Designing the distributed architecture DIPS for cooperative software engineering

Distrib. Syst. Engng **4** (1997) 160–168. Printed in the UK PII: S0967-1846(97)82167-5

Designing the distributed architecture DIPS for cooperative software engineering

Daniel Scherer†, Tobias Murer‡ and Andy Würtz§

Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), CH-8092 Zurich, Switzerland

Abstract. Cooperative software engineering typically involves many actors and resources that cooperate in a complex distributed and heterogeneous world. In the DIPS (Distributed Integrated Process Services) project, a three-dimensional model is used for the definition, enactment and tracing of software development processes, which expresses both structure and evolution of such processes. This paper discusses how an optimal architecture was evaluated to implement the process model in a process support framework. Process-specific and general requirements are identified, and expected usage patterns of a DIPS-based environment are analysed. A set of potential architecture variants is proposed, and implications of the requirements and usage patterns on the variants are discussed qualitatively. An evaluation of the architecture alternatives leads to the design of the hybrid DIPS architecture based on distributed heterogeneous objects. The prototype DIPS implementation is briefly outlined.

1. Introduction

The emergence of distributed objects and component technologies together with decreasing communication costs allows for increasing numbers of applications to be run as widely distributed heterogeneous systems. Software engineering is increasingly regarded as a widely distributed cooperative process involving many actors and resources. Producers weaken their enterprise boundaries by sharing and linking their development processes, resulting in tightly cooperating networks of enterprises, so-called virtual enterprises. This, together with the globalization and increasing pace of software development requires highly integrated software development environments. An environment for software development processes must run on many heterogeneous platforms and integrate heterogeneous tools, and provide a homogeneous perception of the process to users [3, 4, 12]. The DIPS (Distributed Integrated Process Services) project (subproject of GIPSY (Generating Integrated Process support SYstems)) [8, 9] defines a formal process model for this purpose, and implements this model in a distributed framework for cooperative software engineering. It is based on an architecture of distributed objects which supports integration of heterogeneous tools on widely distributed platforms, and this paper presents the ideas leading to the design of the architecture.

† E-mail: scherer@tik.ee.ethz.ch
‡ E-mail: murer@tik.ee.ethz.ch
§ E-mail: wuertz@tik.ee.ethz.ch

After briefly introducing the DIPS process model, this paper deals primarily with the issue of how to map the model to a distributed architecture of a process framework that is suitable for implementation. It presents architectural requirements that are derived from the model, including usage patterns, and discusses how different architecture variants meet the requirements, thus leading to an optimal architectural choice. While the usage patterns are initially estimated, they have been confirmed by preliminary measurements performed with small processes, and future measurements will provide more exact data that will allow the prototype to be improved in a bootstrapping way.

The main focus is on features that distinguish different architecture variants, and, to a lesser degree, on features that are common to all designs, in order to be able to perform a meaningful comparison of variants to find the optimal one. The emphasis is on keeping the design simple, modular and extensible, and on finding the most necessary properties of such an architecture through a qualitative rather than a quantitative discussion. The analysis of requirements and usage patterns in absolute terms would provide a worthwhile exercise in itself but is beyond the scope of this paper—the assumption here is that upon implementation of the derived architecture, an adequate technology is chosen. The implementation of the DIPS architecture under construction based on a simple home-made object request broker (ORB) is presented, which may later be ported to a CORBA-compliant ORB [10] and possibly integrated in the WWW.

2. The DIPS 3D process model

The DIPS process model represents a formal model for software development processes [3], on which the DIPS process support framework is based [8, 9]. The framework can be used to create a specific environment by plugging specific highly integrated tools (used in individual process steps) into the framework.

2.1. Process structure

The working hypothesis for the process model is that all data (contract, specification, implementation, code, test case, manual, etc) produced during development and maintenance of a software product can be represented as a partially ordered set of objects (documents), created by different tools. Every object carries the information of part of the software product and typically depends on other objects. These dependencies define the ordering of the set. The definition of the process structure is then derived from the product structure, leading to a two-dimensional directed acyclic graph (DAG) whose edges represent the dependencies (figure 1). The nodes in the DAG consist of atomic processes representing individual objects, and of compound processes which contain other processes (atomic or compound). This defines a hierarchy of processes (sub- and superprocesses), with a compound process at the top, representing a software product's process.

