A Generic Integration Architecture for Cooperative Information Systems

John Mylopoulos Avigdor Gal Kostas Kontogiannis
Martin Stanley
Department of Computer Science
University of Toronto

Abstract

Cooperative information systems consist of existing legacy systems integrated in terms of a generic architecture which supports data integration and coordination among the integrated components. This paper presents a proposal for a generic integration architecture named CoopWARE. The architecture is presented in terms of the mechanisms it provides for data integration, and coordination. Data integration is supported by an information repository with an extensible schema, while coordination is facilitated by a rule set and an event-driven rule execution mechanism. In addition, the paper describes implementation and application experiences for the architecture in the context of a 3-year software engineering project.

keywords: *cooperative information systems*

1 Introduction

Traditionally, information systems have been defined as software systems consisting of databases, application programs and user interfaces. However, current trends in business organizations point to a paradigm shift in organizational structures, away from traditional, task-based forms and towards goal- or customer-oriented processes [16]. These trends are forcing a new view of information systems, hereby referred to as *cooperative information systems*. According to this view, the effectiveness of an information system within an organization is not determined solely by the quality of its components, which may have been developed independently and may pre-date the cooperative information system, but also (and perhaps most importantly) by the architecture through which the system is integrated with the rest of the organization so that it can contribute to global business processes and organizational objectives. A cooperative information system consists of legacy systems, workflows and other software components developed independently and changing over time. All these are integrated through an architecture so that the overall system continuously provides useful information services to the organization within which it functions.

Such *integration architectures* are being explored in several different areas of computer science, including Databases where the emphasis is on interoperation and data integration [19], Software Engineering where tool and environment integration issues dominate [5], AI where systems consisting of distributed intelligent agents are being developed and explored [3] and of course, Information Systems [23].

This paper proposes *CoopWARE*, a generic integration architecture, based on active database technology. It presents a natural, flexible and easy to set mechanism for supporting cooperative information systems. In particular, CoopWARE supports a cooperation with different levels of data integration, which results in a highly flexible tool for emerging cooperation among information systems. Our reported experiences are based primarily on a project whose aim has been to integrate a variety of reverse engineering tools in support of reverse engineering tasks [31].

2 The CoopWARE Integration Architecture

In a nutshell, an *integration architecture* provides a generic framework for *information exchange* and *coordination* among a variety of existing software systems. A primary requirement for the integration architecture is that it provides a set of information services which may not have been anticipated during development of any of the component software systems. Moreover, these services evolve over time to meet changing

0-8186-7505-5/96 $5.00

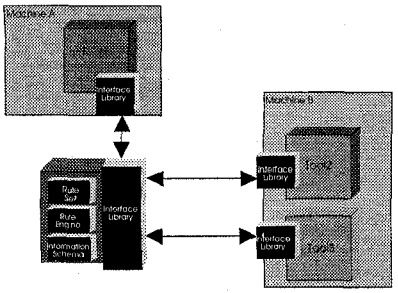

Figure 1. The Integration Architecture

business requirements. 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. Two primary design goals for the integration architecture proposed here are modularity and the ability to operate either on a single host or over a network. Modularity was adopted because it facilitates customization (or even replacement) of implementation components as needed. For instance, the underlying message transport software (in our case mbus, see below) can be easily changed, if a more desirable system is found.

These requirements are addressed in the *CoopWARE Integration Architecture* through a centralized architecture which consists of an *information repository* for representing and maintaining a variety of information generated or used by any of the integrated components, also a *coordination mechanism* through which the integrated components can participate and contribute in a coordinated fashion to global information services provided by the cooperative information system.

Figure 1 presents the general structure of CoopWARE. A particular instance of CoopWARE consists of a *coordinator* which contains an *information schema* based on an extensible information model,[1] a *coordination rule set*, a rule execution engine, also a collection of *interfaces*, one for each integrated software component and one for the coordinator. Each interface defines a set of *services* which can be executed by the component associated with the interface, also a set of *events* through which the component signals the beginning or end of execution of an internal process. One set of services common to all interfaces provides facilities for message passing among components.

As a concrete example of the integration architecture, suppose that we wish to integrate two reverse engineering tools, Refine and Rigi. Refine is a commercial code analysis tool [21] which provides facilities for parsing the code being reverse engineered to generate abstract syntax trees, as well as a high level language for specifying patterns to be matched against abstract syntax trees. Such patterns can be defined to support the analysis of code by determining, for example, the call- or use-graph structure of a given piece of code. Rigi, on the other hand, is a research prototype [25] whose primary function is to assist in program understanding by offering useful visualizations of code structure. Rigi also offers a set of code analysis functions. One would ideally want to use either Refine or Rigi to analyze code and then display the results of the analysis using Rigi. Hence, there is a need to integrate the two tools in support of a reverse engineering activity.

