International Journal of Software Engineering and Knowledge Engineering
Vol. 1, No. 2 (1991) 165–188
© World Scientific Publishing Company

THE ORGANIZATIONAL IMPACT OF INTEGRATING MULTIPLE TOOLS

MIKE P. PAPAZOGLOU
Australian National University, Department of Computer Science, GPO Box 4, Canberra ACT 2601, Australia

LOUIS MARINOS
Erasmus University, Laboratory for Artificial Intelligence, 3000 DR Rotterdam, Holland

and

NIKOLAOS G. BOURBAKIS
IBM, 5600 Cottle Rd, San Jose, CA. 95193, USA

Received 8 January 1991
Revised 5 March 1991
Accepted 9 March 1991

This paper elaborates on the data modeling and data sharing issues that arise when developing and interconnecting modern software development environments. It primarily-focuses on the investigation of adjoined object management issues for tool integration, software development environment extensibility, and interconnectivity.

1. Introduction

Software Engineering Environments (SEEs) aim at providing an integrated and coherent set of tools for managing the design, construction, testing, use, and eventual reuse of software, thereby increasing software productivity and product quality [1, 2]. A key objective of SEEs is to support software projects in the generation, management and control of the volume of data and associated information which is generated and utilized during the software development life-cycle. The term environment is reserved to encompass the entire set of automated facilities, such as a unique linguistic and operating framework, required to support all the activities of the software life-cycle and allowing to switch smoothly from one task to another. Desirable features of such an environment include: ease of tool integration, reusability of internal components, extensibility, support for prototyping, and automated support for the various tasks performed during the life-cycle of multi-user, multi-version software development projects.

The development of such a complex software system involves the management of a significant volume of diverse kinds of data and associated information (such as data properties, constraints or relationships to other data) which make up a

software project. The result of user interaction with an SEE is a number of identifiable complex data entities, called (software) "objects",[1] which carry information relevant to the software development life-cycle at a suitable level of abstraction. Typical examples of objects are constructs generated by a project to support the software under development such as source, object, and executable code objects as well as requirement, specification, schedule, test plan and test data objects.

To reduce the efforts spent in maintaining this enormous and highly interrelated amount of data and at the same time improve the overall productivity for software development, SEEs provide a special component for storing, managing, and manipulating the objects. This component, referred to in the literature as the Object Management System (OMS), can normally range from any ad-hoc and application specific storage system to modern DBMSs with flexible object-oriented modeling facilities [3]. The OMS is the core component of an SEE: it is used as the key platform to achieve tool integration and communication [1] while primarily contributing to the construction of an integrated CASE environment [4, 5].

This study emphasizes two important goals of modern software development environments, namely their *extensibility* and *interconnectivity*. Extensibility and interconnectivity are two of the most essential features of modern SEEs. There is clearly a need for systems to evolve over time: it is expected that new tools will be gradually added. Moreover, since many (if not most) software projects are developed on computer networks it is also reasonable to expect that the domain of the local data will exceed the boundaries of the local OMSs. To cope with this issue it is required to interconnect the individual SEEs (physically distributed over different workstations) by providing the operating services of a distributed object manager capable of handling objects of diverse types.

Extensibility implies the ability of tools to be easily augmented in terms of the operations performed and the data they operate upon. An SEE is intended to be extensible in order to support experimental investigation of software process models and evaluation of novel tools in the context of a complete environment. Extensibility leads to the notion of system *openness* where integration of software development tools is supported in such a way as to facilitate the task of adding new tools to the environment. Interconnectivity, on the other hand, implies the ability to support the smooth integration and maintenance of multiple SEEs developed over a network of computing resources. Interconnectivity leads to the notion *tool interoperability* where tools are viewed as interoperable processes.

This paper outlines the technical problems associated with the topics of tool integration, extensibility and interconnectivity and proposes appropriate solutions. We start by briefly explaining how tools in an SEE can be integrated and how they

[1] The term object is used in a generic sense, it bears no direct connotation to the object-oriented programming style. What constitutes an object is relative: it largely depends on the object management system used and on how users utilize the typing system of this particular model to define software objects.

can communicate via the use of an object management system. We then outline the various technical problems and conflicts associated with tool integration and propose solutions in terms of a common set of modeling requirements that should be offered by future OMSs to guarantee extensibility and interconnectivity. Finally, we explain how different SEEs can become interoperable and can exchange information and services under the use of a common integrating platform which exhibits "intelligent" behavior.

2. The Organizational Impact of Using Multiple Tools

An SEE consists of a set of CASE tools which assist users with the various software development activities of the software development life-cycle. CASE tools support all the aspects of system design and construction. Depending on their functionality CASE tools may be divided into *upper* and *lower* CASE tools. Upper CASE tools include tools that support techniques like structured analysis [6] conceptual data modeling (e.g., ER-modeling, entity life-cycle history), and techniques such as pseudo-code specification, action diagrams or minispecs. Lower CASE tools comprise traditional fourth generation systems (i.e., application generators with tools for dialogue design, screen painting, automatic generation of database schemas, code generation, etc.). Both upper and lower CASE tools must have connections to *implementation tools* which facilitate code production (e.g., program-development tools such as editors and command interpreters) and manipulation activities (e.g., language-specific tools such as compilers, linkers and loaders, symbolic debbugers, test tools, etc.). A tool[2] can be viewed as comprising a set of functions which accept as operands (and manipulate) the various objects created by other tools or users [7, 8]. Commands, i.e. invocable tool services, are essentially interpreted as the application of tools on objects and the production of new objects as a result of applying tool operators on input objects.

2.1. *Types of tool integration*

Conventional CASE tools specialize in automating diverse very specific software tasks offering, thus, only a very limited support for the various phases of the software development life-cycle. In some cases tools offer a functionality which is complementary to that of some other existing tools. The objective is to combine, or *integrate*, these tools into a coherent, comprehensive, automated software development environment which supports users in activities related to software development tasks. In fact, there exists a necessity to integrate upper CASE with lower CASE tools, and both of these types of tools with implementation tools. It is also important that the integration is achieved in such a manner that changes in the upper and lower CASE tools are automatically reflected in the implementation level. Tool integration must accomplish the following two interrelated purposes:

[2] Henceforth signifying an upper, lower CASE or implementation tool; unles otherwise stated.

1. Software developers should have at their disposal a uniform working environment where distinction between different tools is completely transparent.
2. Tools must support the diverse tasks that a software developer may choose to perform during the systems development life-cycle.

The foundation that allows integration of tools is the object repository: tool combination and inter-tool communication is achieved at the data, viz. the OMS, level. The true value of tool integration may be defined as the ability of tools to create and manipulate objects (according to agreed-upon definitions from the part of tools) which are freely shared and have common applicability across all the integrated tools. As a general rule, all tools must have complete access to the OMS so that they can define the various types of objects required for their tasks, and obtain access to the objects produced and shared by other tools participating in the same or related development activities. The results of operations performed by tools can be seen by all other tools, which in turn can contribute to that operation. This form of integration may be described as a *tight integration* of tools, as they are required to communicate only via shared access to a common object repository and object management system.