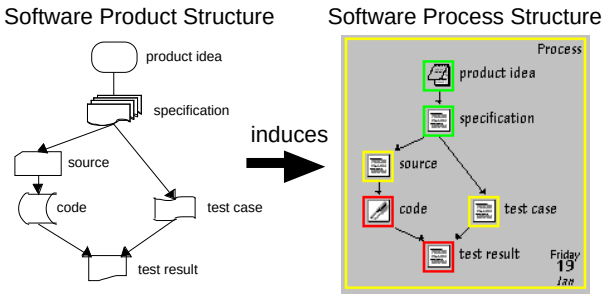

Figure 1. Product structure induces process structure.

Every atomic process carries one content attribute which represents its sole object under development. Objects are created and edited using highly integrated tools, and are not only simple documents but are entities in the object-oriented sense, i.e. consist both of data and of metadata, i.e. conceptually include the data's code (tool). Objects are therefore very heterogeneous. High-level semantic integration of tools is achieved by using extensible tool components created by using the tool generator of the associated CHIPS (Components for Highly Integrated Process Support) project. Every process has multiple structural attributes used to store dependencies and (for atomic processes) all information about the represented object except its content (completion state, type, access privileges, etc); structural attributes may be compared to directory information of a filing system.

During process enactment, the resulting parts of the emerging software product are stored in the (content attributes of the) atomic processes, and dependencies to neighbouring processes are validated. Processes can assume different states, represented by colours in the DAG. A process can attain at most four different states from creation until it reaches a non-modifiable state: planned, in work, confirmed and history (detailed below). The completion of a process requires a well-defined formal condition associated with the process to be fulfilled (confirmed); a dependent process can only be completed after the previous ones have been done; the colouring of the DAG thus illustrates the workflow. Checking these formal conditions may require the execution of evaluation procedures involving multiple processes and is only possible due to high-level semantic tool integration which prevents misinterpretation of data.

2.2. Process evolution

The process structure may change either when it is edited, i.e. when new processes and dependencies are added or existing ones are deleted, or when a rollback operation is performed to allow previously done process steps to be redone. In order to record the previous results, the process model provides a third dimension, the history dimension (figure 2). The process to be revised and all its dependants move down the history dimension, now creating a history layer (not modifiable any more), and a new part of the top (current) process layer is created. Different versions of a process may thus be found by navigating along the history dimension of a process. Since full versioning and dependency information is contained in the process structure, this enables straightforward integration of configuration management in an architecture that implements the process model.

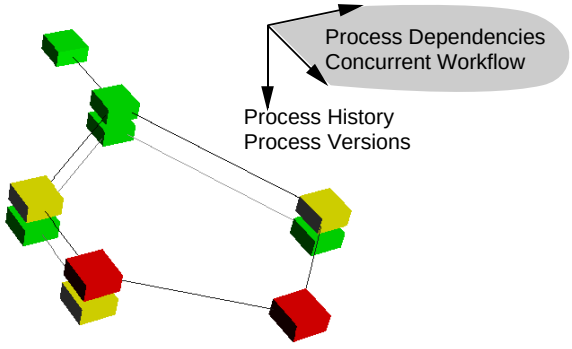

Figure 2. Process dimensions.

2.3. Process linking

A software product's development process often requires information about existing products. This fact may be reflected by installing links to other processes containing completed products, so that they are referenced. This powerful feature extends the scope of a process while ensuring that processes remain independent during definition and enactment. Likewise, the tools used to carry

out individual steps within a process may also be linked to their respective development processes. As a vision, the processes of many individual developers (enterprises) together may be regarded as one large global development process consisting of many linked processes.

3. Derived requirements

In order to implement a software engineering framework for processes defined by the DIPS process model, an architecture for such a framework must be defined. Features and requirements for the architecture as well as expected usage patterns may be derived from the process model. These may then be used to design an optimal architecture.

3.1. General features and requirements

An architecture for a software engineering framework based on the DIPS process model must provide support for distributed cooperative operation on multiple processes, allow process definition and subsequent editing, process enactment (execution, control, tracing), and process navigation and queries. It must provide storage and management of all process data (structure and content) with version and configuration management for objects and adequate levels of security, and it should be implemented in a modular and extensible way and be scalable to support widely distributed processes.

It needs to support heterogeneous objects and tools on heterogeneous platforms, and integration services for high-level semantic integration of CHIPS-generated tools, together with messaging among objects. Messaging allows for independence of objects from platforms, thus enabling heterogeneous objects; it allows for independence of objects and tools from the process framework; and it allows for independence of individual objects from each other. It requires a basic set of DIPS messages to be defined and adherence by the objects, tools and process support framework to the defined message protocols.