In integrating a new component into the environment, there are three steps that need to be taken, possibly in an iterative fashion, namely *data integration, definition of services, events and rules* and finally *registration.*

Data Integration determines the information needs of each new component to be integrated and updates the information schema as appropriate. In the case of the reverse engineering project, this step determines what information is input or produced by each tool. For instance, both Rigi and Refine operate on files, but treat them differently and associate to file objects different information. If there already exists a schema integrating k components, data integration of the (k+1)st component involves extending this schema to accommodate the information used by the (k+1)st tool. To properly support the data integration activity, an information model needs to be extensible and to provide for annotations of attributes, so that one can note which component is using which attribute.

Services, Events and Rules: This step determines what services will be offer by the new component to the cooperative information system, also what internal events it will report on. In addition, this step determines what coordination needs to exist between the newly integrated component and others. This is accomplished through event-condition-action rules which trigger certain rules when certain events-condition pairs occur.

[1] We use the term *information model* in the sense that *data model* is used in databases, i.e. an information model that provides a set of data structures and associated operations and constraints for representing information.

Rigi services:
Parse: ⟨*Source-Code*⟩
SG-Analysis: ⟨*Rigi-AST, Grouped-Components*⟩
Rigi-Upload: ⟨*KB-Name, Query*⟩
Rigi-Download: ⟨*KB-Name, Set-of-Tuples*⟩
Rigi events:
Rigi-Parses: ⟨*Source-Code*⟩
AST-updated: ⟨*Rigi-AST*⟩

Refine services:
Parse: ⟨*Source-Code*⟩
CBC-Analysis: ⟨*Refine-AST*⟩
Refine-Upload: ⟨*KB-Name, Query*⟩
Refine-Download: ⟨*KB-Name, Set-of-Tuples*⟩
Refine events:
Grouped-Components-Updated:
⟨*Grouped-Components*⟩

Figure 2. The services, events and messages of the case study

Figure 2 presents an example of some of the services and events that might be declared for Rigi and Refine. Each service has a name, possibly followed by a set of parameters, some of which may be part of the information schema. For instance, *Parse* is a service offered by both tools and takes source code to generate an abstract syntax tree. *SG-Analysis* performs a code structure analysis by taking as input an abstract syntax tree and generating a graph. *CBC-Analysis* is a Refine service that performs a clone-based-clustering analysis on code data, measuring a number of metrics on code fragments and then clustering the code fragments according to the metric values returned. Both components have upload and download operations for retrieving and updating data. Services are activated through messages. In general, the coordinator needs to maintain a mapping from the service logical name to the physical procedure which carries out this service within a particular component. For example, Refine can translate a Parse operation to a command as given in an Emacs environment. Each such service execution results in a success/failure message from the tool to the coordinator.

Events represent occurrences of some phenomenon happening within a tool. For example, *Rigi-Parse* informs the world each time Rigi performs a parse operation. This event has as associated information a pointer to the source code that is being parsed. Rigi's *AST-updated* informs the world that an abstract syntax tree has been updated. The event *Grouped-Components-Updated* notifies the world of a change in Refine's grouped components.

Registration make use of an existing interface library to provide a message-based communication interface between a new component and the coordinator, (...and through the coordinator, communication with any other integrated component). How each component actually connects to the architecture depends on what kind of access to its internals is available. We support a range of methodologies from *black box* to *white box* [30] and various shades of gray in between. For example, given that Refine runs on top of LISP, we may want to treat Refine as a black box with which we communicate through an external mediator program. In the case of Rigi, on the other hand, where we have full access to its source code, we may be able to integrate directly. From the point of view of the coordinator and the integration architecture in general, both tool interfaces operate identically.

Once the communication architecture is in place, the new component then registers with the coordinator. This is done dynamically and becomes effective immediately. Thus, components can be added to the architecture at any time and they can also modify their capabilities at will. As each component registers its capabilities with the coordinator, the capabilities of the cooperative information system grow.

The architecture can evolve both because of changes in the registration profile of particular components or because of changes in the coordinator's information schema. For example, the notion of *Request-Message* may be added to the schema so that the coordinator can deal with a new type of messages. For the current implementation, there are a number of generic message types and events, e.g. *Request* and *Response* messages, and *Message-Delivered* events. If one wanted to define a new, special purpose message type, it is a simple matter to specialize the *Message* class, adding whatever message handling is required for the new message type through coordinator rules.