In tightly integrated tool environments the communication traffic and data interchange between tools is delegated to the OMS which provides the standard tool interface through which tools can create, access and modify objects associated with the software development life-cycle. This shared repository maintains information about the semantic content of objects in a common semantic dictionary, together with tool-specific views that specify modes of tool interaction. Typical examples of this approach include structure editor-based environments [9, 10] and some modern knowledge aided systems like Marvel [11], and Worlds [12].

The tight form of tool integration offers numerous advantages such as fine-grain tool cooperation, data sharing at an abstraction level, with a common homogeneous kernel—viz. the OMS, providing a common interface to all tools and guaranteeing tool evolution, reusability and extensibility. To achieve tool integration the OMS must provide upgraded data modeling facilities suited to SEE infrastructure and should also cater for both data exchange and integration services [13].

2.2. *Tool composition and data interchange*

Tool integration may be thought of as the complex task of allowing the diverse tools used in different life-cycle phases to operate together to produce the total software product [14]. Regardless of whether objects of an SEE are local or distributed (they span local or remote workstations), several generic mechanisms must be provided in order for an OMS to be in the position to keep track of consistent creation, deletion, re-definition and tarnsformation of objects. For practical purposes we may differentiate between two kinds of activities which underly the OMS framework: *conceptual modeling* and *object implementation* activities.

Conceptual modeling activities capitalize on the standard repertoire of modeling primitives, provided by the OMS, and are used to prescribe the appropriate solution in the context of the desired application domain. Here, the abstract components, i.e., tools, making up the system are indentified and their respective data structures, structural relationships, and the pattern of tool communication are specified. Object implementation activities, on the other hand, employ those data management techniques that are required to implement and coordinate the objects that result from conceptual modeling. In the following we will outline these two kinds of activities and examine the implications they bear on the task of multiple tool integration.

1. Conceptual modeling activities employ two complementary methodologies to represent functional tool constituents and specify the type of data exchanged between tools. We refer to the former methodology as the functional composition of tools and to the latter as the data interchange methodology. In the following, we will examine how these two techniques provide the basis for composing tools and combining multiple tools for constructing integrated toolsets.

- *Functional composition of tools:* An SEE must be a vehicle for providing extensive and growing tool capabilities. These capabilities should be furnished primarily by collections of tool fragments, rather than by a few, large, monolithic tools. Under this perspective, a tool is seen to be composed of a series of functional fragments each one implementing specific portion of a tool operation, e.g., a symbolic debugging tool may be defined as an object providing a collection of methods for displaying the status of a program in execution, browsing through the call history, inspecting actual values of data structures, stepping through the program calling sequences, etc. Function composition aims at mapping identifiable physical fragments of tool functionality into a set of functions and to synthesize tools from related tool fragments. Tools should be viewed as aggregates of tool fragments whereby inter-fragment dependencies are explicitly specified. A further requirement is the ability to combine existing functions, distributed across multiple tools into a single functional unit. For example, the task of pretty-printing may be synthesized from a collection of tool fragments which include a lexical analyzer, a parser and a formatter. This results in promoting consistency of operations and avoiding the duplication of implementation efforts.

  One of the main advantages of composing larger tools out of smaller fragments, or tools for that matter, is that if the tool fragments are carefully chosen they will prove to be usable as components of a variety of larger tools, thereby enabling the creation of those tools at much reduced costs. It is, thus, obvious that this methodology serves as basis for extensibility and reusability. Here, a specific tool fragment may be replaced by another as long as this replacement produces and draws upon objects of the same type as the other tool fragments in the tool. This enables not only considerable flexibility by

replacing tool fragments with equivalent ones, but also facilitates the integration of new tool fragments.

The value of the functional composition methodology lies in finding autonomous operational tool fragments. Once these fragments are identified, one can identify their data interchange needs and satisfy them through the functionality provided by the OMS.

- *Data interchange methodology*: Functional behavior can be defined in terms of input and output objects involved in or produced by tool invocations. The type of these input and output objects as well as the possible ways of combining and/or structuring them to satisfy the data requirements of tool fragments is guided by the data exchange methodology. The tool data exchange methodology makes sure that tool fragments or individual tools are able to communicate by either producing compatible objects to be read by other tool fragments or tools, or by reading objects produced by other tools or tool fragments. Consider for example a pretty-printing tool which must work in conjunction with a lexical analyzer and a parser. If the pretty-printer were to access the output of a proven lexical analyzer and parser and be conceptually based on such data constructs as lexical tokens, parse trees, and symbol tables, this tool is likely to be better for its reliance on a robust and proven code than had it been created from scratch.

  The most basic form of data interchange between tools is through a mutually-agreed-upon format of exchangeable data (objects) defined as the interface between the tools. Thus, tool fragments access a needed object only through an accessing primitive designed to accept as parameter the type of which this particular object is an instance. This enables tool fragments to share and manipulate objects created by other tool fragments while also incorporating those primitives that will eventually ensure the integrity and extensibility of an SEE. This methodology guarantees tool interoperability at the level of data interchange, viz. at the object-level. For our purposes two or more tools are interoperable if they are able to communicate or interact to perform software development tasks jointly or in a cooperative fashion.

The aim of the preceding methodologies is to facilitate the development of a versatile environment that supports the activities of software developers by coordinating their actions and operations. However, to meaningfully compose and integrate tools it is not only necessary to employ the functional composition and data interchange techniques but also to resolve all kinds of incompatibilities which may arise during the tool composition and integration process. In general, the inherent inability to freely interchange data (objects) across a diversity of tools stems from at least three forms of incompatibilities which arise as a result of the attempt to achieve a meaningful integration of multiple tools:

(a) *Functional incompatibilities*: These are incompatibilities which arise from the diverse data requirements imposed by the fragmentation of the tool

functionality and by distributing different tool functions across many tools. The data requirements of tool fragments which need to be combined into a single tool and tools which need to communicate may be based on different data structures and representations. Consequently, in order to integrate tools one needs to cater for the rectification of structural and semantic incompatibilities found among the exchangeable forms of data [15, 16].

(b) *Methodological incompatibilities*: Such types of incompatibilities arise from the use of differing *methodologies* promoted by different tools. A methodology may be defined as an ordered collection of *methods*, like structured analysis, ER-modeling, and project management methods, performed by tools in order to develop a software product. A method may be viewed as the specific approach taken to solve aspects of a software development related problem. For example, data flow diagrams represent a method for developing process decomposition during the requirements phase. As different tools promote different methodologies, any viable tool combination must resort to consistency checks carried out on different parts of the specification. If say, two CASE tools are used in conjunction, one promoting data flow diagrams (DFDs) for functional specification and the other one ER diagrams for data definitions, then it should be guaranteed that the data stores on the DFD hierarchy correspond to entities and/or relationships on the ER diagram and that their respective names match. As a result we must ensure that correspondences between equivalent structures of differing degree of granularity and detail are established. As already stated, it should also be perceivable to develop a certain software product by following different development strategies, e.g. by using different tools with overlapping functionality.