These qualitative features must be met by any design for an architecture. They are mandatory and serve to determine which architectures may be considered at all, but they do not provide any further distinction among the possible architectures; therefore, they are not discussed in greater detail here.

3.2. Specific requirements

There are a number of requirements that need to be met only to a reasonable degree, and the design of an optimal architecture should attempt to meet them as well as possible. These requirements include simplicity, reliability, availability, and, of greatest interest when comparing designs, appropriate efficiency.

Efficiency should minimize resource usage (memory, disk space, network bandwidth, time, etc), whereby time is the most important factor: while no absolute real-time limits need to be considered, the average time for an automatic operation to complete should be kept as low as possible; in particular, the more frequent an operation, the smaller the response time should be. An optimal architecture would be designed in such a way that most operations involve only the host where the operation originated or its local area network (whose delay is already noticeable under typical network bandwidth assumptions), and few involve communication over a wide area network (whose delay is tolerable for infrequent operations).

3.3. User operation types

In order to provide for an efficient design which should minimize the usage of resources, the operations to be performed when using a DIPS-based environment need to be analysed. This requires a definition of basic operation types, after which usage patterns may be established that characterize the number of such basic operations effected on specific data during a given time interval.

Performing distributed cooperative software engineering, multiple participants (users) work on different parts (subprocesses) of the process concurrently, but two cannot work on the same part at the same time. The envisaged process support framework provides graphical representations of the process structure to give an overview of structure and state of the process, and to facilitate process navigation and browsing (which are expected to be frequent operations). An operation executed by a participant originates on his part of the process, called ‘source’ with respect to that operation, and may involve other parts of the process, in the same compound process as the source (called ‘near’ logical distance) and/or any other parts of the process (called ‘far’ logical distance). All accesses to the content attribute of an atomic process are handled by the atomic process, involving some structural attributes too. Therefore, no operation can involve only content attributes. A message from a represented object in a content attribute to another such object is routed via their respective atomic processes.

User operations performed during development of a software product in a DIPS-based environment include process definition and enactment, editing and confirmation of individual objects, and navigation and browsing of processes and individual objects. From these, five types of basic user operations may be identified (which will be abbreviated), as follows:

(i) **Struc**: editing of structural attributes through process definition and enactment (including rollbacks). Operations which modify dependencies, states and other structural attributes of processes (except for operations of the next two types), which may involve multiple processes at once.

(ii) **Cont**: editing of content attributes of atomic processes. Operations which modify content attributes (representing objects) of atomic processes; these operations only involve one represented object at a time but may effect multiple read accesses to the content, and some also modify structural attributes such as states.

(iii) **Eval**: execution of evaluation procedures on content attributes. Operations which evaluate formal conditions associated with atomic processes (concerning the objects in the content attributes under development, which may effect read accesses to other such objects).

These may create new content information of atomic processes, and may thereby also modify structural attributes such as states.

(iv) **Nav**: navigation on process structure. Operations which only perform read accesses and only on structural process attributes, and may involve multiple processes at once.

(v) **Qry**: queries and browsing on process structure and content. Operations which only perform read accesses and which may involve accesses to any process attributes (structural and content) and to multiple processes.

Further operations may be envisaged which may be regarded as combinations of the basic operation types, e.g. the definition of a configuration may involve both navigation through processes and storage of the configuration in the process (process editing).

Since some operations involve only the structural process attributes and others involve both structural and content attributes, this suggests that an analysis of operations should distinguish between accesses to structural and to content attributes. Whether or not the distinction then is reflected in the architecture is an issue to be discussed in the architectural design.

3.4. Usage patterns

After defining basic operation types, usage patterns may be estimated by assuming a process of a specific size and quantitatively analysing a specific assumed load of operations performed by users on that process. The usage patterns are then established by careful estimation. Obviously, measurements could not have been performed before the prototype was built, so estimates were used as a basis for the usage patterns needed to design the architecture of the prototype, but thereafter the estimates may be tuned by performing measurements in order to improve the architecture in a bootstrapping procedure. The orders of magnitude of the usage patterns have been confirmed by preliminary measurements performed on small processes using a prototype environment under construction, and more thorough measurements are envisaged for the future.

3.4.1. Process size assumptions As a basis for a quantitative analysis, a large nearly-completed process containing a hierarchy of approximately 500 subprocesses is assumed, as follows:

- on average four history layers per process (the total number of history layers in the whole process hierarchy will be much larger than four, but not every process has an older version in every history layer);
- on average four subprocesses (atomic or compound) per compound process;
- 20 compound processes in the original process definition (i.e. original hierarchy without history).

