Doctoral Thesis

Internet-wide software component development process and deployment integration

Author(s):
Scherer, Daniel Bernhard

Publication Date:
1998

Permanent Link:
https://doi.org/10.3929/ethz-a-002031325

Rights / License:
In Copyright - Non-Commercial Use Permitted
INTERNET-WIDE
SOFTWARE COMPONENT DEVELOPMENT PROCESS
AND DEPLOYMENT INTEGRATION

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
DOCTOR OF TECHNICAL SCIENCES

presented by
Daniel Bernhard Scherer
Dipl. Informatik-Ing. ETH
born February 3rd, 1968
citizen of Lucerne LU

accepted on the recommendation of
Prof. Dr. Albert Kündig, examiner
Prof. Dr. Hanspeter Mössenböck, co-examiner

1998
Acknowledgements

I would like to thank Prof. Dr. Albert Kündig for being my thesis supervisor and for providing a liberal research environment. I would also like to thank Prof. Dr. Hanspeter Mössenböck for accepting to be my co-advisor. My thanks go also to my colleagues who have participated in the GIPSY project or have made use of it, Reto Marti, Tobias Murer, Andreas Würtz, Andreas Helbling, and Daniel Schweizer. I am also grateful for the funding provided in a large part by the Swiss National Science Foundation (SNF) in the Swiss Priority Programme (SPP) Informatics Research (grant no. 5003-039002) [SPP94].
Leer - Vide - Empty
Abstract

This thesis presents a new approach to support the complete lifecycle of originating, evolving and executing software systems and their components in a global heterogeneous world. Development and deployment issues are integrated in a unified and as yet unique concept of broad scope, which also differs from others in significant aspects of integration and interoperability. Specifically, the thesis presents a comprehensive yet simple graphical modeling language for software development processes, and the design and implementation of a distributed framework for both software engineering environments and software deployment systems that implements the common language.

The language supports management of development processes for software components guided by the coarse-grain structure of the evolving software product. It supports planning and recording of the structure and its evolution in a multi-dimensional graphical form holding the artifacts of the product and their dependency, origin, version, and other metainformation. It integrates process definition and enactment, product and process evolution, repository issues such as version and configuration management, and metaprocess issues in a common yet open form. It allows usage of arbitrary tools in individual process steps and optionally supports formal methods to specify semantics of artifact dependencies on a high data integration level. While the emphasis is on simplicity, openness and interoperability, unlike other languages it extends the scope of individual distributed processes in two significant aspects: Different processes may be linked together on a global scale to reflect component dependencies, and the scope of processes is also extended beyond the development phase to support deployment activities. This allows developers to record and to obtain essential dependency and history information of related components not normally managed and to make this interoperability information available to users, who may apply it to install appropriate valid configurations involving numerous dependent components.

The distributed framework consists of a common process support system based on heterogeneous distributed objects implementing the fundamental language properties used by both development and deployment activities, and separate enhancements for development and deployment. For developers, it is extended by a distributed process definition and enactment engine as well as process-specific tools to make up a software engineering environment, allowing distributed development, evolution, configuration and release of software in globally linked processes. The process structure is replicated and synchronized on developers' distributed
hosts, while individual heterogeneous artifacts remain local during development. For users, the common system is extended by a deployment and runtime system that allows distributed retrieval, local installation and invocation of software, and manages user-site process bases that hold installed software as process aware artifacts instead of as conventional files. Developers publish the completed parts of their development processes holding software components via Internet and intermediary release servers, and users copy such linked parts from different developers’ processes containing related components to their local process bases to make up their individual composition of software applications. Since deployed software retains its process awareness, users may check installed configurations for compatibility using process metainformation.

Different implementation possibilities are presented, from a portable interoperable system to a fully integrated one. An experimental prototype has been implemented in Oberon System 3 to demonstrate the system’s feasibility, including WWW integration. While the concepts constitute a major paradigm shift, they seem technically promising and commercially attractive and could inspire new standards for globally distributed interoperable development and deployment of heterogeneous software.
Zusammenfassung


Das verteilte Framework besteht aus einem gemeinsamen Prozessverwaltungssystem, basierend auf heterogenen verteilten Objekten, das die grundlegenden Spracheigenschaften implementiert, die sowohl

Diverse Implementationsmöglichkeiten werden aufgezeigt, von einem portablen interoperablen System bis zu einem vollintegrierten System. Ein experimenteller Prototyp wurde in Oberon System 3 gebaut, um die Machbarkeit zu demonstrieren, einschliesslich WWW Integration. Obgleich die Konzepte einen bedeutenden Paradigmenwechsel darstellen, erscheinen sie technisch vielversprechend und kommerziell attraktiv, und sie könnten neue Normen für die global verteilte interoperable Entwicklung und den Einsatz heterogener Software inspirieren.
Contents

ACKNOWLEDGEMENTS ................................................................. i

ABSTRACT .................................................................................. iii

ZUSAMMENFASSUNG................................................................. v

CONTENTS .................................................................................... vii

1 BACKGROUND AND MOTIVATION ......................................... 1
  1.1 THESIS OVERVIEW ............................................................. 1
  1.2 SCOPE AND MOTIVATION .................................................... 1
    1.2.1 Software Component Lifecycle ...................................... 2
      1.2.1.1 Software Development .......................................... 5
      1.2.1.2 Software Deployment ........................................... 6
    1.2.2 Software Component Interoperability Levels .................... 7
    1.2.3 Background and Related Issues ..................................... 9
      1.2.3.1 Software Engineering ........................................... 10
      1.2.3.2 Software Component Reuse .................................... 11
      1.2.3.3 Software Quality ................................................ 12
      1.2.3.4 Distributed Component and Internet Technology ........ 13
      1.2.3.5 Virtual Organizations and Global Distribution .......... 14
      1.2.3.6 Software Development Processes and Process Models .. 14
      1.2.3.7 Process-Centered Software Engineering Environments. 17
      1.2.3.8 Software Deployment Systems ................................ 19
      1.2.3.9 Configuration Management ................................... 19
    1.2.4 Conventional Requirements ......................................... 20
  1.3 OBJECTIVES ......................................................................... 21
    1.3.1 Design Objectives ...................................................... 24
    1.3.2 Design Non-objectives ............................................... 25
    1.3.3 Realization Objectives ............................................... 25
    1.3.4 Realization Non-objectives ......................................... 25
  1.4 CONTRIBUTIONS ............................................................... 26
  1.5 PROJECT HISTORY ............................................................. 27

2 PROCESS MODELING LANGUAGE .......................................... 31
  2.1 PROCESS MODELING LANGUAGE OBJECTIVES AND REQUIREMENTS ..... 31
    2.1.1 Process Modeling Language Paradigms ............................ 31
    2.1.2 General Process Modeling Language Properties and Objectives .. 33
    2.1.3 Project-specific Objectives ......................................... 34
2.2 CONCEPTS .............................................................................. 36
  2.2.1 Software Product ................................................................. 36
  2.2.2 Software Process ................................................................. 37
  2.2.3 Metaprocess ........................................................................ 39
2.3 PROCESS DEFINITION .............................................................. 40
  2.3.1 Dependency Semantics ......................................................... 42
  2.3.2 Definition Operations and Rules .......................................... 46
2.4 PROCESS ENACTMENT .............................................................. 47
2.5 PROCESS DIMENSIONS ............................................................. 50
  2.5.1 Process Iteration ................................................................. 52
  2.5.2 Process Evolution ................................................................. 53
  2.5.3 The Third Dimension .......................................................... 53
2.6 PROCESS LINKING ................................................................. 58
  2.6.1 Interprocess Dependencies .................................................. 58
  2.6.2 Metaprocess Dependencies .................................................. 60
2.7 CONFIGURATION MANAGEMENT ............................................. 62
  2.7.1 Configurations .................................................................... 62
  2.7.2 Configuration Validation ...................................................... 64
  2.7.3 Configuration Sets ............................................................... 65
2.8 WORKSPACES AND ACCESS PRIVILEGES .................................. 66

3 FRAMEWORK ARCHITECTURE DESIGN ................................. 67
3.1 OBJECTIVES AND REQUIREMENTS ......................................... 67
  3.1.1 Integration Types ............................................................... 67
    3.1.1.1 Data Integration .......................................................... 68
    3.1.1.2 Process Integration ....................................................... 69
    3.1.1.3 Organizational Integration .............................................. 69
  3.1.2 Process Modeling Language Requirements ............................ 69
3.2 ARCHITECTURE LAYERS .......................................................... 71
  3.2.1 Overview .......................................................................... 71
  3.2.2 Basic and Distribution Services .......................................... 72
  3.2.3 Common Distributed Process Services ................................. 72
  3.2.4 Development System ......................................................... 73
  3.2.5 Deployment and Runtime Systems ....................................... 74
  3.2.6 Tool Integration Services ..................................................... 75
3.3 DISTRIBUTED DEVELOPMENT FRAMEWORK ......................... 76
  3.3.1 Overview .......................................................................... 76
  3.3.2 Development Activities ...................................................... 76
    3.3.2.1 User Operation Types .................................................. 76
    3.3.2.2 Usage Patterns ............................................................ 78
  3.3.3 Design of Architecture ....................................................... 79
    3.3.3.1 Architecture Types ...................................................... 79
    3.3.3.2 Architecture Variants .................................................... 80
    3.3.3.3 Deriving the Optimal Architecture ................................. 82
5 DISCUSSION ................................................................. 147

5.1 EXPERIENCES .......................................................... 147
5.2 EXAMPLES ............................................................... 148
  5.2.1 Process Browsing .................................................. 149
  5.2.2 Process Definition ................................................. 149
  5.2.3 Process Enactment ................................................. 150
  5.2.4 Variants ............................................................... 151
  5.2.5 Process Iteration .................................................. 152
  5.2.6 Process Evolution ................................................. 153
  5.2.7 Configurations ..................................................... 154
  5.2.8 Release ............................................................... 154
  5.2.9 Retrieval and Installation ...................................... 155
  5.2.10 Loading and Linking ............................................. 156
  5.2.11 Removal ........................................................... 156
5.3 RELATED WORK ........................................................ 156
  5.3.1 Systems and Projects ............................................. 156
    5.3.1.1 Process-centered Software Engineering Environments .. 156
    5.3.1.2 Configuration Management Systems ........................ 158
    5.3.1.3 Software Deployment Systems ............................... 159
  5.3.2 References and Standards ...................................... 160
    5.3.2.1 ECMA Framework Reference Model ......................... 160
    5.3.2.2 PCTE Open Repository Standard ........................... 161
5.4 COMMERCIAL OPPORTUNITIES ..................................... 161
5.5 CONCLUSIONS AND OUTLOOK ...................................... 164

REFERENCES ........................................................................ 169
Chapter 1
Background and Motivation

1.1 Thesis Overview
This thesis presents issues in the DIPS (Distributed Integrated Process Services) software engineering research project carried out at the TIK Laboratory of ETH Zurich. The title “Internet-wide Software Component Development Process and Deployment Integration” indicates that an integrated system is conceived that handles both development process and deployment issues for software components, allowing for globally distributed operations such as process definition, enactment, and evolution, and deployment of process aware software, both for distributed processes and for linking of multiple processes. “Internet-wide” also specifically means that simple scalable protocols and URL-like addressing are employed to support global distribution, WWW integration is envisaged and TCP/IP is employed. Although the term process has many meanings in computer science, in this thesis it stands for software development process (also known as software process), which is a set of partially ordered steps intended to reach a goal [FH93], the goal being the realization of a software product and the steps describing its development.

The thesis is structured in five main chapters. First, as a motivation, specific problems in current software engineering practice are identified and discussed, and our objectives in addressing them are presented together with our contributions. Then, our proposed process modeling language is presented as a formal basis for our approach, then the architecture design of a framework for a system implementing the process modeling language is presented, followed by a presentation of how the framework is implemented in our prototype system. Finally, the presented system and its results are discussed with examples and comparisons to related work.

1.2 Scope and Motivation
Specific software engineering issues and problems which are addressed by the DIPS project are presented in this section, together with existing related work in the respective areas.
1.2.1 Software Component Lifecycle

The key software engineering issue addressed by the DIPS project is to provide support for the software component lifecycle by managing the organization of numerous electronic artifacts created and used in the lifecycle. This is supported by a process-centered software engineering environment (PSEE), for which DIPS provides a framework. This section outlines the scope of the DIPS project and the associated terminology used in this thesis.

Software: The DIPS system is designed to provide organizational support in software development processes, managing primarily software artifacts (documents). This includes executable programs in any form, source code and object code, and all associated documentation including specification and manuals. The main emphasis is on text-based artifacts, both formal and informal, that are of medium- to long-term value to a process (e.g. still of interest after a week), with minute day-to-day modifications to artifacts typically being handled separately. Ultimately, process participants must decide what artifacts to include in a process and when to do so. Desirable extensions to the system, but which are beyond the scope of this thesis, include support for numerous small informal documents (e.g. e-mail messages among process participants) [Kai97] and support for multimedia and other non-text documents. The envisaged granularity (size) of artifacts in a process is typically a document, file, module, component, etc., but not an individual object in the object-oriented sense.

Also, since the DIPS system currently targets mainly development processes for software, we do not study how information or data artifacts which are not related to the development of the software, but which on the other hand make use of the completed software, could be managed in processes. While beyond the scope of this thesis, it would be highly desirable and promising to study how such “information processes” could be managed similarly to development processes. E.g. given a data artifact such as a graphics file, this could allow an appropriate application to be identified with which the data artifact may be viewed or edited, e.g. a suitable graphics editor, by introducing a link (reference) from the data artifact’s information process to the application’s development process. Similarly, in the case of objects in the object-oriented sense, object instances would be managed in information processes containing links to the object implementation in the object’s development process to avoid storing a copy of the code for every instance, even though from the programmer’s point of view an object is a single entity containing both code and data.

Furthermore, hardware is not supported in processes, since every physical entity would require an electronic representative in the process, which
introduces consistency problems, since the association of physical to electronic entities is not automatable, and deployment requires physical manufacturing and distribution, unlike software. However, since all software ultimately depends on hardware, and in view of increasingly popular hardware/software co-design activities, where the distinction hardware/software is not precisely known when a project commences [HP94], it would be desirable to investigate extensions for hardware development support in the DIPS system. Also, the pre-manufacturing activities in hardware development are increasingly similar to software development, e.g. using tools to create hardware specifications that can be regarded as software.

Concerning deployment, while the DIPS system in principle targets development of all kinds of software, we do not study management of access privileges and user lists that would be required for all but public free software. Also, issues such as electronic payment, copyright and licensing are beyond the scope of this thesis. Leaving these issues open however allows other interesting approaches to be considered such as the pay-per-use scheme for software proposed by Cox [Cox96][Cox97], instead of the conventional pay-per-copy scheme. However, marketing information could be stored in processes as additional information, and different projections of a process could be used to display only information currently relevant (e.g. development or marketing information).

**Components:** The DIPS system supports development of all types of software, but since the main benefit is in managing dependencies among software, we use the term “software components” to emphasize the fact that primarily software is targeted that again depends on other software (or perhaps on hardware). Thus, “software component” is used in a general sense meaning a part of a larger software system, i.e. software with dependencies, and not strictly meaning a set of related objects in the object-oriented sense, that are discussed e.g. at the yearly ECOOP WCOP workshops [WCOP98]. Nearly all software can thus be regarded as a component, e.g. even a conventional application depends on an operating system. But conventional applications are more and more being replaced by “systems of systems” of software components, thus introducing increasingly numerous and complex challenges of managing dependencies among these components, which will make systems even more essential that allow different processes to be linked together to represent such dependencies, such as the DIPS system.