(c) *Distribution incompatibilities*: The tool data exchange methodology may be viewed from the perspective of multiple cooperating developers who rely on communicating toolsets distributed among different workstations with each workstation possibly relying on its own private object-base for data management. This specific category of incompatibilities introduces not only incompatibilities between different OMS data definition costructs but also incompatibilities which arise from the use heterogeneous machine configurations, operating systems and so on.

One of the major purposes of the OMS is to resolve these forms of incompatibilities by making sure that the structure of the shared data is made conformant to all interacting tools in a toolset. The OMS handles the details of tool connectivity, including representations for data structures, organizing the structure of shared data and checking consistency between connections. Our view is that the desirable conceptual modeling primitives which assist the OMS in their effort to resolve all forms of incompatibilities can be provided by recent developments in information systems technology, i.e., object-oriented and knowledge-based data models.

2. Object implementation activities require a common data repository for coordinating access to and storage of the objects shared by tools. Thus, the object implementation activities instrument the conceptual modeling framework. This repository handles functions which span conventional data management activities such as storage management, concurrency control and recovery and extends them by providing persistence for typed objects. Such issues have been extensively researched and are addressed by the relevant literature [17] and, thus, will not be examined in this paper.

In the following section we will first elaborate on the desirable OMS modeling primitives and explain how these primitives can be used for managing the structure of the data exchanged between the tools, and consequently promote tool extensibility and interoperability, and then we will explain how these features can be used to resolve inter-tool conflicts and incompatibilities.

2.3. *Desirable OMS modeling primitives*

The objects which require management in an SEE range widely in size and character. Sophisticated object management support is required to faithfully represent objects of varying granularity such as parse trees, abstract syntax graphs, symbol tables, source codes, test plans, designs and so on. This suggests that the SEE's infrastructure must provide the basic facilities for explicitly modeling the intricate structure of software objects which represent information both at a coarse and at a fine grain level. Several systems have been, or are being, developed around semantic or object-oriented data modeling facilities [11, 2, 18, 8, 3], to provide a rich repertoire of object management facilities regarding the definition, manipulation and storage of complex entities which may be used to model software objects.

The use of the OMS as an integration platform unfolds the possibility of building powerful information retrieval facilities which can collate and present component information generated at different stages in the software life-cycle. The OMS employs an *object-base schema* to publicly define the types of objects that are handled by tools and specify the data structures communicated among tools. Object definitions, thus, do not belong to the code of specific tools but are made publicly available by the OMS.

This section summarizes the most salient conceptual modeling features that contribute to the overall OMS functionality. The proposed conceptual modeling features combine object-oriented programming with rule-based modeling. Each feature is studied from the perspective of OMS extensibility and with a view to ultimately supporting OMS interoperability. The capabilities sought for each of these issues and some interesting problems are briefly discussed below.

- *Typing facilities*: The OMS can manage objects produced by tools used in all phases of the software development life-cycle. The key factor to the OMS's expressiveness is the notion of *typed objects*. The OMS enforces typing which includes definition of type properties (e.g., file objects and associated properties

like the name of a file, content of a file and so on) and behavior (i.e., how an object acts in terms of its state changes and message passing) as a set of operations (typed methods) attached to a typed object (e.g., operations on file objects such as create, open, close, concatenate and so on). In an SEE where all objects are instances of abstract data types, it is easier to share objects or to modify their implementations. Thus, the OMS is able to hide the internal structure and implementation of objects behind well-defined interfaces and can accordingly control the operations performed on those objects.

The type system needs to be flexible and powerful enough to capture information which relates to tools, processes, and even types themselves and treat it as objects. This implies that we may have objects which are types of types or *meta-types*. Recently, work on type systems, in the context of object-oriented OMS models, has dealt with these and related problems and has focussed on forming a compatible set providing the rich typing capabilities required for SEEs [3, 8, 9, 1].

- *Support for composite and complex objects*: For the administration of objects, it is often desirable to use the notion of object composition as a way to collect together related objects into a single composite (or complex) object. Composite/complex objects can be treated as a single unit while their component objects can be treated as separately identifiable and accessible units.

  Complex objects possess the ability to establish simple inter-object references, whereas composite objects exploit this mechanism further by granting existence dependency properties to inter-object references. Consider for example the attribute *compiles-into* that relates a *source-code* object to its object code counterpart. If we choose to represent the *source-code* object as a composite object, then the *compiles-into* object becomes an integral "part-of" the *source-code* object. If the *source-code* object is deleted then its *compiles-into* counterpart object is also automatically deleted.

- *Varying object granularity*: Most conventional SEEs are able to store large objects, like programs or documents, as a single unit. However, it is desirable that such coarse objects are decomposed into smaller components. For example, it is desirable to store each of a program's procedures as a separate object, rather than storing them in a single coarse object, as is the current practice with file systems.

  The granularity of objects in modern SEEs is normally chosen to be in a fine-grained form so that the administered objects are compatible with the data structuring mechanisms encountered in high level programming languages. This choice of granularity at a coarse level poses a real threat: sub-granular references should be kept to maintain the consistency and validity of information. When objects get modified, existing sub-granular references to them may be invalidated. This is a typical case when dealing with program code modules (e.g., Ada packages) where we need to support libraries of compilation results.

The issue of granularity is very much related to that of complex/composite objects as they are used when dealing with larger software units, such as program systems, documents or even entire projects whose structure must be made available system-wide. Such objects cannot be stored in the OMS as a single unstructured object. They must be modeled as an abstraction of a set of related simple or complex/composite objects and their interrelationships. Therefore, it is evident that with the use of complex/composite objects the problem of sub-granular references to component objects (The integral parts of the composite object) can be eliminated by appropriately mapping it to that of representing simple or existence dependent references towards the component objects.

- *Inter-type relationships*: The ability to capture and maintain relationships between objects in an SEE is closely related to its ability to define and maintain typed objects. Through the typing system, relationships between types can be properly expressed and manipulated. An association between objects may itself be considered as a typed object and further relationships may be built upon this relationship type. For example, the fact that a particular *object-code* module is related to some *source-code* module by a relationship *compiled-from* is an important property as it indicates that the *link* operation may only be applied to instances of that type.

  Currently, there exist several tools which reason about or exploit relationships among objects. Typical examples include version control systems and automated system building tools. As a rule, defining the relationships among the environment's tools and information contents, viz. objects, facilitates the upgrading of the environment since the effect of upgrades (modifications) can be relatively easily determined by examining the inter-object relationships.

- Constraints: Depending on the domain and semantics of object properties, it may be necessary to constrain object creation in such a way as to ensure that their attributes assume only permissible values which belong to a specific range of the value domain. This feature guarantees consistent creation of type instances as opposed to unconstrained creation which involves the obligation to verify data consistency a posteriori (and thereby scales down the performance of the environment considerably). Constraints are incorporated into the type definitions of objects, a fact that gives the OMS additional versatility and expressive power.