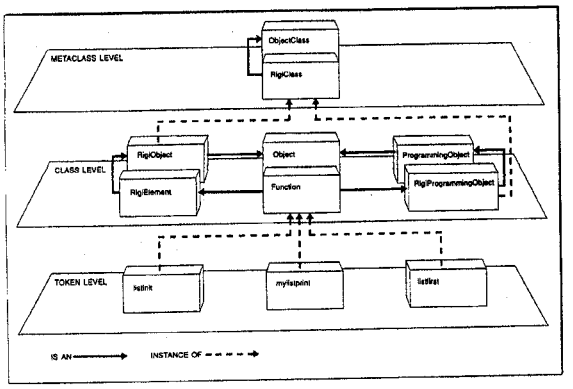

Figure 3. The three layers of the repository schema

3 The Information Repository

An *information repository* is a computer-based system which stores, accesses and manages information about information sources and/or information processes. These information sources may be computer-based files, databases and other data structures, while information processes may be application programs or other procedures which operate on information sources to update or retrieve information. An information repository is different from a database in that, like data dictionaries and data warehouses, it stores information *about computerized information*, rather than information about some application external to computers.

The repository's main function is to facilitate data integration among the various components of the architecture. As such, it needs to provide an information model that is expressive, extensible and efficient. It needs to be expressive so that it can model the data sharing needs of the components easily and understandably; it needs to be extensible so that it can accommodate run-time changes (in particular it needs to support dynamic schema evolution). The information model needs to be supported with an efficient implementation so it does not become a bottleneck.

The information model adopted for CoopWARE is based on Telos [26]. Its features include an object-oriented framework which supports generalization (including support for multiple inheritance), classification (including support for multiple instantiation) and attribution (with multi-valued attributes), a general meta-modelling facility whereby classes are objects too and are instances of metaclasses and a novel treatment of attributes including multiple inheritance and instantiation of attributes and attribute classes. This treatment of attributes provides a powerful mechanism for defining multiple overlapping views of an information object. This allows the repository to have a consistent and singular representation of information that is shared among the components in the integration architecture and at the same time allow for each component's unique perspective on that information.

Figure 3 presents a schema to facilitate information sharing among the integrated tools to share data for the reverse engineering project. It consists of three layers. The top layer exploits the meta-modelling facilities of Telos to define: (i) the types of attribute values that the repository will support; and (ii) useful groupings of attributes for purposes of distinguishing information that is pertinent to each of the individual tools. The use of this layer facilitates schema evolution and provides a filtering mechanism by which each tool can access its own view of a common data object. The middle layer defines the repository schema with the aid of the metaclasses and attribute definitions from the top layer. The bottom layer stores the actual data shared among the individual tools.

Figure 4 shows a detailed example of a portion of this global schema. At the metaclass level, the schema includes the metaclass *ObjectClass* which has associated attribute metaclasses *singleValue*, *setValue*, and *sequenceValue*. All instances of this metaclass can have associated attributes which are classified under one (or more) of these categories, depending on the range of values intended. *ObjectClass* has two specializations, *RigiClass* and *RefineClass*. The former has as instances classes that are manipulated by the Rigi tool, while the latter has instances that are classes manipulated by the Refine tool. As indicated in Figure 4, these metaclasses have attribute metaclasses which identify attributes used by Rigi and ones used by Refine.

At the class level, the class *File* is defined as an instance of both *RigiClass* and *RefineClass*. This declares that a file object may be shared (and manipulated) both by Rigi and Refine. *File* has many attributes classified under one or more attribute metaclasses. Each of these attribute metaclasses define capabilities (attributes) specific to their respective tool. For example, the attribute metaclass *rigiAttribute*, when used in the class *File*, serves to group together all the Rigi attributes. In the Refine case, we have sub-divided its attributes into two classes, mirroring the way Refine represents its data (either as a tree or non-tree attribute). Having done this we are now able to easily and efficiently "filter" objects so that, if desired, each tool gets only the attributes it is interested in.