Resulting process size:

$20 \times (1 \text{ compound} + 4 \text{ subprocesses}) \times (1 \text{ current} + 4 \text{ history}) \text{ layers} = 500 \text{ processes}.$

Further size measures:

- average size of one process (all structural attributes of an atomic or compound process, not counting content attributes or subprocesses): 1 kB;
- average size of one content attribute (representing one object): 50 kB;
- average number of participants working on a process, respectively workstations involved (one process hierarchy representing a software product): 10.

While larger processes are conceivable, it is more likely that such processes will be defined in individual parts, enacted independently and connected using the feature of process linking. It is, however, possible that multiple independent processes run simultaneously in one process support environment, but since primarily the relative numbers of operations within one process are of importance to the design, multiple processes are not considered further here.

3.4.2. Analysis For a quantitative analysis, an estimated maximum load of user operations performed during one day on one process of the size described above is assumed. While the chosen time unit of one day should allow for a reasonably large number of operations, the time unit is of little relevance insofar as the focus of this analysis is on a relative comparison of the number of reads and writes effected by user operations, and less on absolute numbers. The consideration of relative numbers allows for architectures to be compared and an optimal one to be found with respect to requirements such as efficiency.

In table 1, the estimated effects of the above-mentioned load of user operations on the numbers of reads and writes are presented, distinguished by structural and content attributes of processes, and by the logical distance of the processes involved. The numbers represent the product of the number of operations times either the number of processes with involved structural attributes or the number of content attributes concerned by an individual operation.

3.4.3. Results Since the numbers only represent estimates, they should only be used to draw qualitative conclusions. The following five results may be gained by studying the table, and these should be considered in the design of an architecture (whereby the second and third results have been derived together with the first one).

(i) Read accesses to structural attributes are much more frequent than any other read or write accesses.

(ii) Read accesses to structural attributes are much more frequent than to content attributes.

(iii) For structural and content attributes, reads are much more frequent than writes.

(iv) Write accesses to content attributes only occur where the effecting operation originates.

(v) For structural and content attributes, for reads and writes, access frequencies to structural or content attributes are greater for ‘near’ logical distance than for ‘far’ logical distance.

Table 1. Reads and writes effected by user operations. **Key:** • Struc, Cont, Eval, Nav, Qry: five basic types of user operations as detailed above; • Reads: number of effected individual read operations; • Writes: number of effected individual write operations; • Struc Reads/Writes: involving one process' structural attributes; • Cont Reads/Writes: involving one content attribute; • Src: source process where operation originated; • Near: process in same compound process where operation originated; • Far: any other process in the process hierarchy.

Reads and writes			User operations					Total R & W
			Struc	Cont	Eval	Nav	Qry	
Reads	Struc	Src	50	100	50	500	100	800
		Near	100		50	5000	1000	6150
		Far	20		10	2000	300	2330
	Cont	Src		100	50		50	200
		Near			50		200	250
		Far			10		50	60
Writes	Struc	Src	20	20	20			60
		Near	40					40
		Far	10					10
	Cont	Src		50	10			60
		Near						0
		Far						0

4. Design of architecture

In order to perform cooperative software engineering in an environment based on the DIPS process model, its framework's architecture needs to be a distributed one which enables multiple participants to work concurrently on individual workstations [3, 4, 7]. The architecture is designed by considering the necessary requirements and primarily by using the results gained from the analysis of the estimated usage patterns to compare a representative selection of potential architectures. After briefly discussing different types of architectures in general, the focus is on architectural variants distinguished by storage location of data.

4.1. Architecture types

A distributed architecture may be either object-based or not. A non-object-based solution such as a distributed filing system signifying that code and data are stored separately would not support integration of heterogeneous objects and tools very well and is therefore not considered here. Object-based solutions would provide for all process data (structure and content) to be stored in the form of objects, which is preferable considering the heterogeneous types of data involved.

One object-based solution is an object database (ODBMS) [2]. An ODBMS provides a high abstraction level for objects, including access to objects through protected actions having ACID properties [1, 6]. However, the protocols ensuring ACID properties demand considerable resource usage (particularly communication bandwidth) that prevent scalability of the architecture to world-wide dimensions. Assuming an ODBMS client/server architecture using a central storage server, access to objects by clients typically involves (costly) transferral of the objects to the clients to execute them locally (which demands a homogeneous environment). Support for heterogeneous platforms, tools and objects is generally difficult to achieve. An ODBMS is more suitable for millions of fine-grained (primarily homogeneous) objects than for smaller numbers of larger heterogeneous objects as they are typically required in a process support environment.