  Existing constraint mechanisms can be characterized as value-based and, thus, do not support constraints that can explicitly specify and delimit permissible operations on objects. However, modern OMSs require *action-based constraints* applied on types and their instances. Action-based constraints are provided by modern OMSs through the trigger concept (similar to exception handling mechanisms in programming languages, and to demons in AI systems) [8, 9]. Triggers enable a repair activity to perform certain corrective actions so

that their associated constraints are always satisfied and object integrity is preserved. A typical example of the use of triggers is for automating, replanning and rescheduling tasks when software development activities have to be redefined and/or re-initiated [9]. For example, a change in a specification requires the design activity to be reinitiated. Such events are otherwise manually entered into the OMS which subsequently triggers compensating activities depending on the specified events.

- *Rules*: Rules can serve as the main agents which control active participation in the software development process in terms of activities (e.g., text editing, compilation or debugging) carried out using tools such as editors, compilers, debuggers, etc. Rules define the conditions that must be satisfied for a particular tool to perform an activity which must be carried out on a set of objects; and also specify the effects of the tool invocation. Rules normally implement a chaining model to impose their effects and maintain consistency. In fact, rules are used to implement action-based constraints, just like triggers, and control corrective actions.

  Rules may also be used to automate activity initiation by automatically performing activities which otherwise would have to be carried out manually. For example, in response to the completion of implementing a program module, the rule-system may initiate the testing activity for this particular module. Another important use of rules is that they may be used to apply a backward chaining policy, similar to that of Marvel [11], for invoking mechanisms to rectify attribute mismatches, see Section 2.4. In general, rules are attached to types as *multi-methods* [19], which are sent to a collection of objects of different types, making thus use of inheritance of structure and operators.

To summarize, we may state that the four basic approaches, based on the above OMS modeling primitives, which seek to rectify inter-tool conflicts and improve efficiency via the use of objects are: *abstraction and information hiding, functional decomposition/composition, parameterization* and *rule-based processing*. With the use of abstraction we produce intermediate abstractions for tools whereby tools pay no attention as to how their counterparts came into effect; with functional decomposition/composition we partition the tools into, or synthesize tools from, disjoint fragments which materialize complementary functionality; with parameterization we identify which system parameters affect which tool fragments; and finally with rule-based processing we guarantee automation, consistency and integrity of inter-tool invocations.

2.4. *Resolution of inter-tool incompatibilities*

The focal point of our discussion in this section concentrates on how the OMS modeling primitives like typing, behavioral modeling, inter-type relationships, constraints and the rule system can be used for the resolution of functional and

methodological incompatibilities. Physical distribution incompatibilities are the subject of the next section which elaborates on the concept of corporate software development environments.

It is important to point out that tool management capabilities complement object management capabilities: tools are viewed as functional agents defined and acting upon objects, where objects are the instances of types tailored in such a way as to satisfy the data requirements of the functional tool fragments. Object-orientation provides the natural framework and set of modeling primitives for supporting effective tool management, resolution of inter-tool conflicts and subsequent tool integration. The usefulness of the object oriented paradigm as basis for the resolution of functional and methodological conflicts does not only stem from its superior expressive power (the object model lends itself to the construction of software environments that embody the properties of well-structured and complex systems) but also from the fact that it provides natural support for the data-driven analysis and top-down decomposition methodologies [16, 20]. In the following we will examine how the OMS modeling primitives (based on the object-oriented and knowledge-based paradigms) aid in resolving inter-tool incompatibilities.

The process of resolving functional inter-tool conflicts is a three step activity. First we assert the existence of the relevant tool fragments involved in a certain development sequence; then, we evolve their interfaces in terms of conformant data interchange patterns; and, finally we embed them into a single organizational unit, the object tool type. A tool fragment architecture is always materialized in terms of its functional components and the data structures upon which these functional fragments operate. It is important to point out that the data requirements of the functional tool fragments are of paramount importance: the functional fragments of a tool are completely described in terms of the input and output objects that this specific fragment accepts as parameters or produces as results. Once the data requirements of the functional fragments have been specified, as a result of performing some form of data driven analysis, the input and output objects of the tool fragments that are to be combined are compared to ascertain their commonalities and degree of compatibility. The objective of this step is: (i) to guarantee that functional fragments which are placed in the same tool use compatible data structures (and if not to devise some mapping mechanism); (ii) to additionally develop uniform interfaces in order to facilitate tool fragment interaction with their intra- and inter-tool counterparts. The final step in this process requires that all identified data structures and functional components are incorporated as structural and behavioral properties in their encompassing object type which materializes a concise tool in the SEE.

We exemplify the three steps of the resolution of functional incompatibilities by means of the debugging tool depicted Figs. 1, 2 and 3. This debugging tool is seen to comprise the three identifiable tool fragments (or methods in the object-oriented sense): *ReadObjectCode*, *SetUpStructures* and *Interaction*, see Fig. 1. This figure also depicts the data requirements of those fragments specified in terms of input and

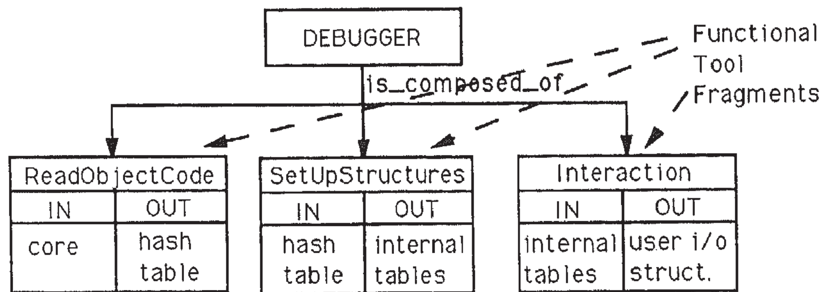

Fig. 1. Functional composition of a debugger tool

output (IN, OUT) in object type parameters. These tool fragments represent the three major functional fragments of a rudimentary debugger and perform quite independent tasks. The tool fragment referred to as *ReadObjectCode* accepts a core file object as input and produces an object of type hash table (of the compiled program) as output. The fragment *SetUpStructures* is responsible for setting up the appropriate structures required for debugging. It accepts as input the hash table object produced by the previous functional module, and produces a set of internal table objects. Finally, the fragment *Interaction* is responsible for providing user access: it accepts as input the set of internal table objects produced by the previous fragment and produces I/O structures for interaction with users.

The specific approach taken to resolve functional and methodological incompatibilities by bestowing an object-oriented flavor on the debugger's (and for that matter any other tool's) functionality is succinctly examined below.