**Lifecycle:** The lifecycle of a software component may be split into two distinct phases, the *development phase* and the *deployment phase*, each of which again consist of many activities. In general, the two phases may be interleaved, e.g. maintenance activities (belonging to the development
phase) may follow deployment activities. However, if a specific version of a software component is regarded, the phases are not interleaved, since maintenance development modifies the component and therefore by definition creates a new version that differs from those deployed earlier. The development phase incorporates the software development process (also known as software process or process) and includes all software engineering activities related to development and maintenance of software, while the deployment phase includes configuration, release, distribution (delivery), installation, runtime instantiation, and finally removal (retirement) of software. (Configuration and release could however also be viewed as development activities, since they are typically carried out by developers.) Fig. 1.1 illustrates the activities and phases in the lifecycle of a typical software component. Many versions (revisions and/or variants) of a component may be developed, and a version may be installed at numerous user sites, and an installation may be instantiated for execution many times.

Fig. 1.1: Typical lifecycle of software components from idea via development and deployment to execution respectively removal.
There is some confusion in terminology, as some terms have been used differently by different authors or at different times. The term software lifecycle was originally coined to depict a software's complete life from identification of the need for the software through creation, deployment, maintenance and until retirement of the software [ISO95]. More recently, the term software process was introduced. Some use it as a more modern synonym for software lifecycle [Blu94], while others (including us) view the software process as only a part of the software lifecycle, specifically the process of creation and modification, up to but not including deployment [HHHW97b].

Pragmatically, since most software process research projects nearly exclusively handle only the creation and modification aspects of software, but very rarely deal with deployment, it stands to reason that the term software process should be limited to development activities. Conceptually, deployment should be distinct from the process, since development involves a limited number of participants working on a limited number of replicas of a process and having read and write access thereupon, whereas deployment (at least from distribution onwards) typically involves a large number of participants mostly restricted to read-only access but working with a large number of identical process replicas or parts thereof (respectively the software developed therein).

Although directly influencing a component's lifecycle, we do not study the encompassing business process that manages the decisions about developing and deploying software. However, some of this information could be kept in requirements documents within a software process. The distinction between business and software processes is not always clear, and it would be desirable to share some information among the processes. Information about financing, planning and timing, and allocation of manpower and resources, may well be included in a software process, and in particular information about technical issues such as tools and environments used should be included (including their versioning information).

1.2.1.1 Software Development

Development activities, particularly for a larger software project, are typically organized in a software development process and include both the initial development and the subsequent revision or maintenance (correction, extension, porting, etc.) of a system. Besides revisions (versions developed in sequence), development may also involve several variants (versions developed or used in parallel) of a system [CW98][CW96][Moh94][Tic88] or "program families" [Par76], e.g. for different target platforms. Individual development activities include specification, design, implementation, testing, verification, validation, etc., and use a multitude of tools for the different activities or
development steps. Allocation of tools, time, manpower, finances, and other resources also needs to be managed. Development may involve from one person to large numbers of collaborating developers and even teams in a widely distributed setting. In view of these activities, in a large project there are numerous artifacts of different types as well as dependencies among artifacts to keep track of, which calls for automated support for their management, as well as for the management of their tools and environments. While conventional automated support ranges from simple tools such as editors, compilers and other individual CASE tools to workbenches and complete environments [Fug93], the complexity of organizational issues in a large project requires an integrated approach to manage artifacts and dependencies in an environment rather than with individual and separate tools, thus the DIPS project investigates an integrated approach for such an environment.

1.2.1.2 Software Deployment

Deployment of a software component includes all activities immediately following development up to commencement of execution of the component. While deployment does not form part of the process since it introduces no new artifacts or dependencies in the process, it is closely related to the process because it accesses artifacts, dependencies and other information in it. Also, development will typically continue on the process after or simultaneously to deployment, on other (newer) process parts. The first two deployment activities are configuration, which selects sets of related artifacts by specific criteria, and release, which defines access privileges on configurations to allow access to selected configurations to different customers and possibly to the public. Configuration and release information are added to the process as additional information, thus these deployment activities have write access to the process and are typically performed by the developers themselves or other closely associated persons, e.g. in the same company.

While the further deployment activities may involve large numbers of customers that have been granted access to specified (released) parts of the process, their access is limited to read-only operations. The activities include distribution (on-line delivery, replication), which copies parts (configurations) of processes (holding the software) to customer sites via network, and post-deployment or on-site configuration management, which involves local validation of copied configurations to check compatibility with existing configurations as well as identify missing configurations at customer sites. Since deployment, from distribution onwards, typically creates and involves numerous copies of process parts, it cannot any more form part of a centrally coordinated process such as development [HW96][HHHW97a][HHHW97b]. Finally, deployment may include runtime issues such as dynamic loading and linking of components to commence execution. This assumes a modern operating system
supporting dynamic loading and linking, which is increasingly the case with component-based systems. Loading, i.e. copying from local or close-by (in terms of network latency) persistent memory to volatile memory, may be included as a deployment activity if components are stored at customer sites not simply as files, but in a process aware format in the customer's process base (database) consisting of configurations potentially copied from many different sources, where the loader must access the process base instead of the filing system, as in the DIPS system.

1.2.2 Software Component Interoperability Levels

During a software component's lifecycle, various types of information about the component are used at different times and by different people. In the DIPS system, integration of both the development and the deployment phases in one system allows management of such information to be improved. Specifically, we regard information about how a software component interoperates with other components. We have thus classified interoperability information of software components in three levels, as we have first presented in [SMW96], by introducing a new intermediary level between two established ones:

1. **Interface Level:**
   The interface level provides information about the syntax of a component's interface (e.g. its API, application programming interface), as expressed in an interface description language such as CORBA's IDL [OMG98], or in any appropriate programming language. It does not contain information about semantics, versions or origin of a component. It addresses compile-time and loading-time issues of whether components fit together syntactically. Identification of component dependencies is defined by properties of the programming language, compiler, linker, and filing system used; generally, identification of formal documents (source code, object code, symbol files) is by name only, whose scope is (arbitrarily) determined by search paths in the local filing system; the Oberon systems additionally use numeric keys for module validation [WG92]. Dependencies of informal documents are usually managed manually. Versioning concepts may be provided by separate tools, if at all.

2. **Originator Level:**
   The proposed originator level enhances the interface information by information from the software development process where the component originated, specifically version information and dependencies of components. It specifies exactly the context in which the component was developed, including which versions of other components it depends on, and which documentation and other information is associated with it. The originator level relies on the developer to provide this information in the development process. It
addresses compile-time, loading-time, linking-time and runtime compatibility issues of components, as defined in the process by developers. Dependencies may also be established on components after their development [Szy96]. Identification of component dependencies is independent of the programming language or system used, since all documents are managed in the process, the process manages versions and dependencies, and every document has a unique identifier in a well-defined (ideally global) scope, similar to Internet URLs. The process thus provides an additional restriction on any existing syntactical compatibility information.

3. **Semantic Level:**

The semantic level provides a description of a component's semantics, its functionality. No general established way of describing complete semantic information about components exists. Its implementation, the source code, is often the only existing formal description of a component's semantics. The ideal case of a complete formal specification of the semantics that is sufficiently abstract and independent from the implementation rarely exists. Solutions are available only for specific classes of problems, e.g. which may be described by finite automata, abstract state machines [JK98], or algebraic specifications [Gau94], and signatures and constraints of component interfaces are still a matter of research [Han98]. Ideally, component semantics should be specified in the least powerful formalism that is sufficiently powerful to describe the semantics, in order to allow reasoning about specifications to proceed as simply as possible. The semantic level addresses semantic compatibility issues of components as defined by formal specifications, which may manifest themselves at compile-time, loading-time, linking-time, or only at runtime. Although the semantic level does not explicitly contain origin or version information, it implicitly does, as only components that fit together on the originator level will also do so on the semantic level.

From the interface level to the semantic level, every subsequent level reduces the set of potentially interoperating components, since they are described more precisely. The interface level is not sufficient for proper employment of a component, as it provides very little compatibility and no semantic information [BM91][Man95][Hei95][Tho98], as its concepts are inadequate for covering such issues. On this level, interoperability information is left to the documentation (e.g., in an accompanying "Readme" text or in an on-line bug database [Bug98]), or worse, to the interpretation of the person working with the component. The semantic level, on the other hand, provides full information about the functionality (meanings and behavior) of a component. In order to access this information, either its implementation (source code) or a formal specification would need to be studied, which are rarely published or
hardly ever exist. Besides, a component's user often is not interested in implementation details, but only in compatibility issues. No general established and understandable abstract way to describe complete semantics exists yet, thus semantic level information is neither feasible for developers, nor is it desirable for users, if compatibility issues can be described on the originator level.

Our more pragmatic approach focuses on the development process and makes use of the fact that interoperability issues are often investigated during development already, but unlike interfaces and semantics, are usually not retained. It therefore assumes a certain confidence in developers to correctly provide this information. We postulate that the approach to regard the development context represents an improvement over current practice in managing interoperability which often only deals with the interface level, while bearing in mind that the next improvement, dealing with the semantic level, is at present usually not well feasible. Furthermore, the reasons information is required from the semantic level frequently have to do with understanding the correct usage of a component, and these questions can often be addressed by information in the originator level without specific semantic knowledge. The proposed intermediary level thus offers steps towards a solution to find correctly interoperating sets of components that is feasible with today's technology.

Within our projects (as will be detailed later), interoperability information issues are addressed on both levels above the common interface level: By managing version and dependency information in the development process, retaining this information in the process beyond development time and making it available for deployment, the DIPS system addresses originator level interoperability information issues. Furthermore, DIPS supports use of formal methods in the process, and our associated CHIPS (Components for Highly Integrated Process Support) project allows semantic level interoperability information to be described in DIPS processes for a specific case: CHIPS demonstrates a possibility for formal specifications of tools to be described with extensible attribute grammars (EAGs) and the tools thereupon to be generated using a compiler-compiler, which in turn allows specific semantic properties of artifacts edited by such tools to be formally described. As will be seen below, CHIPS and DIPS together form the GIPSY project [MWSS96][GIPSY98].

1.2.3 Background and Related Issues

The following sections present both the background of areas DIPS is directly involved in as well as related issues where there are further opportunities for DIPS to provide contributions.
1.2.3.1 Software Engineering

The term “software engineering” was coined in 1968 to encompass measures designed to address problems identified in the so-called “software crisis” at the time, when it was realized that design principles that were adequate for small systems could not simply be scaled up to cope with increasingly large software systems [Boe81][GJM91][DK76], in particular as software engineering may be defined as “multi-person construction of multi-version programs” [Par78].

Compared to the phenomenously increasing performance and decreasing cost of hardware (CPU performance, RAM size: Moore’s Law about the density growth of ICs) without precedence, and the even greater increases in network performance [Shi98a], and considering ever increasing requirements and features [Wir95], software’s cost/performance ratio cannot keep up. Also, due to the omnipresence of software, many failed projects and software errors are perceived in low-quality software. However, the term “software crisis” is a very relative one, depending on one’s point of view. Economically, the software industry is often very successful, the amount of errors is not beyond expectation [Hoa96][DeM95b] and its cost is justifiable [DeM93][DeM95a][Boe81][Bot97].

Inherent properties of software, particularly evident in large systems, make software fundamentally different from any other man-made entities, including hardware, therefore known engineering principles cannot be readily applied. These properties include complexity (difficulty to abstract, scale, or find regularity) [Lev97], conformity (compatibility requirements with existing systems), changeability (perceived ease of modification without taking into account the consequences), invisibility (again, difficulty to abstract and reason about), and others [Bro95][Par85][GJM91][Mey97][Cox96][Der81][Ham97][Par94].

The relatively limited abstraction possibilities, compared to other technical or to natural sciences [Weg95][Weg97], make it difficult to assemble larger software systems out of existing smaller software components. Software’s properties thus make software design and implementation to be largely creative, innovative tasks, whereas the duplication or manufacturing is trivial, unlike in other industries where (repetitive, automatable) production is more involving than design. Software performance is not growing at the same pace as hardware performance, since hardware advances are due to a large degree to technological advances, therefore, compared to software, hardware performance increases are relatively less dependent on advances in the increasingly complex design, whereas software requires more design, i.e. creative, advances. This leads to large perceived costs of software.
Brooks distinguishes “essential” and “accidental” difficulties in software engineering [Bro86], where the accidental difficulties, such as translation of design to implementation, continue to be reduced by technological advantages, such as high-level programming languages, software engineering environment support, or increased software reuse, whereas he does not see any silver bullet in the foreseeable future that could substantially reduce the essential or hard difficulties, such as design of software. Thus, even if it would be more beneficial to solve the essential difficulties, these creative tasks will remain to be done by humans, and computerized support in software engineering is best suited to handle the accidental difficulties. Thus, computerized support should address mainly organizational issues in software engineering to complement the creative work to be done by humans [Blu92][DL87][Rus89][WR93].

1.2.3.2 Software Component Reuse

Reuse of software components is a possibility to decrease software costs, provided the overhead of making components reusable is mitigated by reusing them a sufficient number of times instead of developing new ones [Tra95]. While reuse has been identified as an important software engineering goal a long time ago [Bro95][LC85], success has been limited, although recently emerged component and distributed object technologies have sparked a renewed interest in reusability [Ude94][Adl95][OHE96] [Bro96][BM97][Szy96][Szy98][Voa98]. As indicated above, inherent properties in software still make it difficult to build generic reusable components or frameworks, although these are areas of active research, and generally there is also a tradeoff between integration and reuse [BEM92]. Furthermore, widely reusable code is often implemented in hardware sooner or later, which obviously does not contribute to increase software reuse either.

Most authors emphasize a single issue to increase software component reuse. Meyer believes that improved formal specifications of semantics of software components are essential for their reuse [Mey96][Mey97], while Tracz emphasizes organizational issues [Tra95]. Cox identifies a lack of motivation in software engineers to build and employ reusable software components due to software’s non-atomic nature, i.e. the ease of duplication in contrast to atomic or non-electronic goods which makes it difficult to enforce property rights and thus provides no incentive to build reusable components [Cox96][Cox97]. He thus proposes to make software freely copyable but use a hardware-supported pay-per-use scheme for software instead of the conventional pay-per-own scheme.

While reuse of components, i.e. code reuse, would contribute to solving the accidental difficulties of software engineering, as identified by Brooks [Bro86], design reuse would be even more beneficial, as it would be directed at the essential difficulties. This can be exploited by recognizing
design patterns [GHJV94][CS94][Gar95]. Furthermore, component reuse requires precise, i.e. formal specifications of their semantics, which is still a difficult problem, also considering that source code is rarely released, although there are some attempts at using contracts and other mechanisms [JM97]. Patterns however are more abstract and therefore require less precise semantical specifications than concrete objects, thus their reuse is more promising than the reuse of components, considering the current state-of-the-art.

We believe both design and code reuse should be supported. To increase code reuse, a single improvement alone does not suffice, but several issues should be addressed. The emphasis should be on organizational support as provided by the origin level of software component interoperability information, but the use of formal methods in the software process should also be possible where appropriate. For deployed software, a pay-per-use scheme would also be very beneficial since software would be freely copyable and this would reduce requirements to manage access restrictions, although this is beyond the scope of this thesis. Design reuse is more difficult to enhance with computerized support, but organizational support in the software process may help locate and identify existing design patterns.