Another object-based solution is an object bus (object request broker, ORB) architecture. This is the more universal and flexible approach; it still allows access to databases by means of object adapters where necessary [10]. All objects are accessed via messages on the machine where they reside, i.e. they remain on the host where they are able to live. This enables integration of heterogeneous tools and objects used on heterogeneous platforms by utilizing object request brokers and object adapters that handle the desired protocols and messages. Transferral of messages instead of whole objects requires less communication bandwidth, thus providing a greater possibility of scalability to large dimensions. While the abstraction level of objects is lower than in ODBMSs and access to objects through protected actions would have to be provided by additional object services, this also provides the opportunity to utilize leaner protocols, as full ACID properties are generally not required in a process support environment. An ORB architecture type is therefore chosen for its framework.

4.2. Architecture variants

The most distinctive feature of a distributed architecture represents the storage location of data, therefore this represents the prime feature for distinguishing architecture variants. In order to be able to perform meaningful comparisons, three extreme architectures are chosen: a client/server solution with one central server, a fully distributed (non-replicated) architecture, and a fully replicated architecture. Advantages and disadvantages are qualitatively compared and subsequently summarized in a table, and a hybrid architecture is derived from this which attempts to combine properties of different variants in an optimal manner.

4.2.1. Client/server architecture The simplicity requirement suggests that all process data should reside on one central storage server, which also provides for excellent data consistency at no extra cost. All (other) participants' hosts permanently assume the role of client (in contrast to general object bus architectures, where roles alternate). All data are accessed on the single ORB server via messages, which may require considerable communication bandwidth (particularly for messages involving large contents), providing low efficiency and possibly even leading to congestion. Heterogeneous objects are unlikely to be supported well since they all have to live on the same (server) platform. Redundancy and availability can be improved at a small cost since it is relatively easy to perform periodic backups as only one data source is involved, and a backup server could take over as a whole whenever the central server is down.

4.2.2. Distributed architecture The realization that every participant in a software development process works on a different part of the process suggests that process data should be distributed in such a way that every participant's part resides on his workstation (respectively the nearest machine that always remains on-line, if personal workstations are shut down while not in use). The process data are distributed among all participating hosts without replication. In this way, support for heterogeneous objects is easily provided, and the design remains relatively simple. However, redundancy and availability can only be provided at a considerable cost, as a backup would need to be performed at every host. Even though failure of a host would only concern a small part of the process data, the loss of some structural process information could result in large parts of the process hierarchy becoming inaccessible. (Imagine losing some directory information in a hierarchical filing system!) Data consistency is easily achieved except when data have to be moved due to a failure. In order to assess read and write efficiency, a fair assumption (derived from result (v)) is that most processes will be defined in such a way that the logical distance (source/near/far) of processes correlates with the geographical distance (of hosts where processes are stored)—this leads to an increasing access efficiency with decreasing logical distance (accesses occur via messages forwarded by ORBs).

4.2.3. Replicated architecture Result (iii) stating that in general reads are much more frequent than writes suggests that all process data should be replicated on all hosts, so that every host has a complete set of all process data, resulting in excellent redundancy and availability, but high usage of disk space. If every object is replicated on every host and must be able to live there, heterogeneous objects and platforms are virtually impossible. The ORBs will handle all read access messages locally (very efficiently), but all writes involve costly broadcasts of update messages to all other hosts; these may include large content messages. Write efficiency values here include the cost of keeping replicated data consistent, and complicated protocols are required to ensure a tolerable level of data consistency.

4.3. Deriving the optimal architecture

In order to be able to derive an optimal design by combining the three extreme architectural variants, assessment criteria must be defined, i.e. the relative importance of various features and requirements needs to be specified. Hereby, the list of general features needs to be met by any design, which means in particular that support for heterogeneous objects has a high priority. Of the general requirements, efficiency is regarded as the most important one. The results of the usage patterns should be considered in a design in order to provide appropriate levels of efficiency. Of lesser importance are simplicity of design, reliability and availability, where reasonable levels should be achieved. The requirement of reliability is met in part by providing appropriate degrees of consistency and redundancy. Write operations should ensure that consistency of replicated data is provided at a minimal delay.