1. *The use of composite objects and inheritance to resolve functional incompatibilities*:
   When analyzing the functionality of a tool in terms of the behavior of its particular tool constituents, we obtain several self-contained functional fragments. These fragments accomplish a narrow and specific portion of the overall tool functionality. As shown in Fig. 1, these tool fragments may be represented by the nodes of an **is_composed_of**, or *functional composition hierarchy*, with each leaf node contributing towards a portion of the functionality of its root node. Nodes signify tool fragments, while the root of the hierarchy signifies their common defining object type, i.e., the debbuger in Fig. 1. The tool solution space is seen as being composed of a set of functional units and the process of tool synthesis defines the overall behavior of the tool as a concise object which inherits and is fully characterized by the individual behaviors of diverse tool fragments.

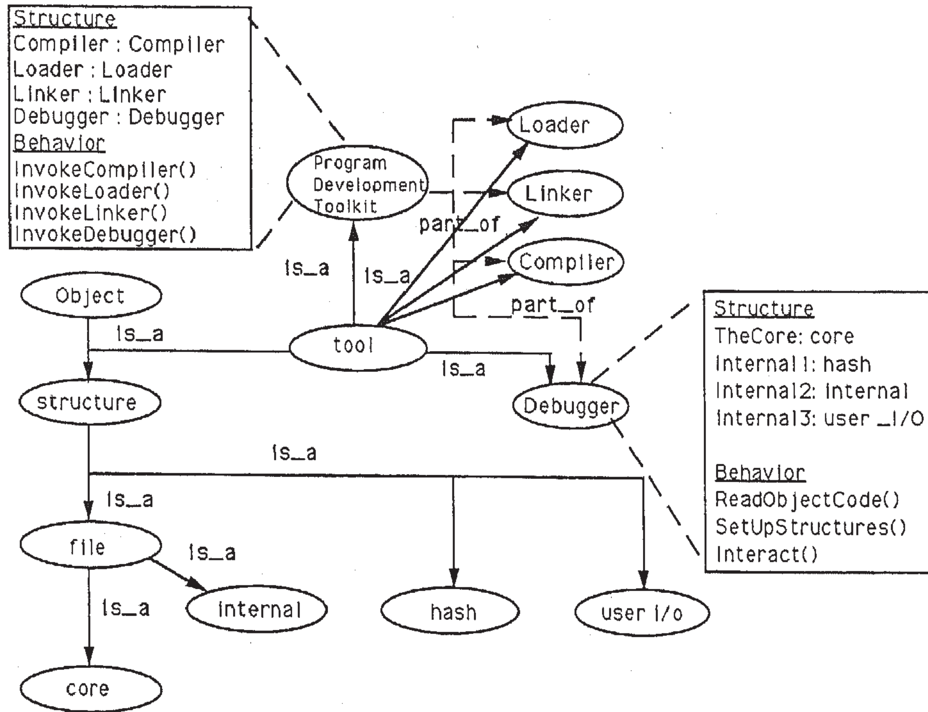

Fig. 2. The tool object type hierarchy

The resolution of functional incompatibilities is accomplished from the fact that any defined component objects directly reflect the functionality of the fragments of a tool, and describe how they are interfaced with the remaining intra-tool fragments. At the same time the functional fragments define an outwardly (inter-tool) visible behavior in that they may be potentially accessible from every other object in the environment. It is worth mentioning, that other objects in the SEE will be in the position to access functional tool fragments as long as they "know" how to tailor their message interfaces to the data structures required to achieve inter-object tool fragment invocations.

This particular methodology employs the concept of composite objects to provide the means for referential sharing of object types, allowing, thus, individual tools to be combined (or grouped together) into a higher-level abstracter tool. A composite object may be profitably viewed as a tree defining an **is_part_of** hierarchy, represented by the dashed lines in the figure. The composite object itself is the root of the tree and the objects which it aggregates are its component objects. Consider for example the case of the composite tool type called *Program_Development_Toolkit* and shown in Fig. 2. This tool type contains the tools *Compiler*, *Linker*, *Loader* and *Debugger* as its component object types to produce higher level functionality.

The functional tool composition and the **is_part_of** hierarchies can be easily combined with the *schema object type*, or **is_a**, hierarchy, represented by the solid lines in Fig. 1, through which tools in an SEE are classified. In this way tools and tool fragments define their attribute domains in terms of object types organized in the object hierarchy. The organizational criteria for the formation of the object type hierarchy are not the same as those of the functional tool composition and **is_part_of** hierarchies: with these hierarchies no sub-typing relationships (signifying property inheritance) are possible. One can only ascertain which functional tool fragments or tools are constituent parts of other tools, but not whether a functionality or a tool is more generic than another—which is the criterion for classifying tools.

Reverting to the debugging example; the specification of input and output data types used as parameters for the tool fragments of the debugger can be organized in a generalization hierarchy as shown in Fig. 2. In addition to this hierarchy, Fig. 2 also shows the individual structural and behavioral properties of the debugger. For example, *The_Core* property specifies the memory map (or core) file object needed for initializing the debugger, while the rest of the debugger's structural properties correspond to the input and output type parameters utilized by its functional fragments like *Set_UpStructures* and *Interaction*. Notice that all the debugger structural properties draw their values from the object types defined in the **is_a** hierarchy. This principle is also employed in the case of the type *Program_Development_Toolkit* depicted in Fig. 2. In the **is_a** hierarchy object types like *Structure*, *File* and so on define a generic structure and behavior exhibited by many object types serving as parameters for either tools or tool fragments.

The preceding discussion has illustrated how the problems of tool composition and construction relates to the general bottom-up strategy of software development which library-based object-oriented design naturally implies.

2. *The use of relations and rules to resolve methodological incompatibilities*: As already stated, methodological incompatibilities are inherent to the utilization of particular tool fragments during the completion of a specific software related task. To be completed successfully such tasks require a synergy of different tools and tool fragments.

   The main problem in defining a methodology for a software development related task lies in providing appropriate input and output data structures that suit the data specifications of the tools or tool fragments used to perform this task. The key issue here is to be able to transform the output data of tools to a format which is compatible to the input format of the next tool or tool fragment in the sequence of the methods required to satisfy a chosen methodology. During this sequence of events all parameter mismatches must be rectified.

inter_structure_transformation

structure	method	structure
internal	int_hash()	hash
core	core_int()	internal
hash	hash_user()	userI/O

method_check

```
transformed_into: Structure
equivalent: Structure

Rule
IF transformed_into ≠ equivalent
THEN look_up_structure_to_structure_production(transformed_into,
equivalent)
invoke(method(transformed_into))
```

Fig. 3. Relationships and rules for methodological conflicts and irregularities

Another important problem in the definition of methodologies is whether a specific order of tool fragment invocations will lead to an effective completion of the requested task. Consider for example the methods of the *debugger* type in Figs. 1 and 2. Although a separate invocation of any method associated with a particular tool fragment in the body of the type *debugger* makes sense, the invocation sequence *ReadObjectCode(core):Interaction(internal)* will lead to inconsistencies. The reason is that these two functional fragments are not called in the right order: the functional fragment *Interaction()* expects to receive as input an object of type *internal* which is unavailable. This object will be properly set up after a call to the functional fragment *SetUpStructures()* which accepts as parameter an object of type *hash* (produced by a previous call to the functional fragment *ReadObjectCode()*) and returns as result an object of type *internal*. In order to prevent this situation from happening, the OMS should be in a position to detect the "functional" gap and data inconsistencies resulting from the ill-chosen method invocation sequence and invoke the appropriate functional tool fragments in the right chronological order.