1.2.3.3 Software Quality

Improvement of software quality is an important issue in software engineering. Various process improvement or quality guidelines as well as process maturity frameworks have been proposed and utilized with some success, e.g. Capability Maturity Model (CMM) [Hum88][PCCW93][Her97], Personal Software Process (PSP) [Hum95], Bootstrap [Car93][Mes97], Spice [Mes97], IEEE/EIA 12207 [FS98] (respectively ISO/IEC 12207 [ISO95]), and ISO 9000 [Pau95]; [She97] compares seven important compliance frameworks (CMM, SE-CMM, IPD-CMM, ISO 9000, SDCE, MIL-STD-498, and Trillium) and mentions dozens more such frameworks, references, and standards. However, the inherent difficulty of measuring and quantifying software quality remains; i.e., only the problems found can be documented, not those that have escaped detection [Jon95][ESA96][LT93]. A possible approach is to increase the process' maturity level, as measured by specified features [PCCW93][Lai93]. An environment can provide such required features in the area of process support, e.g. organizational support and configuration management. The software's quality may then be derived from the quality of its development process [Sch97b]. Nevertheless, it should always be kept in mind that while a high quality process, process assessment, process improvement, as well as extensive testing, validation, verification, etc., may contribute to high quality software, they do not automatically imply it [Voa97][LL95b][Dij65].
1.2.3.4 Distributed Component and Internet Technology

Recently emerged component and distributed object technologies provide new possibilities both for software development and for the implementation of distributed software engineering environments [Adl95] respectively repositories for such environments [BD94][Bro93][BS95][DF95][TG95][NJB97]. While the reuse benefits from component software are still relatively modest, the main advantages lie in increased system and language independence, therefore conventional object or other databases are now used less frequently to construct repositories. The Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA) e.g. provides distributed objects where object interfaces are described in the common Interface Definition Language (IDL) but implemented in different languages on a variety of platforms, thus allowing integration and interoperation of heterogeneous distributed applications [OMG98][OHE96]. While the basic distributed objects are relatively efficient, additional features are available separately as required, e.g. Common Object Service Specification (COSS) services such as support for transactions or replication, object broker interoperability across the Internet via the Internet Inter-ORB Protocol (IIOP), or interoperability with legacy software via object adapters [OMG98][OHE96]. CORBA heterogeneous distributed objects could be used to implement a process framework that runs on many different platforms and uses very heterogeneous tools and objects within one process, as required by the process, and object adapters could be used for existing tools. Microsoft’s DCOM similarly supports multi-lingual distributed implementations but is mainly limited to Windows platforms [Mic98b][OHE96]. Java portably provides distributed programming on several platforms, but only in that language [Sun98].

For many distributed applications, simpler protocols may suffice, e.g. popular Internet protocols such as HTTP, if the emphasis is more on distributed information access and platform-independent display rather than distributed computation [OHE96][FP98]. The enormous growth of the Internet in recent years has demonstrated that simple protocols can be very successful and popular. It demonstrates that simple interoperable protocols are essential and have a greater chance of being widely adopted than complex ones. It is probably the simplicity, openness and necessary incompleteness of WWW protocols that have contributed significantly to their current popularity [Shi98b], therefore such issues should also be considered when designing a process framework that is to be widely employed and used on many different platforms. The wide availability of TCP/IP, of HTTP servers and browsers, of browsers supporting HTML, GIF, plugins, JavaScript, Java, etc., provides opportunities for process frameworks to be integrated in the WWW, since adoption and
compatibility issues on different platforms should be easier if the framework is based on widely available technology.

1.2.3.5 Virtual Organizations and Global Distribution

The recent remarkable growth of the Internet and other improved telecommuting possibilities reduce the requirement for physical presence and increasingly allow geographically distributed collaborative operations within an organization; likewise, these technologies also help promote a looser coupling of organizations themselves and a relaxation of organization boundaries, the most extreme form of which are virtual organizations [Mow97][DM92][DHW98][MK98][FK97][Mur97a]. This poses new or increased organizational challenges, such as widely distributed process support, linking of several distributed processes, distributed deployment, and distributed configuration management, which are becoming increasingly complex [NS97][Arn94][BKMS95] and thus require an organizationally integrated approach to replace existing heterogeneous tools. Although this thesis focuses mainly on the technical issues of organizational integration, it is clear that the non-technical ability and willingness of different parties (e.g. teams or companies) to participate in a virtual organization, to share information, and thus to link their processes together, will also need to be studied, encouraged and supported, particularly in view of the fact that non-technical issues such as project management are difficult already for conventional projects [Wei98].

1.2.3.6 Software Development Processes and Process Models

In a typical software engineering approach to software development, a software product is created or modified by following a sequence of (manual or automatic) steps or activities that make up the product’s software development process. After initially defining the process, it is enacted (executed) to create or modify (evolve) the software product. Typically, neither all the steps nor their order can be fully determined in advance, thus the process itself creating the product also evolves, not only the product.

A software development process or software process or simply (in this thesis) process thus is a sequence of steps planned and usually carried out to create or modify a software product or simply product. A process model or process definition is a simplified formalized rendition of a process, noted in a specific formal process modeling language (notation), simplified by the restrictions of the process modeling language which typically does not allow modeling of all real-world aspects, and simplified to reflect only aspects of concern. The formal notation is a requirement for and is the basis for electronic process support. An abstract or generic process model omits details required to create/modify any specific software product. A new process or a part thereof is typically defined by
combining process patterns (design patterns) of earlier successful processes with principles of generic process models and enhancing these with specific features for the new product under development.

The terminology concerning software processes is still partially contradictory in the literature, although there have been several proposals for defining key concepts and terms in a consistent terminological framework [FH93][CFFS92][Leh87][Lon93][Ost87][STO95][Huf96][GJ96a][GJ96b][FKN94][Mad91][ACF97]. Some authors use the term process model for the real-world process and the term process for its electronically executable simplified formalized counterpart; some authors distinguish process model (abstraction of real-world process in any informal notation, typically noted on paper) and process definition (formalized electronic notation of the process model), etc. We will use the set of process concepts and terms defined in the previous paragraph throughout this thesis, except that where there is no danger of confusion we may use only the term process to signify process model, the formal notation of the real-world process.

Software processes have already been defined several decades ago [Ben56][Roy70][Boe76][Bro95][Hum88][Agr86a][Agr86b][DBC88][Mil80][Som96b], but without electronic support at the time. Computerized process support (for process definition, enactment, evolution, measurement, etc.) is relatively new and has emerged to a significant degree only in the 1990’s [NF92][Rei96][GJ96b][GJ96a][FKN94][WR93][Tul91][Gru91b][Leh97]. Generic process models that have evolved include the waterfall [Ben56][Roy70][Boe76][MJ82][Gla82], iterative, evolutionary, transformation, prototyping, spiral [Boe86], cleanroom [Lin94][SBB87][Bei97] and other models, as well as combinations [GJM91], each with specific advantages and drawbacks. Many different process modeling languages exist, following topology-based or constraint-based paradigms, or product-oriented or process-oriented paradigms, and hybrids (combinations of these), as will be presented in a separate section in the next chapter.

In spite of some contradictory terminology in the literature, we have identified at least three relevant process abstraction levels and three relevant metaprocess levels (process metalevels) (fig. 1.2) with regard to the concepts defined above. The process abstraction levels specify abstractions of the process itself, i.e. the activities to be performed on it and how it evolves, whereas the metaprocess levels specify abstractions of process support, i.e. specification and evolution of process support (tools, framework, process modeling language). In principle, an infinite number of metaprocess levels exists, since the metaprocesses that describe process support are also processes that themselves need to be supported, although we will restrict ourselves to the three lowest levels. On the two lowest
levels, product and process, abstraction and metaprocess issues are not distinguished, whereas higher levels are separate.

The software product being developed or modified is on the lowest level, the product level. On the next abstraction and metaprocess level, the process level, product engineering and product evolution are carried out by enacting and iterating the process definition, whereby the process provides both an abstraction of the product and a metalevel of support for the product. On the next abstraction level are both the generic or abstract process model used to create the specific process definition, as well as the real-world process (not shown), both of which guide process engineering, i.e. process definition and evolution. The development processes of all the entities belonging to process support (used for product and process engineering), i.e. belonging to the environment, are on the corresponding metaprocess level, thus this includes the processes of tools, the framework, and implied by the framework’s process also the process of the framework’s built-in process modeling language. On higher metaprocess levels (not shown), the environment itself is evolved (e.g. the process modeling language), i.e. they are the metalevels of tool, framework, and language evolution. At least the three lowest metaprocess levels, and thereby also the two lowest process abstraction levels, should be well supported by a process support system.

For example, the product could be a software component, consisting of artifacts such as design documentation, source code, object code, and manual. The process would define and manage the steps carried out to create and modify those artifacts, e.g. commence with design documentation, continue with source code implementation, etc. The abstract process model could be the waterfall process model, i.e. a generic
process model that is subsequently refined in the process definition to specify the actual artifacts. The metaprocesses would be the processes implementing the tools and environment used in the software component's process, such as editor, compiler, etc. Thus, the metaprocesses are normal processes too, but which have been completed and whose products (the tools) may now be used. A process whose (completed) product is used to support another (newer) process may thus assume the role of a metaprocess with respect to the new process, here the software component's process.

1.2.3.7 Process-Centered Software Engineering Environments

Systems for computerized software engineering support can be characterized as individual tools (CASE (computer-aided software engineering) tools) used to carry out a specific task (e.g. editors, compilers), as workbenches (a collection of related tools), and as software engineering environments (SEEs) or simply environments (an integrated system of tools) [Fug93][Som96a][Chi93]. Integrated process support environments (IPSEs) or process-centered software engineering environments (PSEEs) are a special type of environment where the common understanding of the software process by all tools involved is emphasized [GJ96a][GJ96b][FKN94][FG94][GCCM95][HKW97][BF93], whereas other types of environments may emphasize other aspects of integration [BEM92][BB95][RBM93][HM97].

In a PSEE, the scope of support for a process may range from the single workstation/individual developer up to large teams in a widely distributed setting, and may include integration of only a few tools or of a wide selection of heterogeneous tools. Besides supporting individual distributed processes, linking of several processes is also envisageable, as is support beyond development time, for process-integrated post-development activities such as configuration management and software deployment [MK98][ACF97].

A PSEE typically consists of process-variant and process-invariant parts. Tools that are used in individual process steps, such as editors and compilers, are process-variant parts, i.e. tools may vary from process to process, as required by a specific process, with some tools even generated for a specific purpose [KS97][CTOD95]. The process-invariant part of a PSEE is a kernel or framework which may be used by several different PSEEs and which provides process definition and enactment services and tool integration services used by all processes and which ensures a consistent understanding of a process by all tools involved. Ideally, all services concerning multiple process steps conventionally provided by additional tools should be integrated in the framework (e.g. configuration management), eliminating the need for any tools beyond the process-variant ones used in the individual process steps.
Chapter 1

Fig. 1.3: Development path of a DIPS-based PSEE
The tools are plugged into the process-invariant kernel or framework in order to make up a complete but process-specific PSEE. The term "framework" is now commonly used for the process-invariant environment kernel [BEM92][ECMA93a][CN92][KS97] and is meant in the general sense of extensible software and not in the strict sense of an object-oriented class framework [Mös95][Mös96].

The DIPS system is a process-invariant framework which may be extended to a full environment (PSEE) by defining a specific process and by plugging in process-specific tools used in the individual process steps (fig. 1.3). Tools are not covered in the current thesis, but the associated CHIPS project investigates tool generation for a specific kind of tools that may be used in the DIPS framework [MWSS96][GIPSY98].

1.2.3.8 Software Deployment Systems

Software deployment is typically carried out manually or by different largely independent tools, but rarely by an integrated system. Release typically involves manual or semi-automatic extraction of object code and documentation from a project database and bundling of these artifacts, together with an installer program, on diskettes, CD-ROMs or on on-line servers. Users with personal workstations (not network computers) typically transfer these artifacts to their hard disk, either manually or automatically by executing the installer program. Such programs may check whether an earlier version of the program to be installed should be overwritten and whether it is compatible with the operating system.

It is envisageable that a deployment system handles release directly out of development processes, performs deployment directly via network, and carries out more wide-ranging compatibility checks with previously installed software components.

1.2.3.9 Configuration Management

Configuration management involves selection and grouping of related versioned components (and other artifacts) by specific criteria, e.g. revisions and variants, to form configurations, sets of compatible components to be used together [Dar92][LL95a][CW98][CW97][Cag95] [Kil97][AJ89]. These are release activities typically performed by developers, but increasingly users are also becoming involved in configuration management, as "systems of systems" of component software are becoming more and more widespread, i.e. developers do not deliver complete applications any more but instead users assemble applications out of different components. Thus, increasing numbers of smaller components to configure, together with distribution issues make configuration management an increasingly complex task [Jon96][Mac95] [Wal92][HHW95][LR96]. While configuration management is still typically a separate task, its integration in the development process is
1.2.4 Conventional Requirements

In order to build a framework for a PSEE, not only objectives derived from currently challenging issues should be regarded, but conventional requirements on PSEEs should also be reviewed. These can be summarized as follows [GJ96a][GJ96b][FKN94][Huf96]:

PSEEs should provide support for the following three areas of activities:

- process engineering: definition and maintenance of software process models;
- software engineering: development and maintenance of a software product, by following a software process;
- project management: coordination and monitoring of the activities of software engineering, by monitoring the process.

In order to support these goals, a PSEE should provide an implementation of a process modeling language, which allows the following specific activities to be carried out:

- process definition and presentation: creation of the formal process model, distribution to all process members, presentation in various user interfaces;
- process instantiation: if necessary, translation of the process definition (model) into an enactable format and initialization of variables: this is an intermediate step between process definition and enactment required by some PSEEs, but not by the DIPS system;
- process enactment: execution of the process model, to control, guide and execute the software engineering activities;
- process evolution: modification of the process model, after initial enactment;
- process analysis and monitoring: project management activities.

Besides providing support for the above-mentioned activities, further PSEE features include:

- multi-user support: allows several developers to work on a process, on different distributed workstations;
- tool-integration services: allow tools to be plugged into the PSEE to be used in individual process steps.

While there are obviously PSEEs that provide additional features, these are the conventional requirements any PSEE should fulfill. They will be
reviewed and adapted for the DIPS system, and enhanced by additional requirements to provide a broader scope for the DIPS system than in conventional PSEEs, to include issues such as process linking, configuration management, and deployment support.

13 Objectives

The Internet's remarkable growth and the emergence of distributed component technologies influence both software development and software deployment and demand their traditional requirements to be revised. As seen in the section on conventional requirements, process-centered software engineering environments must support definition and enactment of software development processes, allow different tools to be used for different process steps, and enable multiple participants to work on a process [GJ96b]. Distributed software engineering [Lof95] (in the sense of collaborative engineering, not engineering for distributed systems [Kra94]) allows multiple, even widely distributed, developers to participate in individual processes, but typically provides little support for connecting different processes or for working in heterogeneous environments.

Due to the Internet's growth and the new component technologies, software is increasingly being developed as a "system of systems" by a heterogeneous group of organizations, in virtual organizations [DM92] or independently (e.g. if a developer uses an application framework), requiring large and complex dependency graphs of software components to be managed. Likewise, software systems are becoming less self-contained, with numerous components from different sources required to make up an application, and more users have easier access to downloadable software. Since configuration management is complex already for development and maintenance [Jon96][Leb94][Rus89], calling for integration in the process [Joe97][NS97], deployment tasks such as installation of valid component configurations will provide a serious challenge to users. With larger numbers of components making up "systems of systems," dependencies become more complicated and cannot be managed by simple installer programs any more [Goo97].

Thus, support for software development processes should emphasize more the communication among heterogeneous widely distributed developers than within individual processes, by providing a common understanding of processes and allowing different processes to be linked together to reflect component dependencies. Comparatively less emphasis should be placed on large sophisticated environments respectively process modeling languages that do not support integration of different processes and of heterogeneous systems well. Similarly, due to increasingly complex dependencies in "systems of systems," better support for software deployment is desirable, particularly since many dependencies can be
formalized, allowing some deployment tasks to be automated. Therefore, product structure information such as dependencies and versions should be incorporated already in the software development process by developers and made available and used in subsequent deployment of the software product, since developers have more information about their components than anyone else. This provides a common understanding of the process and of components both to developers and users, incorporating component origin (globally unique identifier of an implementation), dependencies (what other components, operating system, documentation, etc. a component requires or is related to), versions (name and unique revision and variant), and further metadata.

As identified in the sections on background issues, integrated organizational support can provide a significant contribution to improve software engineering practices, component reuse and software quality. This includes integration of configuration management and software deployment features in a system incorporating a process-centered software engineering environment. Widely distributed development, virtual organizations and interoperating components require many heterogeneous tools and platforms to be employed within a process, and require different processes to be loosely coupled.