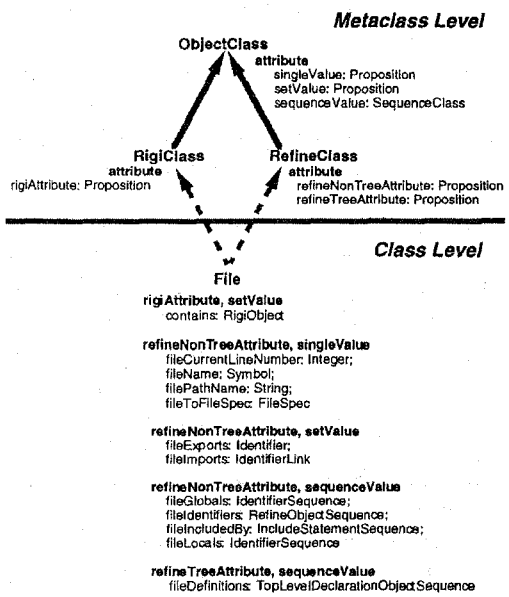

Figure 4. The File class description

4 The Control Rule Set and Rule Execution Engine

The *coordinator* is a set of tools and mechanisms that enables an active monitoring, coordination and cooperation among systems. The coordinator is designed to provide services while enabling the required level of local autonomy. It does not require full data integration, although the more integrated the cooperative information system is, the better the performance of the coordinator would be. The coordinator uses *rules* to define cooperation among information systems. Its intellectual origins resides in the active database research area, and it uses latest advances in this research area [14] to generate a mechanism that maintains the relationships among information systems in a cooperative environment. This section presents the control rule set and its use in the cooperative information system architecture environment (Section 4.1), and the execution engine for rules (Section 4.2).

4.1 Rules

Rules are programming constructs which make it possible to specify the activation of services in response to event occurrences. In accordance with active database conventions [10], each rule consists of three parts, namely an *event*, a *condition* and an *action*. An event is a logic formula that consists of system events

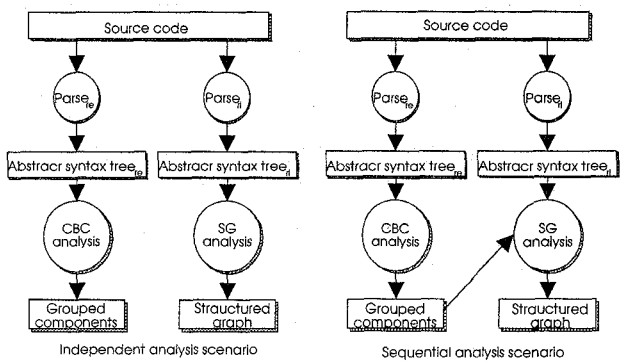

Figure 5. Possible cooperation scenarios between Refine and Rigi

and messages. Events can capture an occurrence external to the system as well as the starting and ending of services. Each rule is associated with one or more event types. A condition is a Boolean formula that consists of data values, constants, etc. An action is a set of services to be activated, each service with the appropriate parameters.

For example, consider again the reverse engineering case study. While each system can perform the parsing and analysis of a source code independently, the combination of the analysis results of both systems provides the user with a better understanding of the source code. Figure 5 presents two possible scenaria of cooperation among the two tools. Data elements are represented using rectangles, and operations are represented using circles. $Parse_{re}$ and *CBC-Analysis* are the operations associated with Refine, while $Parse_{ri}$ and *SG-Analysis* are the operations associated with Rigi. The independent analysis scenario represents a scenario where each tool analyzes and stores its own data. The sequential analysis scenario represents a scenario where Rigi generates a structure graph based on the analysis of Refine and its own. In the latter scenario, the results of *CBC-Analysis* is combined with Rigi's source code representation. Consequently, nodes that are considered clones collapse into a single node in the graph forming modules. Following this scenario, links in the graph represent multiple calls between subsystems to generate a more abstract system decomposition. This scenario can be easily achieved by combining Rigi's and Refine's analysis capabilities, through the use of two rules, as given in Figure 6. Whenever a *Rigi-Parses* event occurs, Refine parses the source code and performs the *CBC-Analysis* service. One result of this analysis is the instantiation of the *Grouped-Components-Updated* event. This event, combined with the *AST-updated* event, activates the *SG-Analysis*.

Rigi	*Event:*	Refine-Grouped-Components updated (Refine-AST) and AST updated (Rigi-AST)
	Condition:	none
	Action:	SG-Analysis (Rigi-AST, Grouped-Components)
Refine	*Event:*	Rigi-Parses (Source-Code)
	Condition:	none
	Action:	$Parse_{re}$ (Source-Code); CBC-Analysis (Refine-AST)

Figure 6. Rules of the case study

The use of the ECA (Event-Condition-Action) model enables us to make use of existing execution models for activating rules. In the case of cooperative information systems, the use of an explicit event clause is essential. Since the event is generated by the information system, while the condition is verified by the coordinator, it is important to differentiate the condition clause from the event clause.

4.2 The Rule Execution Engine

Rules are triggered by events. An event detection results in the evaluation of a condition. If a condition is evaluated to be true, the services in the action part of the rule are activated. If there is no condition for activating a rule, the action part is activated as a direct result of an event detection. For example, the detection of the event *Rigi-Parses* results in the direct activation of the first service $Parse_{re}$.