Both of the aforementioned problems can be solved by using inter-object relationships and rules. Relationships help to rectify methodological incompatibilities and the problem of ill-chosen tool invocations by stating additional dependencies between object types used either as input or output tool fragment parameters. Such dependencies are concerned with providing information regarding the compatibility of input and output functional fragment type parameters during a sequence of tool fragment invocations. For example, in Fig. 3 the ternary relationship *inter_structure_transformation (structure_1, method_name, structure_2)* provides information about the equivalence between its input and output tool fragment parameters

structure_1, and *structure_2* by returning the name of the method which allows to transform the structures of one type into those of the other type. The figure shows three instances of this relationship with the first instance signifying the fact the objects of type internal can be transformed into equivalent objects of type *hash* if the method *int_hash()* is invoked.

To utilize this information, one can think of an additional auxiliary tool, represented as a type called *method_check()*, whose purpose is to check the validity of a given methodology and rectify attribute mismatches. This type is shown in Fig. 3 to implement the functionality of a checker module responsible for establishing the compatibility between input and output tool fragment parameters used during a sequence of debugging activities. To achieve this effect the checker module uses a rule which ascertains and invokes the method needed to transform an object of one type into its equivalent counterpart of a different type. Consider the call sequence with the steps *ReadObjectCode(core):interact(internal)* again; the checker type is in the position to find out whether the attribute core conforms to the type *internal* and if not to find out which sequence of methods is needed to establish this equivalence. The method *look_up_structure_to_structure_production(transformed_into, equivalent)*, called from within the rule where the formal parameters *transformed_into* and *equivalent* correspond to the actual parameters *core* and *internal*, utilizes the the relationship *structure_to_structure_production* to find out that the method *core_int()* will eventually have to be called. This method is subsequently automatically invoked by the rule in the checker module, see Fig. 3. It must be pointed out that since the method *look_up_structure_to_structure_production()* accepts arguments of type *structure* it establishes polymorphic references to objects defined in the **is_a** hierarchy.

The use of rules to rectify methodological inconsistencies is in accordance with the concept of *opportunistic processing* whereby simple development activities can be performed automatically by the model so that software developers need not be bothered with them [11].

3. Toward a Corporate Software Engineering Environment

It is anticipated that large software development projects will gradually evolve toward a growing set of tools that can be used from a variety of workstations with each workstation managing its own object-base. In this environment different members of a development project may invoke the same or different tools to operate on one or more of the same objects stored either on a local or a remote data repository. When multiple developers cooperate on a corporate software development project, they share a single conceptual (virtual) object-base physically dispersed between multiple client and server processes and workstations with each workstation possibly supporting its own OMS and object-base. Currently, interrelated projects have no means to remotely invoke tools or exchange objects

stored in (heterogeneous) data repositories. However, with true integration of multiple SEEs this is a very desirable feature: products generated by projects relying on different types of OMSs can be reused within a common network.

The inherent inability to access and integrate objects across a variety of SEEs, henceforth referred to as *distributed objects*, and tools stems from differences in structural and semantic content of similar, or almost similar, objects created by different tools in different OMSs. When integrating multiple SEEs it is reasonable to expect that models of different kind are used by their associated OMSs to represent the data relevant to the software development process. Obviously, the elementary manageable resources in the corporate SEE are the objects produced and managed by the individual OMSs. For reasons of simplicity we assume here that the individual OMSs in the corporate SEE support some variant of the object-oriented data model. In reality some of the OMSs may support filing systems, may be based on the relational or a navigational data model or variants thereof. The coexistence of two (or more) heterogeneous data models in an integrated environment preassumes that these models are in a position to acquire data from one another and subsequently convert these data to their internal representation format. This requires the ability to establish a common means of expressing software development activities within the corporate environment; to keep track of these activities; and finally, to provide support for distributed object management in order to allow multiple users and tools to work conveniently on the corporate object-space. Distributed tool sharing poses the problems of introducing various layers of distribution incompatibilities such as those hinted at in Section 2.2 and unfolded in the following:

- Incompatibilities due to the use of heterogeneous data models: differences in the representation of objects utilized by projects which rely on diverse data models. To bridge these differences a meta-model, or *canonical representation*, which comprises a super-set of the underlying data models, must be utilized. The purpose of the meta-model is to facilitate inter-OMS conversion: OMSs of one kind must be in a position to understand the type model of a different kind of OMS.
- Incompatibilities in object-base schema definitions: an additional problem is the inter-OMS differences encountered at the object-base schema level. Presenting the same kind of objects (local or remote) through different perspectives, requires a mechanism for keeping views and actions upon them consistent with respect to the base objects from which these views have been derived. This may be alleviated by adopting corresponding techniques from the database research field, namely distributed database and view integration methodologies [21, 22].

The problem of distributed object management and the resolution of its associated conflicts represents an active area of distributed systems [23] and distributed database systems research literature [24] and poses a series of technical issues and problems such as dynamic migration of objects, dynamic connections and

disconnections of workstations, effective communication protocols, concurrency and distributed transaction management and so on. Although the above issues represent challenging problems, attention will be focussed on the problems identified so far which in fact are the immediate derivatives of distributed object sharing in a corporate SEE. Additional information about the previous problems can be found in the information systems literature [21, 22, 25].

3.1. *Inter-OMS communication*

In the corporate SEE environment, tools are considered to be processes connected to and communicating via a common network. Communication can be performed by issuing remote procedure calls. For example, this kind of inter-tool communication can be achieved through a broadcast message server [26] where communication between tools that reside in remote workstations can be implemented by using TCP/IP-domain sockets.

Remote tools that work together share the behavior and the view of the object-base schema defined by their respective OMSs. We assume that each tool that can be remotely called is supplied with a strict interface which specifies the constructs that the tool imports or exports. Such constructs may include tools, types, relationships and rules. A tool may have its import ports connected to several export ports of remote tools and vice-versa. With this scheme name clashes between imported and local facilities are treated as different views of the same construct. The general objective here is to establish interconnection between tools and tool fragments in terms of services required and services rendered.

3.2. *Inter-OMS cooperation*

Even when inter-OMS communication is existent, it is extremely difficult to guarantee that meaningful network-wide references can be established from local tools and users to remote heterogeneous objects and services. Distributed tool sharing in a corporate SEE concerns itself with the sharing of a set of distributed objects which draw their data from diverse object-bases — physically situated at different workstations. To resolve the severe technical problems and diverse forms of incompatibility exhibited by the OMSs a novel architectural framework must be introduced. With this architectural framework distributed object management can be achieved whereby each local OMS will still be in charge of the objects and pre-existing applications developed on its workstation, and would cooperate with remote OMSs to achieve sharing of remote tools and objects.