The conventional requirements for a PSEE of using a sophisticated process modeling language should thus be revised to provide a simpler language, but with a broader scope [DHW98][NS97]. This can be compared to simple, yet highly successful and widely distributed Internet protocols [Shi98b]. Distributed software engineering should extend the locality scope of processes beyond individual processes, by additionally supporting linking of different processes, including evolving processes and completed ones, within an enterprise, or within a virtual organization, or processes of otherwise independent developers. As a vision, this will form a large “global software process.” Process linking reflects dependencies of components under development among each other as well as on existing components, which may be supplied by different developers. Such dependencies precisely and uniquely identify versions and origins of components, including object code, source code, documentation, etc., which is important information for development, e.g. specification, documentation, implementation, compilation, linking, testing, configuration and release, as well as for deployment, e.g. retrieval and installation by users.

Furthermore, not only the locality scope but also the duration scope of processes should be extended, beyond development time, by using process information in deploying software. Deployment has been identified to encompass configuration, release, installation, and removal of software, both for initial deployment and updates, and involves developers to make
software available and users to retrieve software and manage individual user-site software registries [HHHW97a], while software releasing requires developers to provide dependencies, source location, and further metadata for every software component [HHHW97b]. Since developers best know about their components, we believe much of this information, particularly dependencies, should be inserted in the process already at development time, with additional information, e.g. for marketing (pricing, source location, etc.), added in the process at release time. Our hypothesis is that deployment and runtime dependencies are a subset of dependencies known already at development time. This does not contradict independent component development, since components only reference those they depend on, but not vice versa. The process thus becomes the primary integrated source of information about its software, allowing the process to be used beyond its traditional purposes, for the full life cycle of a component including development and deployment, instead of dispersing such information in several unrelated databases. This supports the originator level of interoperability information.

Since development of linked processes occurs in widely distributed heterogeneous organizations, and users similarly represent widely distributed heterogeneous organizations, it is essential that heterogeneous platforms and systems are supported, which demands development and deployment systems to be simple as well as interoperable both among developers and among developers and users. Ideally, there should also be some integration in the WWW to facilitate both distributed deployment and operation of the environment itself. Support should also be included for tools for formal specifications, in order to allow dependencies to be described formally where appropriate, thus supporting the semantic level of interoperability information where possible.

To satisfy these objectives, we propose a modeling language for software development processes that uses a common model for both process structure and product structure and provides a common understanding of product and process to all involved parties (developers, users, tools), including origin, version and dependency information, thereby allowing dependency and version management to be incorporated in the process. We propose a distributed framework implementing the language which supports both software development and software deployment and thus provides a common interoperable understanding of the software process. This integration allows all lifecycle tools to access this information consistently, including development tools, deployment and runtime systems, and for customers allows managing this information together with other site information.
To summarize, the main distinctions from other process modeling language and process-centered software engineering environment projects include the following criteria:

- support for the full component lifecycle: both development and deployment phases;
- support for the process history;
- process access for developers of the process, for external developers, and for users;
- global scope of processes;
- linking of processes;
- support for all three interoperability information levels;
- different types and levels of integration services for heterogeneous tools and platforms.

Both the objectives and the non-objectives of the DIPS project are summarized in the following sections, to show not only the goals but also the limitations of the project. A further distinction is in design issues and realization issues.

### 1.3.1 Design Objectives

The design objectives of the DIPS project can be summarized as follows:

- to provide electronic support for software development processes;
- to handle software components and related mainly text-based artifacts in processes;
- to integrate development and deployment activities;
- to allow multiple processes to be connected;
- to support evolution on multiple (meta-) levels;
- to support various types of integration (artifact, process, organization, as will be presented in a separate section);
- to support heterogeneous artifacts;
- to allow formal methods to be used in processes;
- to allow tools to be used that describe semantics of dependencies in processes;
- to design a simple but powerful process modeling language meeting the above design objectives;
• to design a distributed framework for an environment implementing the process modeling language.

1.3.2 Design Non-objectives
Some desirable design objectives which are related to the DIPS project but which are beyond the scope of the project can be summarized as follows:

• to support small-scale artifacts and other informal minutiae in processes, multimedia and other non-text-based artifacts, and hardware artifacts.

• to provide sophisticated features in the process modeling language such as comprehensive support for process analysis;

• to support description of dependencies with multiple semantics in the process modeling language;

• to support evolution of the process modeling language and other higher (meta-) levels;

• to support large-scale organizational issues beyond software processes such as business processes;

• to support other types of processes such as information processes.

1.3.3 Realization Objectives
The realization objectives of the DIPS project describe the specific activities to be carried out to meet the design objectives and can be summarized as follows:

• to carry out the design of a process modeling language meeting the design objectives;

• to carry out the design of a distributed framework for an environment implementing the process modeling language;

• for validation purposes, to implement a limited-functionality prototype of the framework demonstrating most of the ideas presented, and to discuss examples and results;

• to demonstrate integration in the WWW.

1.3.4 Realization Non-objectives
Some desirable realization objectives which are related to the DIPS project but which are beyond the scope of the project can be summarized as follows:

• to add features to the framework beyond those required by the process modeling language;
to implement a fully-functional production-quality framework incorporating all ideas presented;

to provide integration services for numerous existing tools.

1.4 Contributions

In meeting the objectives as defined in the previous sections, the DIPS project provides the following contributions, as will be elaborated in the respective following chapters:

• classification of software component interoperability levels (as presented earlier), and research into support of the originator level leading to the design of a suitable process modeling language;

• design of a process modeling language integrating different features in a single structure, in particular providing:
  
  • integration of process definition (process engineering) and process enactment (process performing) through a directly enactable (interpretable) process definition;

  • integration of and support for three metaprocess levels: metaprocess (process evolution), process (product evolution), and product, thus providing a unified formalism for process and product structure;

  • furthermore, implicit integration of higher metaprocess levels (e.g. environment evolution) by describing environment (tool and framework) processes as product processes too, and integration of the environment processes in the product processes the environments support;

  • integration of both process evolution and product evolution in a common structure (process dimension);

  • linking of multiple processes to describe interprocess dependencies in “global software processes;”

  • integration of development (process) and deployment features (release, distribution, configuration management, interoperability information, possibility for component catalogs), respectively developer and user issues;

  • integration of process (artifact dependencies) and repository (artifact version and configuration management) issues;

  • possibility for integration of organizational issues besides product and process issues in a unified structure;

  • integration of different dependency types for both revisions and variants in a common formalism;
• tool support for formalizing dependencies and process constraints;
• design of a distributed framework for a process-centered software engineering environment implementing the process modeling language providing:
  • widely distributed operation;
  • heterogeneous tool integration and other integration services;
  • development support;
  • release and deployment support;
  • configuration and configuration set management;
• implementation of a limited-functionality distributed research prototype of the framework to validate the process modeling language and framework design;
• demonstration of WWW integration through implementation of a dedicated WWW server providing interoperability information;
• compared to other projects, provision of a process modeling language with a broader scope due to support for linking of several processes, and provision of a framework with a broader scope due to support for the full lifecycle of development and deployment, and as such provision of probably the only project so far that consistently integrates development and deployment in a single system.

1.5 PROJECT HISTORY
The DIPS project is part of the encompassing GIPSY project that was carried out at the TIK Laboratory from ca. 1992 to 1998 by a group of two to four persons, leading to numerous publications and, including this one, three theses.

The first thesis presented theory, design and implementation of the incremental GIPSY (Generator for Incremental Programming Systems) compiler-compiler that allowed tools used in specific process steps to be described formally with extensible attribute grammars (EAGs) [Paa95] to enable data integration on the highest level [Mar94]. As initially suggested in that thesis, the project was subsequently extended to study support of the whole software development process, and, after I joined the GIPSY project in 1994 (after having previously worked in a different project, implementing compilers for a multithreaded parallel machine [FS93]), I consequently introduced the new acronyms CHIPS (Components for Highly Integrated Process Support) and DIPS (Distributed Integrated Process Services) to better distinguish the two parts. The original GIPSY compiler-compiler (tool generator) project became known as CHIPS, and the process project part was called DIPS, with the acronym GIPSY being
retained for historical reasons but obtaining a new meaning (Generating Integrated Process Support Systems) to encompass both CHIPS and DIPS. The "I" in GIPSY now stood for integration, as the incremental features were removed from the CHIPS generator. My "cookbook recipe" for software development was coined, indicating that individual CHIPS-generated tools could be plugged into the DIPS framework to provide a process-centered software engineering environment [GIPSY98]:

"Specify a process with DIPS, generate some CHIPS, and dip them in the DIPS."

The DIPS project evolved an initially Petri net based process modeling language to a two-dimensional and currently three-dimensional graphical process modeling language, building several process engine framework prototypes in Oberon System 3 to implement the language. Besides coping with the developing language, the framework itself evolved from single-user support to a client-server based distributed version to the current distributed object based version. Furthermore, a dedicated WWW server was built to demonstrate possible WWW integration of processes. CHIPS tools, ported to Oberon System 3, were employed in the DIPS framework, and as a milestone, DIPS was also used to define the process of CHIPS development itself. Meanwhile, the second thesis presented both issues of the CHIPS and DIPS project, providing mainly a high-level broad motivational and visionary overview of the two projects, discussing organizational integrity in software processes, and it also republished five papers on selected issues in the DIPS project [Mur97b].

Thus, this is the third thesis; besides introducing new concepts such as integration of software deployment issues, it provides an intermediary link between the two previous theses, by presenting concepts mainly on an intermediary conceptual layer, above the low-level implementation issues of the first thesis and below the high-level visions of the second thesis. This thesis concentrates nearly exclusively on the DIPS project, presenting process modeling language and framework design and implementation issues, and thereby focuses mainly on my own contributions to the DIPS project: The process modeling language is described in greater detail than previously published, including details of evolution and configuration management issues; design and implementation of the distributed framework as well as of the demonstration WWW server are presented; and in particular inclusion of deployment support in the framework is discussed for the first time in a thesis of the project, being the newest extension to the DIPS project.

During the course of the project, numerous publications emerged where I was either principal author [Sch98][SMW97][SMW96][Sch97a][Sch94], co-author [MS98][MS97][MSW96b][MSW96a][MWS96][MWSS96][WSMH94], or otherwise involved, some of which were published twice:
Although this thesis aims at presenting a coherent picture of the current state of the DIPS project, it should nevertheless be noted that, as in a typical research project, the project did not follow a straightforward evolutionary path, as goals were revised and added during the course of the project. Although I initially intended to concentrate only on the repository and distribution aspects of the DIPS framework, thus commencing with TCP/IP implementation for Oberon [Sch94], with several other project members intending to address other DIPS issues, during the course of time I assumed increasing responsibilities also for
other core aspects of the project and thus took over more and more design, implementation and publication tasks. A difficulty therehy was the desire not to unduly discard existing implementations of earlier revisions. Nevertheless, of the 39 Oberon modules currently making up the core of the DIPS framework, 28 were ultimately implemented nearly exclusively by myself, and a further 4 in a significant part by myself, with module sizes ranging from 800 characters/40 lines of Oberon source code/1KB of compiled PowerPC object code to 81000 characters/2400 lines of Oberon source code/55KB of compiled PowerPC object code, and module numbers not including the numerous revisions, variants, utility programs, etc. The total size of the current DIPS kernel is ca. 550000 characters/16000 lines of Oberon source code/370KB of compiled PowerPC object code, which is comparable in size to the first Oberon system. If major revisions, variants and utility programs are also counted, total sizes are increased about fivefold. The CHIPS kernel with major revisions is of a similar size, and all CHIPS-generated tools together are again of a comparable size.

My successively increased tasks concerning design and implementation of the process modeling language did not permit my exploration of the distributed concepts to be pursued to the desired intent, especially concerning my latest idea, deployment support, while similarly the more advanced process modeling language aspects could not be addressed in the desired depth any more. Thus, while this thesis may in places provide the impression that the project is relatively unfinished, due to the substantial project size, towards the end it was my intention to focus on those project aspects required to provide a basic workable system, which I believe has been demonstrated successfully, while at the same time collecting an extensive list of opportunities for further work in the area (in order to ensure that valuable ideas will not be lost). Similarly, efforts to publish results of the DIPS project have increasingly been taken over by myself, even if I do not figure on the author list of all publications I have been involved in.
Chapter 2  
Process Modeling Language  

2.1 PROCESS MODELING LANGUAGE OBJECTIVES AND REQUIREMENTS  

2.1.1 Process Modeling Language Paradigms  

Several attempts at identifying classifications for process modeling language paradigms have been made, according to various criteria, and the issue is still being discussed, as are proposals for new languages and types of languages continually being made [Kel93][OH94][Kel91][Leh97][Ost97][CL95]. Curtis et al. [CK092] (also quoted in [ACF97]) identify five classes of paradigms: programming models, functional models, Petri net (and state transition) models, plan-based models, and quantitative models. Lonchamp [Lon94] (also quoted in [ACF97]) identifies six classes: graphical, net-oriented, procedural, object-oriented, rule-based, and multiparadigm. Armenise et al. [ABGM92] identify six styles: logic rules (rule-based, logic language, attribute grammars), automata (finite state automata, Petri nets), imperative programming languages, artificial intelligence, event-triggered based formalisms, abstract data types, and combinations, and distinguish two types of interfaces: textual and graphical. Huff [Huf96] identifies non-executable paradigms, state-based paradigms (state machines, Petri nets, formal grammars), rule-based paradigms (trigger-based paradigms), and imperative paradigms. Liu and Conradi [LC91] (also quoted in [Con94]) identify five paradigms: rules, database triggers, activity networks, process programs, and hybrids.  

Garg and Jazayeri [GJ96a] propose to distinguish activity-oriented and product-oriented formalisms, as well as language-oriented (textual) and diagrammatic (graphical) notations. Activity-oriented representations include Petri net based ones (e.g. SLANG (SPADE environment) [BFG93], FUNSOFT (MELMAC environment) [DG94][Gru91a], ProcessWeaver [Fer93][BF93]) and rule-based ones (e.g. Marvel [KBS90]). Product-oriented representations include ones based on a relational data model (e.g. Adele [BEM94]) and ones based on an object model. Diagrammatic notations are derived e.g. from statecharts (e.g. STATEMATE [Kel89]), Structured Analysis and Design Techniques
(SADT) and derivatives, or Petri nets. Language-oriented notations are derived e.g. from imperative programming languages (as popularized by Osterweil [Ost87][Ost97], e.g. APPL/A (Arcadia environment) [SHO90] based on Ada) or Prolog-like rules (e.g. Merlin [JPSW94]), or follow object-oriented or knowledge-based approaches.

Most process modeling languages are hybrids, combining several paradigms, e.g. Kaiser et al. propose a bi-level process modeling language following both topology-based and constraint-based paradigms [KPB93], called ASL and derived from Marvel. Sutton and Osterweil [SO97], presenting the JIL language, emphasize the need for semantic richness and multiple paradigms in order to support different styles of processes without favoring one paradigm over another one, and they advocate using a combination of both textual and visual (graphical) representations. However, Sutton, Tarr and Osterweil also point out that process requirements and process language requirements ought to be separated [ST095], i.e. processes contain many informal and creative elements whereas process languages should be formal and complete. They propose to use a simpler language that can accommodate the normal cases yet is general enough to admit programming of alternative approaches to special cases. Whether this should be one semantically-broad language, or multiple, independent, special-purpose languages, or core languages with extensions, remains an open issue.

Process modeling languages can also be classified according to the phase in the process they support, distinguishing specification, design and implementation languages [ACF97], although all phases may also be supported by one language. Osterweil proposes to distinguish process modeling and process coding languages [Ost97].

Most process modeling languages provide rather sophisticated support for specific process topics, such as management of constraints and triggers, while largely ignoring issues which we consider as essential, such as simplicity, support for the process history, process linking, and support for both development and deployment, as seen in the section on objectives in the previous chapter. Remarkably, even though the importance of both product evolution and process evolution is increasingly being emphasized [SHO90][BFG93][GB92], most languages focus only on evolution in the present and provide no management support for the process history or for integration of versioning and configuration management issues [ACF97][ABGM92][CLJ93]. Similarly, linking of several processes, and usage of process information beyond development time, e.g. for deployment support, are issues not covered by current process modeling languages. Thus, the DIPS process modeling language design that provides a simpler yet in scope a broader and more integrated language is rather contrary to other approaches.
2.1.2 General Process Modeling Language Properties and Objectives

This section details general design objectives and requirements for most types of process modeling languages (PML) including the one of the DIPS system, all of which should be met by the DIPS language at least to a certain degree, while the next section presents additional project-specific objectives.