The rule execution engine in the CoopWARE environment differs from the rule execution engine of an active database. While an active database performs all of the operations of the action part of the rule, the coordinator activates services of the related information systems. Consequently, the relationships among services is far more vague than the relationships among actions in an active database. A well defined semantics for these relationships is currently devised, based on previous works (e.g. [12]) to support priorities among services. The relationships between a detection of an event and an evaluation of a condition, and the relationships between an evaluation of a condition and an activation of an action generate a special case of inter-transaction relationships that define the atomicity of the operations within CoopWARE. An algorithm is devised to generate and maintain a correct set of interrelated atomic transactions based on the relationships among events and services. A similar approach was taken in transaction models dealing with long transactions (e.g. [20]).

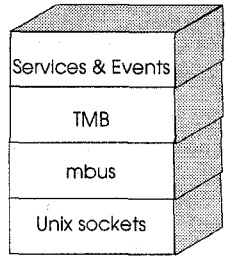

Figure 7. Coordinator implementation layers

5 Implementation and Experiences

This section presents some implementation issues (Section 5.1) and some of the experiences in the framework of the reverse engineering case study (Section 5.2.)

5.1 Implementation

The Information Repository (IR) has been implemented in C++ and uses ODI's object-oriented database ObjectStore for persistence. The IR implements a Telos language interpreter for data modeling and offers services such as object creation, deletion and update, also object retrieval through a query facility. The IR is currently being re-implemented as a main-memory version using flat files for persistence.

Turning to the implementation of the coordinator, we have implemented the coordinator layer architecture from the TMB (Telos Message Bus) layer down (Figure 7) and are currently in the process of implementing the Services & Events layer. The TMB layer, implemented in C++, is an extensible message server through which all tools can communicate, both with the coordinator and with each other, using the common schema. These messages form the basis for all communication in the system. The message server has been implemented on top of *mbus*, an existing public domain software bus technology [6]. This, in turn, is dependent on the presence of Unix software sockets; these protocols have been ported to almost all modern computer platforms, including Dos/Windows and Apple machines. Since the basis for integration is this communication, our architecture can incorporate tools running in heterogeneous environments.

The TMB offers message-passing using message objects with client facilities for message creation, deletion and archiving. The design goals of the TMB included extensibility, in the sense that a client can dynamically inform the system that it can handle a new kind of

request (for example, when Refine implements a new complexity measure it would register this capability, and thereafter other tools could make use of it). An important requirement is that the TMB support point-to-point as well as broadcast communication so that tools, for example, can send messages to a particular tool or to all tools of a certain type. Efficient transmission of bulk data was deemed to be critical (rather than object-by-object retrieval), since that is the intended modus operandi for the integration architecture.

The implementation of the rule language and execution engine is currently under design. It is based on work reported in [15], notably the notion of a dependency graph, built by analyzing given rules and data elements. Figure 8 presents a partial dependency graph for the rules of the reverse engineering case study. Events are depicted as triangles, operations as circles, and data elements as rectangles. Triggering edges connect an event with a condition and a condition with an action of the same rule. In case there is no condition part in a rule, a triggering edge connects an event directly with an action. For example, the event *Rigi-Parses* triggers a rule of Rigi and therefore a triggering edge connects the event with the first action of this rule Parse. Update edges connect actions with the data elements they update. For example, *CBC-Analysis* updates *Grouped-Components*, resulting in an update edge from *CBC-Analysis* to *Grouped-Components*. Request edges are depicted using dotted lines, and represent data element that a service uses, yet its modification does not trigger the service.

The dependency graph is used by the coordinator at run-time, based on an execution model as given in [14]. The results of events and messages are evaluated and any outgoing edges are triggered. For example, consider the dependency graph as given in Figure 8. Whenever a *Rigi-Parses* event occurs, Refine's rule is activated. Refine parses the source code and performs the CBC analysis. The result of this analysis is the update of Grouped-Components. This update is informed through the event *Grouped-Components-Updated* event. This event, in addition to the *AST-updated* activates Rigi's rule, which performs the SG-Analysis.