4.3.1. Hybrid architecture The hybrid design attempts to combine the three extreme architectures in such a way that the individual advantages outweigh the disadvantages, i.e. by utilizing the best features of every design. Realizing that structural information is read very frequently (result (i)) and that this is supported optimally by the replicated architecture, this is chosen for the structural data. Obviously, then for the structural write efficiency values those from the replicated architecture must also be applied, even though here the distributed architecture seems more favourable, but result (iii) stating that reads are much more frequent than writes confirms the better choice. To store the content information, the distributed architecture is preferred over the replicated one, due to the essential support for heterogeneous objects, the large size of content information which would result in inefficient replicated writes, result (iv) stating that content data are (logically) only written locally, and result (ii) implying that even reads (let alone writes) of content information are not so frequent that the added overhead of replication is justified.

This leads to a design where the process structure is replicated on all hosts, and the content data are distributed on all hosts without replication, i.e. every host carries a part of all content data. This involves separating storage of the content and the structural attributes of an atomic process (since the structural data are replicated and the much larger content data are not)—therefore, the content attributes exist as individual (heterogeneous) objects. Objects are accessed via DIPS messages (forwarded by ORBs) on the host where they are stored persistently and where they are able to live, and they merely need to fulfil the DIPS message protocols (to be defined) in order to be able to participate in processes.

The only two features where the client/server design is optimal can only be partly considered in the hybrid design (signified by lighter cell shading in the table of figure 3), as the hybrid one represents a compromise of the three extreme designs. Following the idea of central control in the client/server design, the concept of a *master* host is incorporated in the hybrid design: all updates of replicated data are coordinated by and broadcast from one so-called master host, leading to simpler protocols to ensure

data consistency, although the optimal levels of the other designs are not achieved. Furthermore, considering that the total data size of the structural process information is much smaller than the total content data size, simpler protocols suffice for the replication of only the structural information than for the fully replicated design.

The crucial and frequently accessed structure information (see result (i)) is replicated on all hosts providing excellent read efficiency, availability and redundancy for this data. This also allows another host to take over as master relatively easily if the previous master fails. The less essential and much larger content data are not replicated, but following the idea of central data location in the client/server design, the content data are backed up on the master host in order to provide some redundancy in case of failures.

It is conceivable that in future further optimizations are possible, for instance on the one hand it might be beneficial to replicate some objects, particularly those which have attained a non-modifiable state, on some hosts, while on the other hand it might not be desirable to replicate the complete process structure on all hosts involved. Also, more flexible solutions are envisageable where it would be possible to adapt the replication of data dynamically at runtime in order to optimally support changing requirements—this would, however, increase complexity and require measurements to be performed and acted upon continuously, and the evaluation principles would be similar to the static case and are therefore not investigated further here.

Features		Architectures	Client/ Server	Distrib	Replic	Hybrid
Data Location		Central	x			x
		Distrib		x		x
		Replic			x	x
Read Efficiency	Struc	Src	0	++	++	++
		Near	0	+	++	++
		Far	0	0	++	++
	Cont	Src	-	++	++	++
		Near	-	+	++	+
		Far	-	0	++	0
Write Efficiency	Struc	Src	0	++	-	-
		Near	0	+	-	-
		Far	0	0	-	-
	Cont	Src	-	++	--	++
		Near	-	+	--	+
		Far	-	0	--	0
Heterogeneous Objects			--	++	--	++
Consistency			++	+	--	0
Redundancy			-	--	++	+
Availability			-	--	++	+
Simplicity			++	+	--	0

Key			
x	applicable	0	medium
++	excellent	-	poor
+	good	--	very poor

Features:

- Data Location: central, distributed, or replicated;
- Efficiency: efficiency of read or write (write includes consistent update of replicated data where applicable), distinguished by structural and content process attributes, and by logical distance from effecting operation (source/near/far).
- Heterogeneous Objects: degree to which these are supported;
- Consistency: ease by which data consistency is provided;
- Redundancy: degree and ease to support redundancy;
- Availability: degree and ease to support high data availability;
- Simplicity: overall simplicity of design.

Figure 3. Distinctive features of architectures.

4.4. Summary of architectural discussion

The table given in figure 3 summarizes the features distinguishing different architecture variants. Shaded cells in the three extreme variants show what features have been incorporated (to a certain degree, as a compromise) from the respective variant in the resulting hybrid architecture.