As indicated earlier, we will henceforth use only the term “process” and not specifically distinguish between the (real-world) process and the process model (the formalized representation), since many issues discussed will generally apply both to the process and the process model; concerning issues of formalized representation, it is clear that only the process model is meant by the term “process.”

Blum [Blu94] broadly defines the software process to be the activities regarding a software product from the time the need to which that product is to respond is identified until the time the product is retired. At the beginning, we see some overlap with the encompassing business process that leads up to the identification of the need, and specification of the need within the development process itself. At the end of the process, we view deployment as a related but separate task, as explained earlier, however the development process (e.g. maintenance activities) may continue concurrently to deployment tasks until the product is retired.

Formal modeling of processes is essential in order to be able to provide electronic support for processes. It provides process overviews, supports process analysis at least to some degree, contributes to process reuse, and allows essential management and coordination of process steps to be carried out. Once defined, processes may be enacted (executed) and controlled, mostly interactively (e.g. for editing) and partially automatically (e.g. for compiling), in order to create a software product, and processes may be monitored for management purposes. Enactment and monitoring must occur in a manner that does not restrict human creativity, which remains the most important contribution to a process.

As seen earlier, many abstract process models exist. A specific process model is typically defined by combining elements and patterns from previous process models, abstract process models, and specific knowledge and plans about the product to be created. Thus, a process modeling language must be able to cope with many different kinds of processes.

Feiler and Humphrey define a software process as a set of partially ordered steps intended to reach a goal [FH93]. The goal of the process is the product whose creation the process describes. Such a process may have many different properties whose modeling should be supported by a process modeling language. The language should allow abstraction,
modularization and hierarchical structuring of the process, allow
description of concurrency and non-determinism, and of iteration and
evolution in the process; proper support for evolution is now increasingly
regarded as essential [SHO90][BFG93][GB92][GJM91][PC86][Fug93]
[Dow87][MP93]. An adequately modeled process should at any time
answer questions such as which step is to be performed next, by whom
and with what tools, what criteria the product of that step is to fulfill, and
what information or process context is required to perform the step
[PC86][Boe86][SB95][KPB93][Huf96][GJ96a][FKN94][GJM91]. The
process model thus should contain information about activities, artifacts,
and resources (tools, manpower, date/time), and their dependencies,
constraints and contexts. Whether the process model should reflect more
the process or the product has repeatedly been discussed [Dav95].

2.1.3 Project-specific Objectives

While a process modeling language should address the general issues
outlined in the previous section, the DIPS project defines more specific
objectives that should additionally be met by the DIPS process modeling
language.

It should support not only the present process, but also the process history
and the future planned process. Tracking of evolution is essential for
artifact and dependency management and for reference purposes, and
evolution must be allowed during process enactment since it is impossible
to plan every detail ahead, as Parnas and Clements remark [PC86].
McCracken and Jackson even draw an analogy to the Heisenberg
Uncertainty Principle in physics, stating that system development
inevitably changes the environment out of which the need for the
environment arose, therefore change during the process is inevitable
[MJ82]. Furthermore, any combination of planned (future) process
definition and "process programming by hindsight" should also be
possible [GB92], i.e. future electronic notation of an existing informal
process. Although processes typically define creation of new artifacts and
their modification, this would also allow older (legacy) artifacts to be
included in processes to facilitate their reference.

Tracking of evolution should involve dependency, version and
configuration management of artifacts within a process, and versions
should be managed both as revisions (historical versions) and as variants
(parallel versions) [CW96][Moh94]. Additionally, linking of different
processes should be supported to reflect dependencies of artifacts on
artifacts in the history of other processes, which is typically the case when
software components are combined to form larger systems. Furthermore,
issues beyond conventional process support should be addressed, by using
the process as a common data structure for communicating development
issues among different developers, in heterogeneous systems, and among
developers and users. Process information should be used for deployment support, since runtime dependencies are understood as a subset of development dependencies, and software components should be deployed directly out of processes. For these purposes, process linking and process evolution are essential, and the process modeling language should provide reflective features to support evolution of the product, the process, and even the environment including the language itself.

Although these objectives may seem intricate, special emphasis should be laid on simplicity, if the language is to be built in to portable environments and adopted by developers in very heterogeneous and widely distributed settings. Tracking of evolution with support for dependency, version and configuration management calls for a tight integration of repository concepts into the language, and a directly interpretable language for process enactment. It also emphasizes more the topological structure of the evolving product structure, as typically supported by a graphical process modeling language, rather than sophisticated specification of constraints in individual process steps as often provided by algorithmical process modeling languages. Nevertheless, for every process step that involves formal artifacts, it should be possible though not mandatory to specify pre- and post-conditions of the step, by formalizing the dependencies; ideally, this should mainly be supported by optional tools or mechanisms not included in the standard language, in order to keep it simple.

As apparently no existing process modeling language followed similar objectives, this required a new design. Integration of repository, configuration management, evolution and deployment issues in the process calls for integration of product and process representation in a common data structure. This is achieved through structural unity of process and product: the structure of the software product as planned and under development defines the structure of its development process. The above considerations led us to evolve an initial design of the DIPS process modeling language from a Petri net based one [Mar94], which modeled artifacts as moving tokens that proved difficult to manage, to a bipartite 2-dimensional graphical one with static nodes as artifacts that were easier to manage [WSMH94], to a simpler monopartite but 3-dimensional graphical directed acyclic graph (DAG) structured one that additionally allowed tracking of evolution [MWSS96], with process dependencies optionally formalized separately using CHIPS tools, and finally to a language additionally including deployment support issues such as configuration management [SMW96][Sch98]. Such a language should allow any common abstract process model to be applied when defining a specific process model, e.g. the simple stagewise and waterfall models, iterative or cyclic models such as the evolutionary and spiral models, and
models requiring formal methods such as the transformation and cleanroom models [GJM91].

2.2 CONCEPTS

2.2.1 Software Product

While there are different possibilities for structuring a software product, our objective for integration of product and process leads us to draw an analogy to the definition of the software process by Feiler and Humphrey [FH93]. A (software) product thus is a partially ordered set of artifacts (documents), holding all data which are produced during development and maintenance of the product (contract, specification, design, implementation, code, test case, documentation, manual, etc.), not just those artifacts delivered to a customer. Every artifact carries the information of a part of the software product and typically depends on other artifacts. The ordering of the set is defined by these development dependencies. Information from other artifacts is typically required before an artifact can be completed or executed, e.g. during development, an object code artifact cannot be created by compilation before the associated source code artifact is completed, or during runtime, dynamic linking of different modules occurs before execution. Runtime dependencies form a subset of development dependencies, since development activities specify what artifacts may be used together during runtime and additionally link artifacts that are not formally used during runtime, e.g. design documents.

For example, a simple software product called “A” could consist of three dependent artifacts for design, implementation and object code, called “A.Text,” “A.Mod” and “A.Obj” respectively. The product could be visualized graphically by representing artifacts as squares and dependencies as arrows pointing in the reverse direction of the dependency (e.g. “A.Obj” depends on “A.Mod”) (fig. 2.1).

![Fig. 2.1: Sample software product “A”](image)

Artifacts can exist in different versions (both revisions and variants), and can be part of an enclosing compound artifact, thus defining a tree-like hierarchy with atomic artifacts as leaves. A software product can be understood as a compound artifact, similar to compound documents [OHE96], but extended by dependencies among the versioned parts.
An atomic artifact is created using a tool. A tool consists of a data model implementation and one or more views. High data integration requires tools to be strictly separated into models and views. This allows different tools to share data models on a high integration level instead of requiring every tool to interpret data in its own way. Data models contain all data semantics, including operations, and views provide only user interfaces. Thus, if a data model is an object class in the object-oriented sense, an atomic artifact is an instance of a data model, i.e. of the model part of the tool, as in CHIPS-generated tools, thus the artifact is a (persistent) object. The artifact can have formal or informal properties, as will be seen below, and be related to other artifacts in its context (e.g. an object code artifact depends on its source code artifact), and it can query their properties via messages. For other types of tools, artifacts may also be simply documents or data files, and other artifacts may interact with these via their tools or via optional object adapters.

Querying of formal properties among artifacts requires highly integrated tools such as CHIPS-generated tools to be used that understand CHIPS-based messages. Nevertheless, external tools may also be used for informal artifacts or for formal artifacts whose properties need not be formally confirmed, by employing object adapters that allow conventional files created by external tools to be used as artifacts.

2.2.2 Software Process

A (software) process is the dynamic view of a software product under development and defines how its artifacts are created, edited and confirmed. A process is a monopartite directed acyclic graph (DAG), whose structure is induced by its planned product’s structure: its nodes are process nodes that represent process steps (the dynamic view of the artifacts) and are either atomic or compound: an atomic process node holds one associated atomic artifact with the tools used and permits only one developer to update it at a time, thus defining the granularity of process steps, while a compound process node holds a set of process nodes (atomic or compound), defining a process hierarchy, whose (recursively) associated artifacts make up the compound artifact. Every node obtains a unique identity (process-wide unique number) it retains and which may be used to identify it within the process at any time. Furthermore, nodes can hold numerous attributes containing metainformation about the artifacts or nodes they contain, such as node state (planned, editable, confirmed, as will be explained in the section on enactment), tools, roles, access privileges, date/timing information, etc., and different process views will typically be used to display selected attributes only. The DAG’s edges are arrows representing the artifacts’ directed dependencies and point in the direction of development workflow (which is the reverse direction of the dependency), thus specifying at the same time the development order and the product structure. Specification of a partial ordering per definition
does not allow cycles, thus leading to a DAG structure. In order to describe the DAG and its operations, a graphical notation is best suited to the process modeling language, although an alternative textual notation is also envisageable. While usage of a DAG prohibits cycles in the process definition, during enactment process iteration operations are available that allow parts of a process to be repeated, as detailed below, since iterations are frequently required in typical processes, and the DAG structure may itself be modified by process evolution operations, as will be seen below.

For example, the process for the simple software product “A” seen above (fig. 2.1) could consist of three dependent atomic nodes for design, implementation and object code, holding the respective artifacts of the software product. The process could be visualized graphically by representing nodes as squares and dependencies as arrows pointing in the reverse direction of the dependency (fig. 2.2).

![Diagram of a process](image)

**Fig. 2.2: Sample software process for product “A”**

Thus, the product and the process are technically the same data structure. It is referred to as product when the static view of artifacts and dependencies making up the product and its structure is meant, and it is referred to as process when the dynamic view of dependencies defining the order in which the artifacts are created, modified and confirmed is meant. The process nodes are linked by dependencies and contain metainformation about their planned or associated artifacts, similar to but much more comprehensive than directory information for files in a filing system. A process represents a related set of artifacts typically making up one software product or component, and different processes may be linked together to represent dependencies among several products. Developers may decide whether a product or set of products should be represented as one large or several smaller linked processes, based on their desired degree of interaction.

In the hierarchy of compound nodes containing further nodes, there is always an all-enclosing top compound process node containing all other process nodes with all their versions and all dependencies (e.g. “A” in fig. 2.2). With all artifacts attached to their respective atomic process nodes within, the top compound process node can usually be used as a synonym to the process, except that technically the process manages some additional information that concerns several or all process nodes in the process
(configurations, variants, process-wide access privileges, roles, etc.). Any compound process node may become a separate process, if it is removed from enclosing nodes, since the hierarchical structuring assures that the behavior of any contained atomic or compound process node is similar. The process without attached artifacts is called the process structure.

Every process node, atomic and compound, has an associated responsible developer who is the only one having write privileges on the node (and associated artifact for atomic nodes). Thus, the granularity of process steps requires that only one developer may update a node or artifact at any one time, but several developers may have read access to the same node or artifact simultaneously, while it is not being updated, and several developers may update different nodes or artifacts concurrently. Also, a node's responsible developer may be modified during (subsequent) process definition, to allow sequential write access to a node or artifact for several developers.

2.2.3 Metaprocess

While the product under development and its evolution is managed by the process, the process support (the tools and the environment used) and its evolution is managed by the metaprocesses. As seen earlier, various levels of metaprocesses are envisageable, and in order to provide complete process integration and not to complicate the process modeling language, a fully reflexive language is required, i.e. the same language is used to describe both metaprocess and process issues. Again, the same data structure is used as for the product and the process, with metaprocess information added to the process nodes as additional attributes and additional dependencies, possibly on a metalevel. Thus, metaprocesses are initially ordinary processes, but after completion they may be referenced by other processes in their role as metaprocesses, e.g. by using metalevel dependencies, as will be detailed below.

Metaprocess issues include description of the environment holding the process, i.e. tools and framework, via their respective processes, including precise versioning information, and evolution of the environment including tools and framework, whereby evolution of the framework includes evolution of the language itself.

For example, for the simple software process "A" seen above (fig. 2.2), the tools used could be specified for each of the three process steps (the atomic nodes), i.e. a design editor, a source code editor and a compiler, respectively. In the process, this could be visualized by metalevel dependencies on the three processes implementing the tools (more precisely, metalevel links, as will be detailed below, to indicate that the tools are implemented in separate processes), shown as dashed arrows labeled with "meta" (here, a square could denote an atomic node or a
compound node containing further nodes, even a whole process) (fig. 2.3). An additional metalevel dependency, not shown, could lead to the process implementing the framework for the software engineering environment holding the tools used to carry out process “A.”

![Diagram](image)

*Fig. 2.3: Sample software process “A” with 3 tool metaprocesses*

### 2.3 PROCESS DEFINITION

As detailed in this section and the following ones, the process modeling language consists of definition operations and rules and of enactment operations and rules, all of which are process-independent, and it additionally supports process-specific composition rules where available. As the language is not yet fully defined, the specification given here still allows various details and extensions to be defined more precisely in future; also, the language is under constant evolution reflecting experience gained with the prototype implementation. However, the language presented here is very similar to the one currently implemented, and while some features (derived from recent experience) are not yet fully implemented, the prototype demonstrates that its implementation is feasible and that already the limited-feature implementation is usable. It is envisageable that in future more precise descriptions could specify the language on several different abstraction levels to prevent a single detailed specification from becoming unmanageably complex.

At any time, a process or a selected set of nodes within a process can be in one of two modes:

- definition mode;
- enactment mode.

In *definition mode* (the initial mode a new process is in), definition operations may be performed (adding and removing nodes and dependencies, to modify the process state space), and in *enactment mode*, enactment operations may be carried out (to change node states, as will be seen below, such as planned, editable, confirmed, and rolled back). Transitions from one mode to the other is possible at any time provided the respective rules are satisfied, thus a process may still be modified after enactment. A process is initially defined (created) and subsequently modified in *definition mode* by performing definition operations, which
are restricted by definition rules and composition rules. Satisfaction of definition and composition rules is checked upon transition of the process or the nodes involved to enactment mode and is required for the successful transition. Transition back from enactment mode to definition mode requires the process or the nodes involved to be in state planned (not yet enacted, or rolled back, as will be explained below). Transition from one node to the other may occur implicitly if the developer attempts to initiate an operation belonging to the other mode, which causes the transition rules to be checked and the mode to be changed automatically before the operation is carried out, if permitted.

![Fig. 2.4: Process definition of sample process “Component”](image)

In order to define a process, atomic and compound process nodes are created and linked by dependencies to reflect the structure of the planned software product, which must form a 2-dimensional hierarchical DAG. Fig. 2.4 illustrates the definition of a process for a small software component consisting of two modules A and B, with B importing A, consisting of a top compound node “Component” containing several atomic nodes and two further compound nodes, “A” and “B,” that again contain some nodes (several more examples will be presented in the discussion chapter). All dependencies are directed, and for every type of dependency, an opposite type of dependency exists. For every dependency established among two nodes in the process, the opposite dependency is also automatically established among the same two nodes in the opposite direction; such reverse dependencies would not strictly be required as they do not provide any new information, but they are defined since they simplify navigation and other operations on the process. If node B depends on node A, this signifies that there is a predecessor dependency from B to A (not visualized), since A is B’s predecessor node, and a successor dependency from A to B (visualized as an arrow). Thus, the predecessor dependencies are the ones that define what parts of the product are dependent on each other and that thereby determine the possible order of workflow in the process (which is in the opposite
direction), whereas the opposite successor dependencies provide the
directed edges in the product's DAG structure and visualize the possible
order of workflow. Furthermore, hierarchical structuring is possible
since compound process nodes can contain other (atomic or compound)
process nodes. Additionally, copying of parts from the same or another
process is possible in order to reuse existing definitions.