The generation of the dependency graph is a straight-forward procedure. The time-complexity of generating a dependency graph is bounded by O(|V| + |E|). It should be noted that the number of nodes and edges reflects the number of schema elements and the relationships among them, and not the number of actual objects in the information systems. Thus, the space complexity of the graph, which is bounded by O(|V| + |E|) is relatively small. It is also worth noting

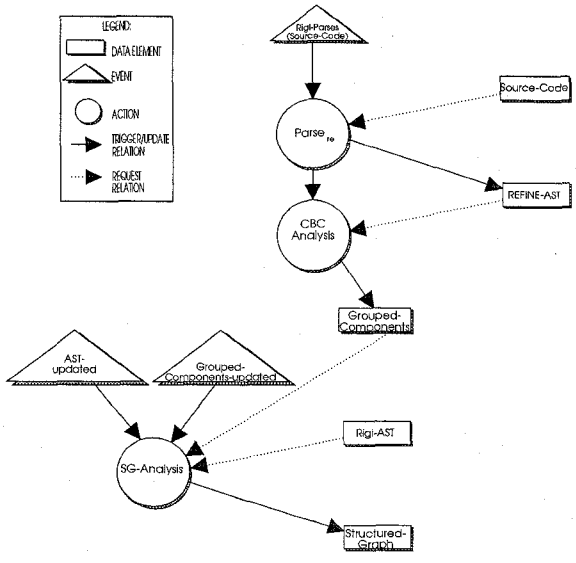

Figure 8. A partial dependency graph of the case study

that the generation of the dependency graph is carried-out only whenever there is a change in the schema definitions or changes in operation clauses. The complexity of the coordination depends more on the number of rules than on the number of attributes devised as a result of the data integration. Therefore, scaling up depends on an efficient implementation of the rule engine rather than on the result of the data integration.

5.2 Experiences

Our experiences in integrating Refine and Rigi was gained over a three-year period and involved use of the integration architecture by three different research groups which used Refine or Rigi and wanted to integrate their code analysis tools with others, in order to support more sophisticated reverse engineering scenaria.

We found that data integration involves not only the identification of commonly used information by different tools, but also the identification of a number of possible use scenaria for this information. For example, the analysis that is carried out by combining Rigi's and Refine's capabilities requires a more complex information schema than that used for other purposes. Moreover, the use of browsers and user friendly interfaces made it possible to experiment with different types of analyses in order to understand the contents of the repository. Consequently, a customized repository

browser was built.

As claimed, the integrated environment provided more functionality than each of the individual tools working on its own. By combining analyses from the two tools we obtained results that would not be easily obtained by using only one tool. For example, a Refine cluster analysis based on use of common resources, that when imported to Rigi and combined with clusters obtained by analyzing the call graph and data types in the Rigi environment we generated system views that are not available by using Refine or Rigi separately. Moreover, the integrated system allows for a number of developers working with different machines and environments to cooperate for the analysis of a software system, thus leveraging the capabilities and the expertise of maintainers at different sites inside an organization. The repository allows for more than one user to access the analysis environment allowing for multiple analyses to be performed simultaneously at different machines. This is especially important for computationally expensive analyses applied to large systems (>1MLOC).

6 Related Work

The CoopWARE architecture relates to a number of different research areas. Distributed databases, heterogeneous databases, multidatabases, federated databases, multi-view systems, workflows, global information systems, Internet applications and many other types of information systems are considered to be cooperative information systems.

Global information systems [24] are systems that involve a large number of information resources distributed over computer networks, with autonomous maintenance of data. Global information systems are similar to multidatabases (e.g. [27]) in their approach. Both system types seek to support global updates, while preserving site autonomy. Distributed databases [7], unlike global information systems, do not have a full autonomy of their data. Issues in structuring heterogeneous databases are discussed in [4], and others. Federated databases [28] can be implemented based on several architectures. For example, federated databases can be implemented using loosely coupled systems, where the responsibility for understanding the semantics of the global system is transferred to the user. Federated databases can also be implemented using tightly coupled systems, where a global conceptual schema is used to define the semantics of the global schema. Research in the workflow area has been influenced by the concepts of long running activities [11], multi-system applications [1], polytransactions [29], and application multiactivities [18]. METEOR [22] is an example of a project whose objective is to support multi-system workflow applications.

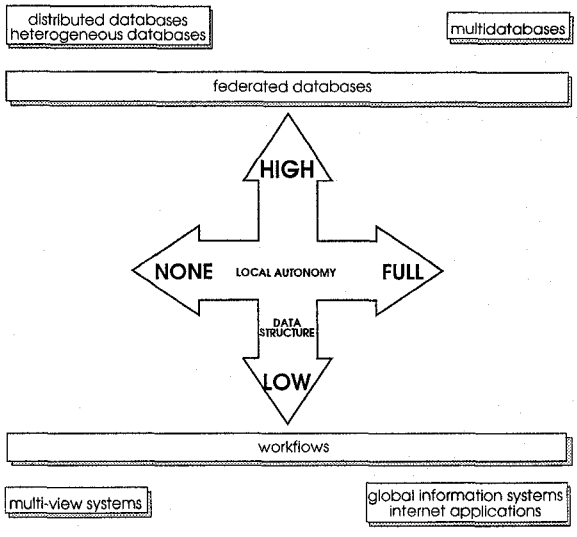

Figure 9. Distribution of information systems

Having considered the common features of the above mentioned models, we design the integration architecture to support common features while enabling different levels of autonomy, rigid structure and data integration. Figure 9 presents the distribution of these information system types on a two-axis grid. The horizontal axis depicts the level of local autonomy given in each information system type and the vertical axis represents the rigidity level of the information system's data structure.