The proposed approach attempts to address these issues by using a blend of object-oriented and knowledge-based techniques. It entails transforming the rather "passive" local OMS components into *active agents* by attaching to them active front-end extensions which embody sufficient contextual knowledge. Each local OMS will entail an *OMS (logical) integrator* and will accordingly be transformed into an agent which is able to reason about a corporate project domain, perform

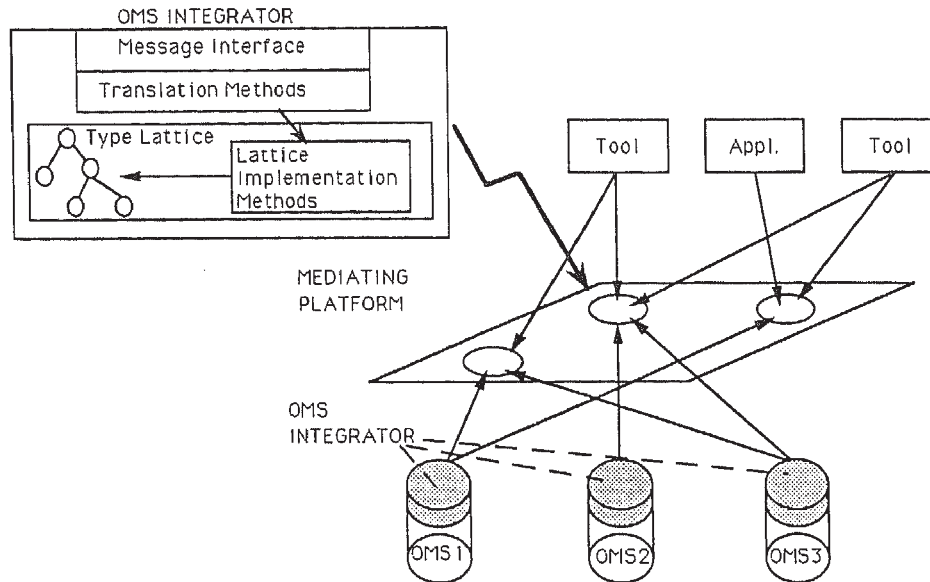

Fig. 4. Use of a set of uniformizing integrators between OMSs and tools

transformations which need to be applied to local OMS components, and automate many corporate software development activities. These advanced abilities grant the OMS the attribute of being *active*.

3.3. *Functional view of the corporate environmnet*

The functional view of the corporate SEE architecture entails a community of cooperating agents implementing an additional layer on top of the already existing local OMSs, i.e., OMS 1 to 3 in Fig. 4. This figure shows how the OMS integrator is interfaced with its underlying local OMSs. It is assumed that an OMS integrator (the shaded areas in the figure) will be attached as a front-end extension to each local OMS in the network and act as a mediator in order to provide information processing and requests addressing remote applications. Project management and administration of inter-project objects will not be performed on the premises of a single centralized OMS in the network. Rather, existing OMSs will cooperate, through their integrating component, and transparently access distributed heterogeneous objects residing at differently structured OMSs. The logical integrators are the only means by which local OMSs can communicate. The entire OMS integrator community may be perceived as forming a distinct mediating layer interposed between the diverse tools and object-bases, see Fig. 4. The mediating layer, or platform, makes the software developers' applications independent of the local or remote data resources and data management intricacies and is used as a vehicle for the unification of local and remote data and services.

Many of the near insurmountable problems which characterize corporate SEE object management would be solved if the tools in the different workstations were to share a single object representation employed by each OMS integrator. The idea of having all remote tools operating on the same data structures is very attractive as it guarantees full integration: with only a single representation of inter-OMS data, tools can communicate directly. The integration mechanism on the OMS integrating component performs several services such as keeping compatibility maps which describe how operations on data in one workstation OMS translate into equivalent operations on the same or equivalent data objects originating from remote workstations, i.e., the translation methods in Fig. 4. Moreover, the logical integrator uniformizes data, views and services provided by the diverse OMSs and makes them available to applications and tools operating on network-wide avaliable objects.

The corporate object space entails virtual objects (depicted as circles in the mediating platform) which draw their data from either remote or local address spaces. Such objects are described in the canonical representation scheme used by the logical integrators which employ a common object type lattice to define the overall corporate schema, see Fig. 4. Existing OMSs retain their autonomy in that local OMSs will not be tampered with. However, software developers who require remote object support will have to utilize the canonical data model to achieve this purpose. In all other circumstances users can still rely on their original data model for software development purposes.

As the logical integrator basically provides the same functionality that a distributed database management system provides in the traditional database world [21, 27] a detailed description of these and related technical issues is beyond the scope of this paper. In the following section we will confine ourselves to the issue of choosing the appropriate modeling capabilities to support inter-OMS integration and remote tool access and cooperation.

3.4. *The corporate SEE modeling requirements*

To choose the most suitable OMS approach for supporting corporate software development, the OMS conceptual data requirements must be identified first, exactly as it was done back in Section 2.3. The present section investigates these issues and then based on the desired capabilities, concludes that an enhanced object-oriented data model in the form of a tight coupling of data and knowledge based formalisms, as already implied in Section 2.3, is most appropriate for supporting corporate SEEs.

The overriding consideration behind the corporate SEE is the provision of a complete development system for practical complex design environments based on object-oriented technology. The logical integrator's extended functionality is driven by a supply of extensive domain and operational knowledge organized as a form of tight coupling between the AI-based knowledge oriented mechanisms and conventional database technology [28]. Such advanced capabilities include the advanced modeling primitives which facilitate the development of well formed interfaces to a

collection of heterogeneous information sources, and knowledge representation facilities (mainly through the use of the rule-based system) used to resolve the acute problem of data incompatibilities in the integrated application environment.

The usefulness of the typing and abstraction mechanisms (described in Section 2.3)are well recognized also for the case of distributed tool invocation and object management. For example, inter-type relationships offer very attractive features for OMS integration: capabilities that rely on relationships, such as reference and derivation [3], will ultimately facilitate inter-OMS integration by allowing users to interact with the environment at a high level, while leaving the intermediate mapping steps to be automatically determined. Through the establishment of inter-object relationships software objects in some space can refer to software objects in another: two source-code objects representing programs *P*1 and *P*2 can have references to objects in a common subroutine library which is represented by a separate object space. However, in our view the issues which are of paramount importance for materializing distributed tool invocation are the use of knowledge and meta-knowledge for OMS integration purposes.