2.3.1 Dependency Semantics

For every atomic node, the tool(s) used to perform the process step must
be specified (respectively the messages used for tool-artifact
communication), and further attributes may be specified per process or
per compound or atomic node such as roles, access privileges, date/timing
information, etc. For every atomic node, if a tool's development process
is available, the tool may be specified by a special kind of dependency
from the node to the tool's development process, a metalevel process link,
as detailed below, which allows the tool to be precisely specified,
including its version. Additionally or alternatively, the tool's command
(or message) to be invoked may be specified textually, and additional tool-
specific parameters may be set in node attributes, e.g. to define the
artifact type more precisely, or to hold compiler options. CHIPS-based
artifacts may internally distinguish a large number of tool-specific states
(reflecting artifact properties), e.g. parsed successfully (token list and
structure tree internally created), tool-specific semantic property fulfilled
(attributes in the structure tree internally assigned [Mar94]), or compiled
successfully, and the tool-specific dependency semantics then define how
these artifact-specific states translate to states known by the process
(which will be detailed in the section on enactment). Typically, the
artifact's node's state editable will not require any artifact-specific state to
be fulfilled, whereas the node's state confirmed will require a particular
artifact-specific semantic property to be fulfilled (where defined), usually
the most high-level one. For messages or commands invoked, different
ones may be distinguished: ones that only query artifacts' states or other
information, ones that check an artifact's property and then set its state
accordingly (e.g. parsed successfully), and ones that modify or produce
new artifact contents (e.g. compilation). Continued editing of an artifact
causes its state to be reset to an initial one, e.g. editing a source code text
causes a property such as parsed successfully to be invalidated, thus
requiring it to be checked again after editing is concluded.

In order to (optionally) formally specify dependency semantics, every
atomic node may have a creation function (previously known as
originator function in our project) and a required predicate (previously
known as confirmer function in our project), which are typically
provided by the tool(s) associated with the node. These are also held in
node attributes and will be used upon enactment, as detailed in the next
section, to create and validate the associated artifact and thereby provide
dependency semantics. Creation function and required predicate together form a specific signature (node type), which itself requires a process context consisting of nodes with specific signatures. A node's predecessor nodes form its process context, i.e. these are the nodes that provide the input artifacts for its creation function, and one of the nodes in the context, the creation node, is responsible for initially creating the node's artifact. E.g. a compiler as a creation function respectively its object code artifact requires a source code artifact (from the creation node) as input, that node's creation function thus must be a source code editor, and an object code artifact may depend on further symbol file artifacts, a source code artifact may depend on interface definition artifacts and specification artifacts, etc. The required predicate, e.g. parsed successfully, used upon confirmation during enactment as explained below, specifies the artifact-specific state an artifact must attain in order to reach the node's state confirmed, i.e. it defines the translation of an artifact-specific state to a state known by the process.

Thus, in order to define a process, process-specific composition rules must be satisfied that require respective signatures, if existing, to match, i.e. a node that forms part of the context of a successor node must have a signature that matches the signature expected in that context. Thus, when combining (copies of) parts of existing processes (typically as empty templates) and/or newly defined sets of dependent nodes to create new processes, such process composition activities may be compared to assembling a puzzle from individual pieces. The details of signature specification (perhaps using regular expressions), and the semantics of signature matching, taking into account a possible type hierarchy of signatures and involving different relationships of atomic and compound nodes, have not yet been precisely defined and demand further research; these issues are beyond the scope of this thesis.

In an example we (partially) implemented, for a real-time system, a formal required predicate based on CHIPS-generated tools for formal languages statically checks execution times of primitive-recursive functions against their maximum times formally specified in a preceding artifact; execution times for partially-recursive functions are specified as "infinite." High data integration allows both the editor with timing analysis functionality for the Oberon module implementation and the Oberon compiler's code generator to share the Oberon scanner and parser, thus assuring that both have the same understanding of the Oberon language and specifically of execution times for primitive-recursive statements.

A process called "Oberon Real-time Component" using these tools contains two nodes (fig. 2.5a) (within the top compound node "Oberon Real-time Component"): A "TimingSpec Document" containing maximum
allowed execution time specifications (fig. 2.5c) per procedure for a particular Oberon module, and a dependent “Oberon Module” containing the Oberon module implementation (fig. 2.5b) that should satisfy the specified execution times. Since the tools are of particular interest here, fig. 2.5a also shows three involved tool (meta-) processes: It shows an “Oberon Timing Environment” process where a CHIPS-generated editor/compiler/timing analysis tool is implemented (consisting of Oberon scanner, parser, editor, compiler, and timing analysis semantics) that is used to create, edit, compile and time-check the Oberon module implementation; thus, the “Oberon Module” node has a metalevel dependency (or link) to the tool’s process indicating that the tool is used by that node (also specifying the tool’s required version). Fig. 2.5a similarly shows a “Timing Specification Environment” process where a CHIPS-generated timing specification editor tool is implemented (consisting of TimingSpec scanner, parser, editor, and semantics for querying), used to create, edit and query the timing specifications document, as specified by the metalevel dependency (or link) from the “TimingSpec Document” node. Finally, fig. 2.5a depicts a “TimingMsgs” process (without showing contained nodes) that implements messages used by both tools to communicate with each other.

Fig. 2.5a: “Oberon Timing”: sample process “Oberon Real-time Component” with tool metaprocesses
MODULE Demo;

PROCEDURE A(m: INTEGER);
VAR i: INTEGER;
BEGIN
  i := 0;
  WHILE i < m DO
    INC(i)
  END;
END A;

PROCEDURE B(x: INTEGER; VAR a, b: INTEGER);
BEGIN
  IF x < 0 THEN
    a := 3; b := 4;
  ELSE
    a := 1; b := 2;
  END;
END B;

PROCEDURE C(x, n: INTEGER): INTEGER;
BEGIN
  RETURN x * n + 1;
END C;

END Demo.

Fig. 2.5b: "Oberon Timing": sample Oberon module

TIMING Demo;

PROCEDURE A INFINITE
PROCEDURE B 30
PROCEDURE C 20

END Demo.

Fig. 2.5c: "Oberon Timing": sample execution time specifications for the Oberon module of fig. 2.5b

For process definition, a node with the "Oberon Timing Environment" tool requires a predecessor node in its process context with the "Timing Specification Environment." Then, during process enactment, i.e. development of the Oberon module, the specified execution times may be queried by the timing analysis function, and confirmation of the Oberon module node is only possible when the specified times are being met, since this is defined to be the node's required predicate. More precisely, the "TimingSpec Document's" creation function is "CHIPSCreators.NewCreatorMsg" with parameter "ProcTimingStrucs.NewEnv", and its required predicate is "ProcTimingStrucs.NewParseMsg;" thus, a new artifact may be created in the "ProcTimingStrucs" environment (here called "Timing Specification Environment") and it must be parsed successfully in order to be confirmed. The "Oberon Module's" creation
function is “CHIPSCreators.NewCreatorMsg” with parameter “OberonTStrucs.NewEnv” to allow a new artifact to be created in the “OberonT” environment (here called “Oberon Timing Environment”) that holds editor, Oberon scanner, parser, compiler (not implemented), and timing analysis (implemented); its required predicate is “TimingMsgs.NewCheckTimingMsg,” whose execution performs the timing check, as follows: As will be seen in the implementation chapter (fig. 4.4g), a “TimingMsgs.CheckTimingMsg” is sent to the Oberon module artifact, which in turn causes its Oberon timing analysis tool to send a “TimingMsgs.TimingMsg” per implemented procedure to the associated predecessor timing specification artifact to query the specified timing information required for the timing check; the obtained specified times are compared to the Oberon timing analysis initiated or performed by the “OberonTiming Semantics” tool component on the Oberon source code (i.e., scanning, parsing, and static calculation of execution times), and the Oberon module is confirmed, if all specified timing restrictions have been met.

While the semantics of dependencies thus may be precisely specified using creation function and required predicate, only one dependency (plus its respective opposite in the other direction) is supported among a pair of nodes, respectively a dependency may not have multiple independent semantics. Support for multiple, more fine-grain or other types of dependencies is an issue for further research, although such applications are envisageable [Mey85][PC90][SS95], e.g. different types of dependencies could form different (sub-) graphs among a set of nodes representing different kinds of specific information, displayed with appropriate user interfaces separately showing only selected dependency types. For instance, such a subgraph could hold information only about Y2K compliance of dependent components.

In order to support variants, we envisage dependencies to be grouped in sets of alternatives, so that during enactment, only one dependency in the set is used at a time.

2.3.2 Definition Operations and Rules

Definition operations include:

- create, modify, delete process (automatically includes top compound node);
- create, copy, modify, delete atomic or compound node in a compound node (automatically handles sub, super dependencies, and involves contained nodes and dependencies);
- create, copy, modify, delete (pred, succ) dependency among two nodes;

...
• create, copy, modify, delete attribute of a node or of the process.

Definition rules include (if not specified, node means compound process node or atomic process node):

• a process contains at least one compound node (the top or outermost node);

• every node has exactly one superior (containing) compound node (defined by a super dependency), except for the top node;

• a compound node may contain 0..n subordinate nodes (atomic or compound) (defined by sub dependencies);

• a node may have (predecessor) dependencies to 0..n predecessor nodes;

• a node may have (successor) dependencies to 0..n successor nodes;

• all dependencies are directed, and the graph of the transitive hull of all predecessor or of all successor dependencies in the process may not contain cycles, thus representing a strict partial ordering of nodes;

• a predecessor or successor dependency among a node within a compound node and a node outside the compound node requires a similar dependency among the compound node and the outside node, but not vice versa.

2.4 PROCESS ENACTMENT

After transition to enactment mode, a process may be enacted (executed) immediately, since enactment is by interpretation and there is no separate process coding, compilation or instantiation. Reflecting the dynamic nature of a process, during enactment its nodes traverse a sequence of three different states, which are visualized by different colors (respectively grey-scales for black-and-white visualization or printing). Enactment operations thus involve transition of nodes from one state to the next (and back, as detailed in the next section), and enactment rules specify that the states must be traversed in sequence (as illustrated by the state machine in fig. 2.6; rollback will be explained in the next section). For atomic nodes, the states are the following:

☐ planned (red respectively white): no artifact is assigned to the process node; all nodes are initially in this state after definition;

☐ editable (yellow respectively grey): an artifact has been created and assigned to the process node and is being developed;

☐ confirmed (green respectively black): the assigned artifact fulfills the required predicate, if specified in the process node, and the artifact is confirmed to be completed; it may not be modified any more.
Artifact creation and editing is performed using the process node’s specified tool, and may involve other artifacts it directly depends on in its process context, but only after their confirmation. Artifact editing is typically the main task in a process and involves creative work carried out by developers that cannot be automated, although for some artifacts there will be no further editing after creation, e.g. for object code created automatically by a compiler.

Confirmation involves automatic or manual validation (checking) of the formally or informally specified required predicate, plus a manual specification by the responsible developer (as an electronic signature) that the validation has successfully been carried out and the artifact is released, i.e. ready for use within the process, e.g. by dependent artifacts. Ideally, the required predicate is specified formally (by a highly integrated tool) and it may even be possible to check it automatically (to perform verification), although formal specification is optional and in many cases confirmation will simply be done manually, i.e. according to the developer’s well-founded opinion.

Since transition from state planned to state editable involves creation of an artifact, this may only proceed if those directly preceding (predecessor) nodes that are required for artifact creation (if any, as defined by the tool involved) are in state confirmed, e.g. a source code artifact must be confirmed before it may be compiled to create an object code artifact (thus, a properly integrated compiler will not compile the source code artifact unless it has been parsed successfully, and confirmation of the source code artifact will be associated with the condition of successful parsing). However, those direct predecessor nodes (if any) that are not used for artifact creation may be in any state, even only in state planned. For transition from state editable to state confirmed, the transition rule is more strict: all predecessor nodes must be in state confirmed, even if they are not formally involved in confirmation, since confirmation expresses that dependencies on all preceding artifacts are valid and thus that these may not be modified any more, which is the case only in state confirmed.

![Fig. 2.6: Process enactment state transition state machine for atomic nodes](image)

**Fig. 2.6: Process enactment state transition state machine for atomic nodes**
These transition rules define a partial ordering of workflow in the process, with different nodes in the process progressing through different states illustrated by progressing coloring, in analogy to moving tokens in a Petri net, yet with assigned artifacts remaining with their atomic nodes. E.g. an object code artifact is created by a compiler using the source code artifact it depends on after the source code artifact has been completed, i.e. confirmed. Or a source code artifact may depend on another source code artifact (e.g. due to module imports): although these may initially be created independently, they must be completed (confirmed) in sequence. As dependencies define only a partial ordering of workflow, enactment concurrency is possible among those nodes in a process that are not dependent on each other.

State transitions of compound process nodes are similar, except that they are governed by the states of their contained process nodes instead of by artifacts, as compound process nodes have no directly associated artifacts. A compound node is in state planned or in state confirmed if all its contained nodes are in the same respective state, otherwise it is in state editable. This ensures a consistent behavior across the containment hierarchy.

Fig. 2.7: Sample enactment sequence of “Component” process

Fig. 2.7 illustrates a possible enactment sequence on the “Component” process (defined in fig. 2.4): First, node “Specification” is created, edited, and confirmed, then node “Design” is similarly enacted until confirmation, and node “A Source” is created and being edited (left half of fig. 2.7). Then, “A Source” is completed and confirmed, whereupon “B Source” may be created; meanwhile, enactment of nodes “Documentation” and “Test Data” may proceed concurrently. Enactment of the second node
within compound node “A” continues until its confirmation (right half of fig. 2.7), and the other nodes are also continued, until all nodes in the process are confirmed.

For alternative dependencies representing variants, we envisage that all combinations of direct dependencies may be confirmed, and at least one confirmation of alternatives is required. Thus, for variants, registration of the state as confirmed must additionally include the combination, as only confirmed combinations may participate in configurations (as detailed below). E.g., given an artifact depending on either an artifact representing platform A or B, and also depending on either an artifact representing framework C or D, confirmation could occur for the combination of variants depending on platform A and framework D. Unlike with normal dependencies, where a node may be either confirmed or not, with alternative dependencies a node may be confirmed a large number of times, for different combinations of alternatives, therefore a pitfall is the potential combinatorial explosion of possible confirmations (e.g. A-C, A-D, B-C, B-D), thus further research is required to define practical rules and limits.

2.5 PROCESS DIMENSIONS

Besides defining and enacting processes, it must also be possible to modify the results of definition and of enactment operations, since change is inherent in the process and cannot be avoided. As far as described above, process definition and enactment involve only two process dimensions, since a 2-dimensional DAG structure may be defined that shows dependencies and possibilities for concurrent enactment. Process hierarchy does not provide an additional dimension, but it provides a structuring facility that allows different views of the same process to be used, using selection and zoom operations.

Fig. 2.8: Various 2-dimensional views on the “Tool” sample process; left: complete process; middle: top hierarchy only; right: contents of compound node “Impl” only

E.g. a view could show only a selected compound node and its contents, or a large view could show the top compound node with all contained
nodes across several hierarchical levels (with directly surrounding nodes (if any) illustrated in smaller size), or visualization of the contents of compound nodes would be omitted at a specific level of detail (with compound nodes distinguished from atomic ones e.g. through a larger size or a different shape or symbol). Fig. 2.8 illustrates various possible 2-dimensional views on the “Tool” sample process.