The architecture is aimed at serving as an underlying architecture for systems with different levels of local autonomy, different rigidity levels of data structure, and different levels of data integration. As such, it provides a mechanism for the cooperation among information systems, without loosing the model generality. Such a model enables research in this area to be more focused on issues that are common to all the above mentioned systems, such as alternative paths of operations in case of network failures and the application of meta-data.

MARVEL [2] is a process-centered environment that supports teams of users working on medium to large scale projects. An instantiated environment is created by an administrator who provides the data schema, process model, tool envelopes (which are equivalent to components), and coordination model for a specific application. The process is described in a process modeling language. Each process step is encapsulated in a rule. The body of a rule consists of a query to bind lo-

cal variables; a logical condition that is evaluated prior to initiating the activity; an optional activity in which an arbitrary external tool or application program may be invoked through an envelope; and a set of effects that each assert one of the activity's alternative results. While CoopWARE use events to generate an automatically triggered environment, the MARVEL model does not use events as part of the model. In addition, the model imposes a rigid structure of rules, while CoopWARE assumes a rule where its last phase (the services) conclude the liability of the central mechanism, thus enabling a simpler transaction model.

ARCHON (Architecture for Cooperative Heterogeneous On-line Systems) [17] aimed to develop an architecture, software framework and methodology for multi-agent systems for real-world industrial applications in the area of power system control supervision. The technology is based on a distributed AI approach, where coordination and responsibility are transferred to the components (termed subsystems in ARCHON) rather than being handled by a central mechanism. This approach may require major modifications to the components, and therefore far less useful in the environment of existing softwares.

In ARCHON, tasks are grouped together into recipes, which are production rules, with TRIGGER, ACTIONS and RESULTS. The trigger part combines events and conditions. This method, in many cases is less effective that the separation of events and conditions to different elements, evaluated at different steps. In addition, the recipes requires a long-term control, while rules in CoopWARE enable a flexible mechanism with short term atomicity enforcement.

CORBA [9] provides a widely accepted formalism for specifying process communication in a client-server environment. In the CORBA model objects provide services, and clients issue requests. Object references are used for a client to locate the appropriate servers, so that it can direct its requests to them. The architecture proposed in this paper targets the enhancement of a CORBA by providing a formalism for modeling communication in terms of rules that allow for the specification of data dependencies between processes as well as high level task scheduling. Moreover, the approach is not tightly coupled to a particular language or implementation (i.e. CORBA/C++) and treats any process or tool as a black box that offers in the environment a set of services as well as data via a registered interface.

The area of active databases has gained an increasing interest in the research community during the last few years, resulting in numerous models and prototypes. The leading paradigm in this area is the ECA (Event-Condition-Action) that was proposed in the framework of the HiPAC project [8]. Many of the ECA-based systems lack global control of an application's set of rules. This may result in non-deterministic behavior which would have disastrous consequences in cooperative information systems.

An alternative approach to the ECA paradigm was proposed in the framework of the PARDES model [13] that is aimed at providing a high-level tool for derived data using a data-driven invariant language. The active model, as given in this work, is an extension of the PARDES and other similar models [14]. It uses the analysis and control capabilities of these models to generate a deterministic, well-understood cooperative architecture.

7 Conclusions

We have presented a view of cooperative information systems which is based on the premise that such systems consist of legacy components integrated through an architecture. The paper then proposed a generic architecture for cooperative information systems called CoopWARE. The architecture supports facilities for data integration of the components constituting the cooperative information system, also mechanisms for coordination and policy enforcement among these components. The novelty of the proposed architecture lies in the integration of concepts from data modelling, active databases and organizational systems. In addition to the proposal, the paper reports on a (partial) prototype implementation of the architecture and some experiences in applying it to build an integrated reverse engineering environment.

Several issues require further research. Firstly, the design of the rule execution engine with an appropriate transaction mechanism and the implementation on top of the TMB is currently under way. Secondly, we need to study several other cooperative information system applications to establish the adequacy of the proposed architecture, or to further refine the current proposal. Thirdly, we would like to study the co-existence of several coordinators within one cooperative information system, each of which manages a partial information repository and only supports some coordination and policy enforcement activities. This will form a distributed CoopWARE environment; we are currently studying how this might be applied to Internet services and data.