Knowledge in the canonical model can be represented by means of rules which are used to represent inter-object dependencies. For example, one rule may state that a user object at a workstation-A can modify the remote parse tree represented by a program object at workstation-B. Another rule may state that a precondition for a user at workstation-A to resume execution of a program at workstation-B is that no changes have been made to any procedure already on the run-time stack. Rules will also be used externally and will not be embedded in the general tool functionality as suggested in [7, 11]. Adding this kind of knowledge to the logical integrator level would make its functionality relatively intelligent.

The logical integrator is designed not only to store the initial low level information, i.e. the objects, but also information used to facilitate corporate SEE tool invocation. This type of knowledge pertains to the structure and functionality of both local and remote objects and software development activities. The central repository of the logical integrator is a knowledge-base which contains information about inter-OMS structural and operational dependencies, application knowledge, software tool knowledge, and can also be used a vehicle for communication between the individual OMS components in the corporate environment. Software tool knowledge is expressed as a form of understanding of how local and remote software methods and tools tie together and is used as guidance to produce correct and effective products. In particular knowledge of this type entails information on how to construct and maintain the corporate object space and how to conduct controlled automation by forward and backward chaining among the rules defined by local or remote strategies. We collectively refer to this kind of knowledge, which for operational purposes needs to be shared by the entire OMS integrator community, as the *meta-knowledge* portion of the corporate SEE.

In sum, combining object-oriented data modeling primitives as described in this section and Section 2.3. provides a firm foundation for addressing the complex

distribution problems. Both paradigms increase software development productivity and reduce inter-object integration and maintenance costs. Object-orientation provides support for data abstraction, knowledge encapsulation, reusability, and extensibility while rule-based processing provides support for representing expertise and inferencing.

4. Conclusion

Until very recently software has been developed predominantly on large centralized computer configurations using a collection of tools bearing little or no relationship to one another. There has consequently been marginal support given to the software developers during the software life-cycle. It is against this background that the concept of SEEs has evolved. An SEE is a development evironment into which tools supporting all of the various phases of software life-cycle are integrated.

The important issues which we addressed in this paper were tool integration, SEE extensibility and interoperability. It was explained how the OMS supports internal tool integration and extensibility by providing advanced modeling primitives and by managing the properties of objects as part of the product management policies and the software development process rather than embedding them in the tools themselves. In this way a given tool may be used in different contexts with different policies and methods and this will ultimately facilitate software tools to be usable on a distributed network of SEEs. Given the trends toward greater portability and distribution, the value of tool integration will be the ability to create and manipulate objects according to common definitions and have distribution applicability across all tools no matter if they are local or remotely accessed in the network. Throughout this paper, we have concentrated on the extensibility and interoperability issues and identified several problems associated with these issues, raised some important open questions and suggested some promising research directions and methodologies.

Acknowledgments

We deeply appreciate the numerous excellent comments and suggestions made by Dan Cooke and Bernd Krämer. Their enthusiasm, help and interest is gratefully acknowledged.

References

1. M. H. Penedo, E. Pleodereder, I. Thomas, "Object Management Issues of Software Engineering Environments", *SIGSOFT* **13**, 5 (1988) 226.
2. A. van Lamsweerde et al., "Generic Lifecycle Support in the ALMA Environment" *IEEE Trans. Softw. Eng.* **14**, 6 (1988) 720.
3. S. Hudson and R. King, "The CACTIS Project: Database Support for Software Environments", *IEEE Trans. Softw. Eng.* **14**, 6 (1988) 709.
4. A. Wasserman, "The Architecture of CASE Environments", Technical Report, Interactive Development Environments. (1989).

5. V. Stenning, "On the Role of Environment", *9th Int'l Conf. on Software Engineering*, Monterey Ca., March 1987, p. 30.
6. E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, 1989.
7. M. Lacroix and M. Vanhoedenaghe, "Tool Integration in an Open Environment", *European Software Engineering Conf.: ESEC '89*, Coventry, England, p. 311.
8. R. Taylor et al., "Foundations for the Arcadia Environment Architecture", *SIGSOFT*, **13**, 5 (1988) 1.
9. S. Horowitz and T. Teitelbaum, "Relations and Attributes: A Symbiotic Basis for Editng Environments", *SIGPLAN Notices* (1985) 93.
10. B. Krämer and H. W. Schmidt, "Developing Integrated Environments with ASDL", *IEEE Softw. (Jan. 1989) 98.*
11. *G. Kaiser, P. Feiler and S. Popovich, "Intelligent Assistance for Software Development and Maintenance", IEEE Softw.* (May 1988) 40.
12. D. S. Wile and G. D. Allard, "Worlds: An Organizing Structure for Object-Bases", *SIGPLAN Notices* (Jan. 1987) 16.
13. E. Adams, M. Honda and T. Miller, "Object Management in a CASE Environment", *Procs. of the Software Engineering Conf.* (1989) 154.
14. D. Smith and P. Oman, "Software Tools in Context", *IEEE Softw.* (May 1990) 15.
15. B. Mayer, "Reusability: The Case for Object-Oriented Design" *IEEE Softw.* (March 1987) 50.
16. G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, 1991.
17. W. Kim and F. Lochovsky (Eds.), *Object-Oriented Concepts, Databases, and Applications*, ACM Press Books, 1989.
18. S. Reiss, "Working in the Garden Environment for Conceptual Programming", *IEEE Softw.* (Nov. 1987) 16.
19. D. Bobrow et al., "Common Loops: Merging Lisp and Object-Oriented Programming", *OOPSLA '86 Procs* (Oct. 1986) 17.
20. Special issue on "Object-Oriented Design", *Commun. ACM* **33**, 9, (Sept. 1990).
21. S. Ceri and G. Pelagatti, *Distributed Databases, Principles and Systems*, McGraw-Hill Book Co., 1984.
22. S. Navathe, R. ElMasri and J. Larson, "Integrating User Views in Database Design", *IEEE Comput.* (Jan. 1986) 50.
23. D. Decouchrant, "A Distributed Object-Manager for the Smalltalk-80 System" in Object-Oriented Concepts, Databases, and Applications, W. Kim, F. Lochovsky (Eds.), ACM Press Books, 1989.
24. P. Bernus and M. P. Papazoglou "Knowledge Based Architectures to Integrate Heterogeneous, Distributed Information Systems", *2nd Conf. on Tools for Artificial Intelligence*, Fairfax Va., Nov. 1990, p.682.
25. M. Manino and C. Karle, "An Extension of the General Entity Manipulation Language for Global View Definition", *Data Knowl. Eng.* **1**, (1986) 305.
26. M. Cagan, "HP Soft Bench: An Architecture for New Generation of Software Tools", *SoftBench Technical Note Series*, **SESD-89-24**, Hewlett-Packard, Software Eng. Systems Division, Nov. 1989.
27. M. P. Papazoglou, "Knowledge-Driven Distributed Information Systems", *14th Computer Software & Applications Conf. COMPSAC-90*, Chicago, Oct. 1990, p. 671.
28. L. Kerschberg, "Expert Database Systems: Knowledge/Data Management Environments for Intelligent Information Systems", *Inf. Syst.* **15**, 1, (1989) 151.