Different types of process illustrations are used throughout this thesis, 2-dimensional sections and 3-dimensional views in perspective (showing process history, as in fig. 2.9 explained below), and manually drawn graphs (to show specific details) as well as views drawn automatically by the implemented system’s graphical user interface (which does not yet visualize all issues discussed).

With the goal of providing comprehensive process management, it is essential that the complete process history is recorded, at a certain granularity as defined by the individual states. Developers frequently require information about older versions of artifacts and their dependencies, and new versions are typically based on previous ones, for compatibility reasons and to reduce redo efforts. The process is the primary instrument to register all information (including contents) about an evolving and existing software product, and to share it among different developers, and even among developers and users. If repository issues and version and configuration management are to be integrated into the process in an effective way, it is essential that older revisions of artifacts and of the process itself remain available at any time. Integration of configuration management is also a requirement for deployment support.

Thus, for the purpose of recording the process history, the process modeling language provides a third dimension, the history dimension, which is managed by process iteration and process evolution operations, as described next. In different history layers, it provides snapshots of parts of the process at different times. Process iteration and evolution operations may be regarded as higher-level operations that allow the previously presented process enactment respectively definition operations to be reset and therefore redone where desired (retaining previous results in the process history), as these operations cannot be repeated by themselves. However, unlike the previously presented process enactment or definition operations, process iteration or evolution operations may themselves be redone, therefore there is no need to provide yet higher-level operations to enable these to be redone. All entities in the process history retain their identification and may thus continue to be referenced as before, and all dependencies of versions (revisions and variants) are recorded in the process history without the need to assign version numbers, unlike conventional version management systems that rely on
usage of version numbers to identify different versions, since conventional files lack dependency information.

2.5.1 Process Iteration

While process enactment includes forward transition of nodes from one state to the next, as described above, it also allows reverse transition, from a state to the previous state or directly to state planned. This is called process iteration or rollback, since it allows enactment operations to be undone (rolled back), and repeated again, thus supporting iterative process models. It occurs in enactment mode and allows artifacts to be revised. It causes the selected artifacts to be invalidated with respect to the current flow of the process, and it also causes all their dependent ones (if any) to be invalidated, i.e. those directly or indirectly accessible via successor dependencies, as their creation functions and required predicates are not valid anymore and must (also) be reconfirmed in order for them to be usable again. Thus, a revision will typically also necessitate a modification of other dependent artifacts, therefore they cannot remain in the process as confirmed. A rollback normally causes the previous (invalidated) results to be stored read-only in the process history, as detailed later, unlike a rollback of a database transaction that typically does not retain uncommitted results [GR93], although in certain cases a simple rollback is also available that does not store previous results in the process history but simply resets existing nodes and thus allows their results to be overwritten.

Invalidation of artifacts causes them to be removed from the current flow of the process, i.e. placed in the process history, as explained below. They may then not be modified anymore, but they do not lose their identity and all their information still remains accessible for reference purposes. A selective rollback, causing an artifact initiating the rollback to be invalidated for some dependent artifacts only but not for others, is not yet available. This would effectively allow work to continue from the initiating artifact in two revisions, one affected and one unaffected by the rollback. However, this can also be achieved by applying an appropriate process evolution operation, detailed below, on the affected nodes to duplicate them before the rollback, and performing the rollback on one set of nodes only. Additionally, nodes in the process history, while not modifyable themselves anymore, may again become predecessor nodes of future newly defined nodes.

Invalidation of dependent artifacts is orthogonal to the issue of independent compilation, which avoids the recompilation of client (importing) modules if a used (imported) implementation changes without modifying its symbol file information. If the employed programming language offers the benefits of independent compilation, it can be determined automatically during (re-) creation of an object code artifact
whether the invalidated artifact may be copied unchanged from the newly formed history layer to the current layer, or whether its associated source code artifact has to be recompiled to create a new object code artifact. For the subsequent confirmation, while all (current layer copies of) invalidated artifacts should be reconfirmed manually, a development policy is envisageable where in certain cases reconfirmation may be automated, e.g. if the artifact has not been modified.

2.5.2 Process Evolution

Similarly, a process, once defined, may need to be modified later on, which is called process evolution. This occurs in definition mode and may be performed even after some parts of the process have already been enacted. Thus, process evolution means modification of the process after it has been transferred to enactment mode at least once, as opposed to modification of the process during initial process definition. While process enactment governs product evolution, i.e. the process manages the evolving software product, process evolution allows the process itself to evolve. Process evolution thus involves modification of the process state space, since nodes may be added or removed, as opposed to enactment (including process iteration), that involves modification of process states of existing nodes. Unlike process iteration, process evolution has not yet been designed and implemented in detail. Similar to rollbacks, detailed below, it will require a precise definition of what nodes (or perhaps parts of nodes) are involved in a process evolution operation and what the consequences are for the involved process parts as well as the other ones.

2.5.3 The Third Dimension

A process iteration or a process evolution operation is in principle performed on a set of selected nodes in a process, although the current implementation limits initiation of a rollback to a single node per operation, which in itself may already affect a large number of dependent successor nodes. With every such operation, a new history layer is created in the process: a new copy of the affected part of the 2-dimensional DAG structure is created (with all nodes and dependencies, and artifacts where applicable) and inserted in its place in the topmost or current layer of the process, while the old copy moves down the history dimension to create a new layer in the process history. Thus, all change is recorded and old revisions of process parts continue to be accessible, though not modifiable, since the process history is read-only.

All definition and enactment operations must occur on the current layer, except that definition permits new dependencies to be appended to nodes in a history layer, in which case they become a temporary current layer again, but the previously read-only nodes must remain read-only. Also, process components (nodes and dependencies) may be copied from the
process history to the current layer to be reused, without affecting the
originals which remain in the process history.

As seen earlier, a node in the current layer can be in one of three states: *planned*, *editable*, or *confirmed*. A node in a history layer can be in one of four states: *editable* or *confirmed* each combined with *full* or *partial rollback* (*planned* nodes cannot be rolled back). Nodes directly affected by the rollback are *fully rolled back*, and other nodes may be *partially rolled back*, e.g. compound nodes not directly affected by the rollback but containing some nodes to be fully rolled back.

A process iteration (rollback) operation affects all selected (atomic and compound) nodes, which cannot be in *state planned* (since planned, i.e. inexistant, artifacts cannot be revised), and all of their successor (atomic and compound) nodes that are not in *state planned*, up to (excluding) nodes that are in *state planned*. Thus, from the selected nodes, all directly dependent (successor) ones that have been enacted already are included, and (recursively) all directly dependent ones from these that have been enacted already are included, since their dependency requires that they all must also be revised (this may even include compound nodes whose contained nodes have not all been enacted yet). The affected nodes thus form a contiguous area of the process that does not contain nodes in *state planned* (other nodes in the process remain unaffected, even dependent enacted nodes, so long as they are separated from the area by intermediary unenacted nodes, i.e. nodes in *state planned*). The originals of all the affected (atomic and compound) nodes are moved unmodified to a new history layer in the process history, and the copies are invalidated (reset to a selected previous state: *planned* or *editable*) and may be revised and reconfirmed. Thus, fully rolled back nodes are transferred to the process history and their copies are inserted in the current layer, whereas partially rolled back nodes remain in the current layer and their copies are moved to the history layer.

Besides the aforementioned affected nodes, all compound nodes containing these up to the minimal single compound nodes containing all of them are also affected (unless the selection specified the top compound node, in which case there is no further containing node). For all these compound nodes (if any), new copies are created and inserted in the new history layer in the respective places in order to duplicate the hierarchical process structure around the originally affected nodes now in the process history. The originals of these compound nodes remain in the current process layer (since they are only partially rolled back), together with the other nodes they contain (if any) that were not affected by the operation. The new copies of all affected nodes are inserted in the current process layer in the respective places to precisely duplicate the original process structure as it was before the operation.
Older/newer dependencies are inserted between the nodes that have been moved to the process history and their respective new copies in the current process layer (and vice versa), in order to register the revision relationships, and the moved (respectively copied) nodes in the new
history layer may continue to have older/newer dependencies with the next older history layer in the process history. Every history layer receives a unique history layer number per process, and every node knows its history (or current) layer number. Additionally, pred/succ dependencies are established among the immediate predecessor nodes of the selected nodes and the selected nodes to reflect the pred/succ dependencies that existed before the operation. Thus, these predecessor nodes now have both the original successor nodes as successors, now in the process history, as well as their respective new copies in the current process layer, and these can be distinguished by their history layer number.

Fig. 2.9 illustrates the three process dimensions in a sequence of enactment and rollback operations carried out on the “Tool” sample process. After process definition, some enactment operations are performed (1.), then a rollback is performed on node “Src” (2.) The new copy of “Src” is enacted again, as well as other nodes (3.), followed by a rollback on node “Spec” (4.) Enactment (and perhaps rollbacks) may then continue until all nodes are confirmed (on the current layer). The top compound node is always the hierarchically outermost node (here the “Tool” compound node) on the current (newest) history layer; different “Tool” nodes could assume the role of top compound node at different times, since a “Tool” node may be transferred to the project history by a rollback operation and a new one may be created on the current layer (although in the examples in fig. 2.9, the “Tool” node is not fully rolled back and therefore the same one remains on the current layer, with a copy being created in the project history during rollback on node “Spec” (4.)).

For process definition operations, management of the third process dimension is similar to process iteration operations, although the specification is still a matter of research, as mentioned above. Process definition operations may only occur with nodes in state planned, therefore, a rollback operation may be necessary first to reset enacted nodes to state planned before commencing definition operations. With copy or evolution operations, new copies of nodes are typically also reset to state planned, although it is also possible to copy artifacts, thus their nodes would be in state editable, whereas copies of nodes in state confirmed are always reset to state planned or editable, i.e. (re-) confirmation is required in every case.

Both iteration and evolution operations record the process history in the same (third) process dimension. However, with the rules presented so far, revisions of process structure (due to evolution operations) and of artifacts (due to iteration operations) are easily distinguished in the process history, since process nodes of structural revisions must always be in state planned, while nodes of artifact revisions are never in state
planned. Nevertheless, when defining process evolution operations, it may be desirable to introduce a combined process iteration and evolution operation that performs the required rollback prior to process definition and only creates one new history layer for the combined operation (to save creating an additional history layer if it differs from an older one only in state but not in structure). In this case, a node's state will not only show the state it had before the operation, but additionally show the reason for the node to have become part of the process history (i.e., iteration or evolution). In certain other cases it may also be desirable to perform process iteration (rollback) or process evolution operations without storing the process history, i.e. by deleting or overwriting previous results. Further research is required to determine when this should be allowed and how it is to be supported in the process modeling language, since it is contrary to the general goal of tracking the complete evolution of products and processes, but it may serve to minimize the amount of data in the process history, e.g. by ignoring intermediate results if differences between results (artifacts and/or nodes) in different layers of the process history are small. And for those selected results that should be stored in any case, storage space may additionally be saved by applying appropriate implementation techniques, independent from the process modeling language, e.g. a delta technique may be employed to store only differences of similar artifacts (typically, revisions of the same artifact) instead of always storing artifacts in whole.

**Key**

- **node (atomic or compound)** (states not shown)
- **variants:**
  - alternative successor dependency from A to B,
  - alternative predecessor dependency from B to A
- **older dependency from A to B,**
- **newer dependency from B to A**

![Diagram](image)

**Fig. 2.10: Example of revisions as variants**

Although variants (alternatives, i.e. parallel versions) are not yet precisely specified in the language, we envisage that they should be possible not only in the 2-dimensional process definition or process layers, but also that revisions (historical versions) may be used as variants. E.g. not only should it be possible that an artifact depends alternatively on variant A or on variant B of a predecessor artifact (e.g. representing two different operating system platforms), but it should also be possible to depend alternatively on revision 1 or revision 2 of an artifact (e.g. older and newer version of the same operating system platform), as illustrated in fig. 2.10. These issues require further research; they signify that confirmation rules will become more complex, since selective
confirmations (with different combinations of alternatives) will need to be supported, and there is potentially a combinatorial explosion of possible combinations. Nevertheless, it is of great practical importance to be able to define a component that depends not only on one revision but that alternatively depends on a large number of revisions of a predecessor component, e.g. all previously released versions of an operating system.

2.6 PROCESS LINKING

2.6.1 Interprocess Dependencies

Process linking allows dependencies to be established among nodes in different processes, called interprocess dependencies, thus linking different processes together (as illustrated already earlier in figs. 2.3 and 2.5a). This allows component dependencies to be reflected in their processes, e.g. when different dependent components are defined by different processes. This is of use not only to describe “systems of systems” made up of numerous components, but practically to manage all types of software, since most software products are dependent on other software products, even if it is just an application depending on an operating system.

These dependencies are established and used by developers, e.g. during system specification, design, and implementation, and they are required in order to assemble multi-process configurations of software components, which in turn are employed by users during software deployment. Thus, developers establish interprocess dependencies for themselves, for other developers, and also for users. In this way, developers pass on essential dependency information to other developers and to users, instead of these having to research this information independently, since the original developers have the most complete knowledge about this information. Ultimately, many linked processes in a widely distributed setting will form several large “global processes.” Links allow developers to perform read-only operations on other processes as within a single process, e.g. to perform interprocess browsing, execute queries, and to access information for carrying out artifact confirmation. Besides supporting development, process linking is essential for software deployment too, as will be seen below, where it is required to validate configurations and allow appropriate software to be delivered via network, installed locally and even invoked automatically.

Process linking not only allows a development team’s different processes to be linked together, but more importantly enables linking to other processes where access is more restricted, e.g. to earlier processes of existing products, to other teams’ processes, perhaps at a distant location, and to processes of other companies. Such process linking supports the management of virtual organizations, where different developers or teams cooperate in a loosely coupled widely distributed scenario. And even
where the same team manages several processes, process linking ultimately allows the granularity of processes to be decided by developers themselves, depending on their desired degree of interaction. A process need not represent one product, e.g. a large product may be represented by several smaller but linked processes, or several small products may be managed in a single process; the process modeling language does not enforce a specific policy but instead allows developers to define representations best suited to their development culture.

Interprocess dependencies are similar to normal (intraprocess) dependencies, except that they may only be established on confirmed (read-only) nodes, and are usually not registered in the nodes they depend on (since these may be in another organization’s process where the developer has no write privileges), and therefore definition or enactment operations are not propagated along these dependencies. This is appropriate, since e.g. the developer of a new editor component depending on a component framework defined in a different process does not desire the editor component to be automatically moved to the process history when the framework’s developer releases a new revision. The editor component continues to depend on a specific (uniquely identifiable) released revision of the component framework, regardless of whether that revision of the component framework has meanwhile been moved to its process history. Per default a developer has no write privileges in another process, thus that process is not affected when a link is specified on it, although optionally a mechanism could be established that registers such links and thus allows notifications to be sent along them, e.g. to inform developers about new releases. Nevertheless, for interprocess dependencies the decision whether and when to make use of a new release remains with the developer using it, not with the one providing it, unlike for intraprocess dependencies that automatically invalidate dependent artifacts upon rollback.

Thus, similar to links in HTML, interprocess dependencies are unidirectional and need not affect the entity they depend on, since a link typically consists only of the linked object’s address without modifying that object. Although on the WWW this bears the danger of having links pointing to nowhere, when objects pointed to are removed by persons who cannot know what links have been established on their objects, this is much less a problem with processes. Interprocess dependencies can be established on unmodifiable parts of other processes only, and these are intended to remain at stable addresses on reliable servers once they have been released in public, for a duration of at least several years (and with an archival system possibly providing longer access), possibly up to a predetermined expiry date. And even if some links may become inoperable, this is a small price to pay to achieve the essential simplicity of the scheme. For WWW protocols, Shirky contends that it is their
simplicity including their easy interoperability with existing systems and the purposeful ignorance of more sophisticated solutions that have contributed significantly to their current popularity [Shi98b], and we believe this should apply to processes too.