References

[1] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth.

Using flexible transaction to support multi-system telecommunication application. In *Proceedings of the 18th VLDB Conference*, Vancouver, british Columbia, Canada, 1992.

[2] N. S. Barghouti. Supporting cooperation in the Marvel process-centered SDE. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 21–30, Tyson's Corner, Virginia, Dec. 1992.

[3] S. D. Bird. Towards a taxonomy of multi-agent systems. *International Journal of Man- Machine Studies*, 39:689–704, Feb. 1993.

[4] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB journal*, 1(2):181–239, 1992.

[5] A. Brown et al. *Principles of CASE Tool Integration.* Oxford University Press, 1995.

[6] A. Carroll. *ConversationBuilder: A Collaborative Erector Set.* PhD thesis, University of Illinois, 1993.

[7] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* Mcgraw-Hill, 1984.

[8] U. Chakravarthy. Rule management and evaluation: An active DBMS perspective. *ACM SIGMOD Record*, 18(3):20–28, Sep 1989.

[9] W. Cook. Application integration, not application distribution. In *ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993*, pages 70–71, Apr. 1994. Published as ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993, volume 5, number 2.

[10] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active object-oriented database model. In *Proceedings of the 2nd International Workshop on Object-Oriented Databases*, pages 140–149, Sep 1988.

[11] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of the 17th VLDB*, pages 113–122, Sep 1991.

[12] O. Etzion. Active interdatabase dependencies. *Information Sciences*, 75:133–163, 1993.

[13] O. Etzion. PARDES — a data-driven oriented active database model. *SIGMOD RECORD*, 22(1):7–14, Mar 1993.

[14] A. Gal. *TALE — A Temporal Active Language and Execution Model.* PhD thesis, Technion—Israel Institute of Technology, Technion City, Haifa, Israel, May 1995. Available through the author's WWW home page, http://www.cs.toronto.edu/~avigal.

[15] A. Gal, O. Etzion, and A. Segev. Tale — a temporal active language and execution model. In *Proc. CAiSE'96*, Crete, Greece, May 1996.

[16] M. Hammer and J. Champy. *Re-Engineering the Corporation: A Manifesto for Business Revolution.* HarperCollins Publishers, 1993.

[17] N. Jennings, L. Varga, R. Aarnts, J. Funchs, and P. Skarek. Transforming standalone expert systems into a community of cooperating agents. *Engineering applications of Artificial Intelligence*, 6(4):317–331, 1993.

[18] L. Kalinichenko. A declerative framework for capturing dynamic behavior in heterogeneous interoperable information resource environment. In *Proceedings of the 3rd RIDE International Workshop on Interoperability in Multidatabase Systems (IMS'93)*, Apr 1993.

[19] K. Karlapalem, Q. Li, and C. Shum. Hodfa: An architecture framework for homogenizing heterogeneous legacy databases. *SIGMOD RECORD*, 24(1):15–20, Mar 1995.

[20] J. Klein and F. Upton IV. Constraint based transaction management. *IEEE Data Eng. Bull.*, 16(2):20, June 1993.

[21] G. Kotik and L. Markosian. Automating software analysis and testing using a program transformation system. Technical report, Reasoning Systems Inc., 1989.

[22] N. Krishnakumar and A. Sheth. Specifying multisystem workflow applications in meteor. Technical Report TM-24198, Bellcore, May 1994.

[23] S. Laufmann, S. Spaccapietra, and T. Yokoi, editors. *Proceedings Third International Conference on Cooperative Information Systems*, Vienna, May 1995.

[24] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 5:121–143, 1995.

[25] H. Muller, B. Corrie, and S. Tilley. Spatial and visual representations of software structures. Technical Report Tech. Rep. TR-74. 086, IBM Canada Ltd., Apr. 1992.

[26] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, Oct. 1990.

[27] M. Rusinkiewicz et al. OMNIBASE: Design and implementation of a multidatabase system. In *Proceedings 1st International Conference on Parallel and Distributed Processing*, pages 162–169, Dallas, TX, May 1989.

[28] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[29] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using polytransactions to manage interdependent data. In A. Elmagarmid, editor, *Transaction Models for Advanced Database Applications*, chapter 14. Morgan Kaufmann, 1992.

[30] G. Valetto and G. Kaiser. Enveloping sophisticated tools into computer-aided software engineering environments. In *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 40–48, 1995.

[31] M. Whitney et al. Using an integrated toolset for program understanding. In *Proceedings of the CASCON'95*, pages 262–274, Nov. 1995.