4.4.1. Illustration Figure 4 illustrates the hybrid architecture by a small example of two processes, each consisting of one compound process in the current layer only. Process A resides only on one host; process B (which has one history layer) is replicated on two hosts, while the individual distributed objects of its atomic subprocesses each reside only on one host. (Backups of distributed objects on a master host are not shown.) Processes A and B are linked.

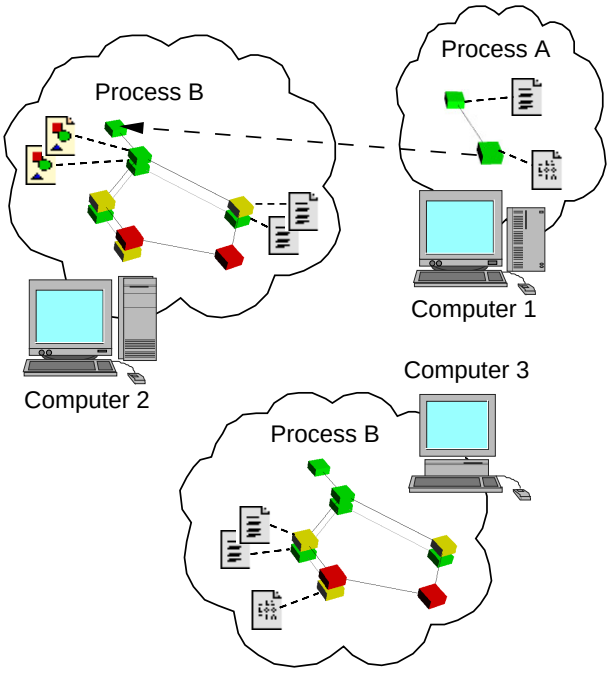

Figure 4. Hybrid architecture.

5. Implementation

5.1. Prototype

A prototype is under construction which implements the hybrid architecture: the process structure is replicated and

the content data are stored as non-replicated distributed objects. It is based on Oberon System 3 [5] (and uses the Oberon programming language), which provides advantages for rapid construction of a prototype system (type-safe language, dynamic linking and loading, simple but powerful system, Gadgets user interface, persistent objects, portability, . . .).

A simple home-made object request broker (ORB) architecture for distributed objects has been implemented, using TCP/IP for communication. As all messages among objects are routed via the process structure, the full complexity of a CORBA-compliant general-purpose ORB is not required. Heterogeneous objects will use object adapters that understand a basic set of DIPS messages (to be defined); general interoperability with IDL-specified objects is not planned for the prototype. A distributed process engine is available which is based on the hybrid ORB architecture, and which provides process navigation/browsing, definition and enactment, and simple version and configuration management, although not all operations are supported as distributed operations yet. Accesses to processes and objects are planned to be protected by simplified transactions (including long-duration ones) having ACID-like properties, although not to the full extent—e.g. no logging is planned to support durability; these parts are in design or under construction; services similar to but simpler than CORBA services are envisaged (persistence, concurrency control, transaction, etc).

The prototype has already been applied successfully to support small processes at exercises of lectures at ETH, and it also supports planning and progress of student projects. As further parts become operational, it is intended to be used to support larger processes including its own development process, and more measurements of usage patterns are to be conducted in order to further the improvement of prototype versions in a bootstrapping way.

5.2. Evolution to a CORBA-based architecture

In order to be able to provide more of the features which are required for large-scale or commercial operation, but which are not well supported by the prototype (security, reliability, etc), the transition to a CORBA-based architecture seems an attractive option in the longer run [10]. This would enable existing services to be utilized (persistence, concurrency control, transaction, query, security, etc), it would facilitate interoperability with existing IDL-specified components, and allow the implementation efforts to concentrate on the actual process engine. A replication service for CORBA is however not yet available.

The currently used ORB already features execution semantics for ORB operations inspired by those provided by CORBA, which should facilitate the transition. The transition would imply exchanging the underlying ORB and services with CORBA-compliant ones, and possibly porting the DIPS process framework to a programming language and system where CORBA is available. Note that it is not currently possibly to directly provide a CORBA-compliant ORB for Oberon-implemented objects, as OMG's CORBA specification does not yet provide an Oberon language mapping.

