It is either a member of the schema class's stored extent, or it exists temporarily during a query to the AVR. In an AVR, the extent of a schema class may never be computed and stored, but neighbouring instances are reachable by querying their properties triggering property routines.

- Instance property – a property whose values vary among instances within a schema extent. It is the opposite of a static property. Most properties are instance properties.

- Instantiation – The relation that holds between an instance and its schema class.

- Inverse association ends – association ends that form the two directions of a binary association (relation). Thus, they are inverse of each other.

- Inverting an association end – This is done by taking all the links instantiating an association end (in a binary association) and making the sources of those links the values of the inverse association end. In the AVR framework, this is done by a derived property routine.

- Link – in XMI or UML, an instance of an n-ary association that connects n instances.

- Listening – an ECA rule is listening if it has been registered with in the Coordinator and is now waiting for the Coordinator to be notified of event instances of a certain class. It is listening for
events of this type.

- Mediator (or mediator metamodel) – an active metamodel which makes several wrapper schemata browsable as one schema. At the instance level it makes the corresponding schema extents (schema instances) browsable as one instance graph.

- Mediator schema – the schema portion of a mediator metamodel.

- Metadata schema – see schema.

- Metamodel – in the AVR framework this consists of a schema and its schema extent.

- Monitor – each wrapper metamodel has a monitor to generate events when the state of the information source being encapsulated changes. This can be done lazily or by polling, etc. The events notify the Coordinator.

- Non-declarative languages – the opposite of declarative languages (see above).

- Notify – a monitor notifies the AVR’s Coordinator when its wrapped information source changes state. The notification consists of sending the Coordinator an event instance encoding the detected change.

- Package (or XMI package) – a schema in XMI or UML.

- Primary data – data from a primary information source.

- Primary information source – An information source that does not derive its data from another information source.

- Primitive – In XMI or UML, a number, Boolean or enumerated type (e.g. a colour) which has no properties. It can be the value type of an attribute. This is as distinct from a schema class which is a non-primitive, which can have properties and which is occurs at one of the association ends of an association.

- Property – an XMI or UML association end or attribute.

- Property routine – an executable extension used to populate a property. There are two types: Tool property routines and derived property routines.

- Raw data – instance data from an information source in its native form. This is translated in the wrapper wrapping that information source.

- Rule – see ECA rule.

- Schema (or metadata schema) (pl. schemata) – A collection of schema classes that model a particular world or perspective on a world.

- Schema class – An entity representing a collection of instances with common properties. Classes can inherit properties from superclasses.

- Schema extent (or schema instance) – The collection of instances that instantiate the schema
classes in a schema. In an AVR, this is not necessarily known, since part or all of a schema may be virtual. Not all classes and properties in a schema need to be pre-populated aggressively by the walker into an extent. Virtual properties can be computed lazily at browse-time. Because of this, the extent of a schema is a partial instantiation of its classes and properties. Any instance data in the extent has been generated from seed instance data.

- **Schema instance** — see schema extent.

- **Seed instance datum (or seed)** — an instance supplied by the data integrator. The walker calculates schema extents using these data as the roots of the respective instance data trees (which comprise the schema extents).

- **Semantic key** — a key to lookup whether an instance has been populated in a wrapper metamodel. It is part of that metamodel's namespace. It is semantic in that it is based on the world modeled by that wrapper (as opposed to enumerating identifiers automatically). Corresponding instances may have differing semantic keys in different wrappers. A name translation service [Finnigan et al.] is needed to map corresponding names in different namespaces.

- **Static property** — a property whose value is determined as a constant at the schema level. Thus, it is invariant for all instances in that schema's extent. It is the opposite of an instance property. Most properties are instance properties.

- **Stored property** — a property whose links (ie. instances), once computed are stored persistently in the repository. It is the opposite of a dynamically generated virtual property.

- **Tool property** — a property that is computed by a tool property routine.

- **Tool property routines** — initiate actions that gather information from a tool (ie. an information source) and thereby populate or update properties.

- **Triggered rules** — ECA rules that are listening for an event class, where the Coordinator from a monitor has just received a corresponding event instance. These rules are now evaluated in order according to rule priority policy used in the AVR.

- **Virtual** — The virtual portion of an AVR is the collection of instances and properties that are generated to answer each browser query but are not cached or stored persistently. Thus, that part of the repository simulates the extent of schema class instances while not storing them actually.

- **Virtual property** — a schema class property whose links are not stored with the owning instance (if the instance is stored). Instead, virtual properties are generated dynamically at browse-time. It is the opposite of a stored property.

- **Virtual wrapper schema (or virtual wrapper)** — a wrapper schema composed of classes whose instances and links (property instances) can be populated at browse-time. They do not require precomputation.

- **Walker** — a part of the AVR that pre-computes and conducts updates on schema extents. It uses composite properties to decompose the extent graph into a tree. Then it traverses this tree, populating stored properties and recursing on composite properties. The roots of the trees are seed instance data supplied by the data integrator.
- **Wrapper** (or **wrapper metamodel**) – an active metamodel that encapsulates an information source so that its idiosyncrasies are reconciled and concealed from the **mediator**.

- **Wrapper extent, wrapper instance** – see **schema extent**.

- **Wrapper schema** – the **schema** portion of a wrapper metamodel.
References

[Anquetile/Lethbridge]

[Bergstra/Klint]
J.A. Bergstra, P. Klint. "The discrete time ToolBus.”
http://adam.wins.uva.nl/~olivierp/toolbus/index.html

[Chaudhuri/Dayal]

[Chawathe et al.]

[CGI]

[Chen]

[Connolly]

[CORBA]

[DOM]
http://www.w3c.org/TR/REC-DOM-Level-1

[Finnigan et al.]

[Gal]

[Gamma et al.]
E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1994.

[Gosling et al.]

[Holt]

[HTML]

[HTTP]

[IBM]

[Katz]


[Koubarakis et al.]

[McCabe]

[Muller/Klashinsky]

[Mylopoulos et al. 90]

[Mylopoulos et al. 96]

[Papakonstantinou et al.]

[RDF]

[Reflection]
Java Core Reflection. Sun Microsystems Inc. 1997. Introduced with Java 1.1.

[Rumbaugh et al.]

[Schreiber]

[Sheth/Larson]

[VAJ]
http://www.software.ibm.com/ad/vajava/

[Widom/Ceri]

[Widom]

[Wiederhold]

[XMI]
XML Metadata Interchange (XMI). Proposal to the OMG OA&D TF RFP 3: Stream-based Model

[XML4J]

[Yu]