5.3. Outlook: integration in the WWW

A further possibility would be the integration of the DIPS process framework in the WWW, in order to facilitate widely distributed cooperative software engineering by providing an attractive homogeneous perception of heterogeneous process data while requiring little administration effort for participants on heterogeneous platforms. This could be achieved by making use of Sun's Java language and toolkit (supported by various WWW browsers), in particular by using a Java-based CORBA-compliant ORB such as Sun's Joe, and by porting the DIPS process framework to Java. A DIPS-based environment could thus be downloaded automatically from the WWW by clients in a bootstrapping manner requiring little administration effort. A first step towards WWW integration has already been investigated [11] by presenting processes on the WWW, i.e. process information can already be transferred to the WWW automatically, but participation is not yet possible via the WWW. The WWW could facilitate access to information about software components required for a successful electronic marketing of components, such as component interoperability information extracted from processes.

6. Summary and conclusions

The DIPS project focuses on distributed cooperative software engineering by providing distributed process definition, enactment and tracing, and it supports high-level tool integration. A simple three-dimensional model to express structure (two dimensions) and evolution (third dimension) of software processes has been introduced, and the model has been used as the underlying concept of the DIPS framework prototype. Implementation and employment of a prototype will allow the DIPS process model to be validated. The evaluation process for the optimal system architecture for the specific process model has been explained. An object request broker architecture for distributed objects provides the appropriate technology for widely distributed cooperative software engineering. Its support for heterogeneous objects, scalability, and modularity (components and services) is an essential feature required by a highly integrated process support environment, in particular heterogeneous objects are necessary for the integration of heterogeneous tools.

Typical expected usage patterns have initially been derived from the process model by estimation, and the architecture is designed by considering the process model, the analysis of the usage patterns, a representative selection of potential architecture paradigms, and specified requirements. The chosen hybrid approach combines the advantages of the different architecture paradigms with respect to the specific requirements, and it is believed that the optimal architectural choice is achieved in this way. It is based on distributed objects and features replication of the frequently read process structure and distribution of

the heterogeneous tool-dependent objects that make up the software product being developed in the process.

A prototype is being implemented based on a home-made object request broker architecture and has been applied successfully to support small processes, and the chosen design has also been validated by performing preliminary measurements on usage patterns which have confirmed the estimates' orders of magnitude. Larger more applicative process examples may be studied upon completion of the prototype in order to carry out more detailed measurements and emphasize the benefits of the proposed design. The prototype is to be applied in a bootstrapping process to develop improved versions of itself, and the results of the evaluation process may also be applied to create more flexible designs for dynamic replication of data.

Acknowledgments

The GIPSY project (which includes the projects CHIPS and DIPS) has been funded in part by the Swiss Priority Programme (SPP) Informatics Research of the Swiss National Science Foundation.

References

[1] Bernstein P A, Hadzilacos V and Goodman N 1987 *Concurrency Control and Recovery in Database Systems* (Reading, MA: Addison-Wesley)

[2] Cattell R G G 1994 *Object Data Management* (Reading, MA: Addison-Wesley)

[3] Finkelstein A, Kramer J and Nuseibeh B (ed) 1994 *Software Process Modelling and Technology* (New York: Wiley, Research Studies Press)

[4] Garg P and Jazayeri M 1995 *Process-Centered Software Engineering Environments* (Los Alamitos, CA: IEEE Computer Society Press)

[5] Gutknecht J 1994 Oberon System 3: vision of a future software technology *Software Concepts Tools* **15** 45–54; see also *Oberon System 3 home page* http://www-cs.inf.ethz.ch/Oberon/System3.html

[6] Gray J and Reuter A 1993 *Transaction Processing: Concepts and Techniques* (San Mateo, CA: Morgan Kaufmann)

[7] Mullender S 1993 *Distributed Systems* (New York: ACM Press)

[8] Murer T, Würtz A and Scherer D 1996 A 3D model for a common understanding of the software process *Proc. Asia–Pacific Conf. on Computer Human Interaction (Singapore, 1996)* pp 318–23

[9] Murer T, Würtz A, Scherer D and Schweizer D 1996 GIPSY: Generating Integrated Process support SYstems *TIK-Report no 22* TIK Laboratory, ETH Zurich; see also *Project GIPSY home page (with subprojects CHIPS and DIPS)* http://www.tik.ee.ethz.ch/~gipsy/

[10] Orfali R, Harkey D and Edwards J 1996 *The Essential Distributed Objects Survival Guide* (New York: Wiley)

[11] Scherer D, Murer T and Würtz A 1996 Towards providing software component interoperability information on the WWW *Proc. 2nd Australian World Wide Web Conf. (Gold Coast, 1996)* pp 109–16; also at http://www.scu.edu.au/ausweb96/tech/scherer/

[12] Sharon D and Bell R 1995 Tools that bind: creating integrated environments *IEEE Software* **12** (2) 76–85