An interprocess dependency may be established from any (atomic or compound) node in one process to any read-only node in another process (as a node dependency), or it may be established from a node in a process to a configuration in another process (as a configuration dependency), which is not possible with intraprocess dependencies. As detailed below, a configuration allows a specific release of a software product to be described, typically consisting of several components, e.g. an application or an operating system. Thus, a single interprocess dependency suffices to describe a dependency on several (atomic or compound) nodes in other (even several) processes, if they are all contained in one configuration. Technically, an interprocess dependency is established on an entity in another process by checking the existence and accessibility of that (read-only) entity and by registering its address (consisting e.g. of host, process, and node or configuration identification) locally without affecting that entity. Alternatively, in order to provide interoperability (on a low level) with legacy systems and with processes defined in other modeling languages, an interprocess dependency may be registered in a textual mode. Unlike in the real mode, this is for informational purposes only for developers and users and cannot be used e.g. for automatic delivery and invocation of software or for validation of configurations (as described below); it may denote e.g. a conventional file holding a software product release.

2.6.2 Metaprocess Dependencies

Metaprocess dependencies are a special type of interprocess dependencies that define dependencies among components across different metaprocess levels. Again, node dependencies and configuration dependencies are possible. They do not describe the dependency of an artifact on another artifact or configuration, but instead they describe the dependency of an artifact on its tool, by providing a link on a node or configuration in the tool’s development process. During process definition, for every atomic node, the creation function and the required predicate may be specified textually (including tool commands and options), and additionally, the implementation of each of these functions may be specified by a metaprocess dependency in order to precisely define origin and version of the tool (again, as illustrated earlier in figs. 2.3 and 2.5a). In this way, the tool implementing the function is uniquely identified, e.g. as a configuration in the process that implements the tool, thus upon process enactment the correct tool release (version) can be invoked by the deployment and runtime system. Thus, metaprocess dependencies belong to the process and contribute to the precise specification of tools in a
software engineering environment used during a process, and thereby contribute to the precise process-dependent configuration of the environment itself.

Not only the individual tools used in a process and available in the environment, but also the framework, the environment's kernel, may be specified by a metaprocess dependency. Therefore, every process additionally has a metaprocess dependency on the environment's framework's process, i.e. on a configuration in the framework's development process specifying the precise release of the framework to be used. Thus, the implementation of the process-independent part of the environment is also specified and so the correct environment release can be invoked by the deployment and runtime system, e.g. when an existing process is opened. Specification of the framework implicitly also specifies the process modeling language used, since it is implemented in the framework. Thus, the language is reflexive, i.e. via specification of the framework it can even specify (an older revision of) the language it is implemented in itself: process X creating framework (and language) X has a metaprocess dependency on process Y of framework (and language) Y that is used in support of process X. However, as with other interprocess dependencies, for bootstrapping purposes and in order to be able to use legacy or other external tools, a metaprocess dependency may alternatively be defined textually only, referring e.g. to conventional files or to processes defined in other modeling languages.

Fig. 2.11 illustrates a possible scenario of a small process whose two tools (editor and compiler) as well as the process framework itself are specified via metaprocess dependencies to their respective development processes (the other four processes are shown only as nodes without detailing further nodes they may contain). Artifact "T.Mod" has a metaprocess dependency on the process of the editor used to create the artifact, and artifact "T.Obj" has a metaprocess dependency on the process of the compiler used to create the artifact. Additionally, process "Tool" holding the nodes of the two artifacts has a metaprocess dependency on the process of the process support framework, thus the complete environment used by this "Tool" process consists of framework (which also implements the language used to define and enact the "Tool" process), editor, and compiler. Additionally, editor and compiler have normal interprocess dependencies (links) on the framework's process indicating that they depend on the framework (since they use the framework's tool integration services that allow them to be plugged into the framework to form an environment), and all four processes have a normal interprocess dependency (link) each on the "Oberon" process indicating that they have been implemented in an Oberon system. Metaprocess dependencies are only shown for the "Tool" process; similar ones may or may not be defined in the other processes too.
Some metaprocess dependency issues are still a matter of research, e.g. different layers in the process history could have individual metaprocess dependencies to reflect different releases of the process framework (and language) used during the (long-term) course of process evolution, thus different releases of the framework might need to be invoked according to the process layer being browsed or enacted. Exchange of tools (e.g. update to a new release) could be handled as a normal process evolution operation, with a new node created, or alternatively the concerned attributes of the node (holding the tool metaprocess dependencies) would manage their own history list.

2.7 Configuration Management

2.7.1 Configurations

A configuration is a consistent set of confirmed (therefore read-only) process nodes in a process or in linked processes, i.e. that fulfills consistency rules described below, and that may be used to completely describe a specific product release, e.g. all source code of a release, of use to other developers, or all object code, of a release, possibly with manuals, for deployment by users. Compound nodes in a configuration include all their contained nodes in the configuration, and atomic nodes include their associated artifacts. Thus, a configuration describes not only all included artifacts that make up the software product, but also their dependencies in the process structure. While configurations generally hold only confirmed and thus read-only artifacts, for testing purposes it may be useful to permit modification of tested software in configurations that
are not widely released, but this issue requires further research to define the release restrictions of such configurations. Also, as implemented, configurations are currently limited to include artifacts of their own process only, although every process may hold numerous configurations. Possibilities to include artifacts of several (i.e., linked) processes in a single configuration are still a matter of research, including the question of how to limit the size of such a configuration, since potentially many processes could be linked together.

A configuration is specified by first selecting a set of nodes in a process (the configuration's leaf nodes), which is a horizontal projection on the process that defines which revisions of artifacts are used (i.e., in which history layers). Potentially, all predecessor nodes of the selected ones are included in the configuration. Additionally, a vertical filter projection is specified on the process to select only desired artifacts out of the potential set of artifacts, e.g. for release the filter may specify to include object code and documentation artifacts, but no source code. If appropriate, the chosen variants are also specified. Whereas configurations may be defined with different revisions and variants of nodes, revisions (evolution) and variants of configurations themselves are not yet supported.

E.g. in fig. 2.12, if node 6 (or both nodes 6 and 7) is/are selected as leaf node(s) of a configuration, then node 5 is potentially also included in the configuration. If node 4 is selected as leaf node, then nodes 1, 2, and 3 are potentially also included. The filter then specifies the actual nodes to be included in the configuration among the potential ones.

![Diagram of a small process with history to illustrate possible definition and validation of configurations](image)

**Fig. 2.12: Small process with history to illustrate possible definition and validation of configurations**

Where intermediate nodes are filtered out of a configuration, they cannot be completely removed. E.g. if node D depends on B and B depends on A, but only D and A are desired in the configuration, then node B with its dependencies is still required in the configuration in order to preserve the process structure in the configuration (fig. 2.12), but B's artifact and most
of B's other metadata do not form part of the configuration, and B will typically not be visualized when the configuration is displayed.

2.7.2 Configuration Validation

A configuration is successfully established if consistency rules are fulfilled. They specify that at most one revision and one variant of an artifact contribute to a configuration. Additionally, for variants, only one alternative per set of alternatives is permitted per configuration, although a configuration and even a node may hold alternatives from different sets. E.g., if alternative set “operating systems” in a process contains the two alternatives “SPARC” and “Mac,” then a configuration may not contain some nodes from alternative “SPARC” and others from alternative “Mac.” All nodes must belong to either the same alternative per set or to none.

For revisions, if three artifacts A, B and C are in a configuration, and B and C each depend on A, then appropriate revisions of B and C must be chosen that depend on the same revision of A, since the configuration may contain only one revision of A (e.g. in fig. 2.12, either nodes 1, 2, and 3, or nodes 5, 6, and 7, but not combined). Vice versa, a choice of B and C that each depend on A determines whether they may be in the same configuration, i.e. if they depend on the same revision of A. Similarly, if an artifact D depends on both B and C, B and C must be chosen that are predecessors of the same D, respectively a chosen D determines the required revisions of predecessors B and C (e.g. in fig. 2.12, either nodes 2, 3, and 4, or nodes 6, 7, and 8, but not combined).

Thus, the validity of configurations (and configuration sets, as described below) can be easily and automatically verified, which is an important precondition both to successful software development and deployment. If any of the consistency rules described above are not fulfilled, the configuration cannot be defined and there is a dependency conflict, but the desired configuration may be split up at the conflicting point to define two configurations, and thus resolution of the conflict may be delayed until installation time. Upon deployment, as will be detailed below, installation includes copying of configurations as well as establishment of dependencies and links among different installed configurations to reflect the original dependencies. Thus, if both of these configurations are to be installed on the same host, the dependency conflict can be recognized and handled in different ways: either only one of the configurations may be installed on the host at any one time, or both configurations are installed and resolution of the conflict is delayed until runtime, thus requiring a mechanism that prevents artifacts from both installed configurations from being invoked (executed) at the same time.

Even if there is no dependency conflict, there may still be a non-dependency conflict that manifests itself at runtime. Two artifacts that do
not depend on each other and that do not even need to directly or indirectly jointly depend on other artifacts (although they will typically at least both depend on the same operating system) may still influence each other at runtime in an incompatible way, e.g. by both using the same non-sharable resource (hardware or other software). Such non-dependency conflicts cannot currently be modeled by the process modeling language, although they also represent an important category of conflicts that warrant further research to allow them to be registered in processes too. Incompatibilities could be modeled by a type of "exclusive-or" dependency, and it should be possible to design a suitable notation. However, whereas modeling seems feasible, detection of non-dependency conflicts (i.e., their recognition in practice) is deemed difficult and cannot be automated, and worse, the number of potential combinations of independent artifacts that could be involved is practically unlimited. In analogy to the testing dilemma where a test can merely depict the existence of a fault but not its absence, it is difficult to a priori restrict the set of artifacts potentially involved in a non-dependency conflict, unlike for dependency conflicts whose involved artifacts must at least be reachable via dependencies.

Configurations are also useful when deployed software is to be removed (de-installed or deleted) from a host, since dependencies, both within and among configurations, show which artifacts may be deleted and which ones are still in use. E.g. if artifacts B and C both depend on artifact A, and B is to be removed, then A may not also be deleted before C is deleted (again, fig. 2.12). Vice versa, if B has been removed earlier and C is to be removed, then A may also be deleted at the same time.

2.7.3 Configuration Sets

A configuration set is a set of configurations, but unlike configurations that are stored with their processes, configuration sets are stored independently of specific processes. A configuration set may be used to describe a single product consisting of several configurations, and it merely holds the information about which configurations it contains. Although not yet implemented, the existence of configuration sets in addition to configurations is justified, as a configuration set may describe a product that consists of several otherwise unrelated configurations. E.g., a product could consist of the three components A, B and C, each defined in separate processes, with mutually unrelated components B and C (e.g. tool extensions) that both depend on A, i.e. are separately linked to A (e.g. a tool kernel), but with A's process not needing to know about B or C, as in fig. 2.13. This information cannot be included in any of the three processes without introducing an undesired link (e.g. B and C are not and should not be linked, and A cannot and should not be linked forwards to B or C), i.e. it cannot be specified in a conventional configuration. Therefore A, B and C are each described by a configuration in its
respective process, and the three configurations are included in a configuration set that describes the whole product but is stored separately from any process.

Fig. 2.13: Three linked processes that could be included in an independent configuration set

2.8 WORKSPACES AND ACCESS PRIVILEGES

Although a detailed study of workspace concepts and of access privileges is beyond the scope of this thesis [Est96], we envisage that different “circles” of access privileges will be used, per node and/or per developer, who may or may not assume different roles [HO94]. In the innermost circle, a developer will have full read and write access to his/her own artifacts under development at any time, but preferably in a private protected workspace, i.e. a process area where access is temporarily limited to the responsible developer. The day-to-day operations (editing, testing, etc.) are carried out within this workspace, and the results are transferred to the process respectively made visible to other developers only in coarse-grain steps [HKO92], when the artifact’s state changes (e.g. from editable to confirmed), and possibly also at certain selected times during editing – thus, other developers will not always see the very newest version during editing, but only after completion or when the responsible developer decides to share intermediate results. Whether more than one developer should be within the same innermost circle, and how the minute day-to-day issues will be communicated within a team of developers, i.e. whether and how fine-grain issues should be recorded, are among the open questions. However, the process and its history should probably only record results in coarse-grain steps, e.g. at state changes, i.e. creating a new version should thus be avoided for every minor modification. In the next circle, all developers of a team will have read access to all artifacts in a process, but only selective write access. Further access circles could include all developers of a company, external developers, and finally users, all of which only have read access, and only to an increasingly restricted set of artifacts; e.g. users may only access released configurations of executable software and its documentation, but no development-specific artifacts. Generally, the less involved a process customer (developer or user) is with a process, the less frequently he/she should be made aware of modifications or new versions.
Chapter 3
Framework Architecture Design

3.1 Objectives and Requirements

3.1.1 Integration Types

Integration is considered an essential concept in software engineering environments and the integration provided by an environment is a key element in providing effective support. However, integration has also become a popular buzzword, and the term can have many meanings. Several proposals have been presented for classifying the types of integration an environment may provide, although there is no general consensus specifying exactly what types exist, and there is not always a clear distinction among integration types. Different integration types are typically supported by specific services in an environment’s framework, although again some overlap is possible, i.e. strict one-to-one mappings of integration types to services are not always possible. Nevertheless, considering information from various sources [BM91][BM92][ECMA93a][Som96a][TN92][CN92][Was90][Am94][MS92], we have identified the following main types of integration in environments and their respective services:

- **data integration** (also known as tool integration or semantic integration): the ability to share information; supported by data repository services (storage and management of data and relationships) and data integration services (management of groups or configurations of data items);

- **process integration** (sometimes misleadingly known as technical integration): the ability to share a common understanding of the software process; supported by process services (or task management services); sometimes, control integration, the ability for communication among tools, supported by message services, and framework integration, the provision of common mechanisms used by tools, are also distinguished;

- **organizational integration**: includes team integration, the support of cooperative work in a team, and, associated with it, management...
integration, the provision of management information to be derived from and used in the process;

- presentation integration (or interface integration): the provision of a common user interface by tools and environment; supported by user interface services.

The subprojects DIPS and CHIPS of the GIPSY project focus mainly on the first three main types of integration, as detailed in the following sections.

3.1.1.1 Data Integration

A classification of data integration in levels proposes five different ones [BM91][BM92][TN92], with every level containing the properties of the previous one and augmenting them by new ones:

- carrier level: sharing of streams of bytes (files), e.g. common I/O via UNIX pipes;
- lexical level: sharing of lexical conventions (symbols), e.g. by using a common scanner;
- syntactic level: sharing of syntactical conventions (rules governing formation of data structures), e.g. by using a common parser;
- semantic level: sharing of semantics of structures (structure definitions and meanings of operations on them), e.g. by using metadata, or by using executable definitions as in extensible attribute grammars, e.g. to provide a common understanding of an individual process step’s artifact’s semantics to several tools;
- method level: sharing of roles (purpose) and context in the process, e.g. by providing a common understanding of the whole process, not only of individual artifacts, e.g. to enable tools in a process to access artifacts of neighboring process steps (such as a compiler accessing several symbol file and source code artifacts).

Data integration provides fine-grain integration of the software artifacts under development in the process, at least at the lower four levels, to ensure a common understanding of the artifacts among different tools involved in a process. Many tools still only provide carrier level integration, with higher levels of integration only provided by more specialized tools for restricted types of artifacts. Our CHIPS subproject provides semantic level data integration by allowing tool components to be specified using extensible attribute grammars (EAGs) and the specified tools then to be generated by a compiler-compiler. Thus, such tool components allow tools to be built that share a common understanding of the structure of the artifacts they operate on and of the operations used to operate on the artifacts, but this is limited to formal text-based artifacts.
As CHIPS-generated tools can be plugged into the DIPS framework to make up an environment, we call this the *artifact layer* of integration in the DIPS framework.

### 3.1.1.2 Process Integration

Process integration provides coarse-grain integration of software artifacts under development in a process, and it overlaps with data integration at the method level. It provides a common understanding of the process for all tools and persons involved in a process, by allowing the process to be defined and enacted, and thereby allowing tools to be controlled and constrained. It manages metadata about the artifacts such as dependencies, development contexts, versions, configurations, developers, date/timing information, and others. This is provided by process services and is the main task of the DIPS framework. Together with organizational integration described next, it provides the *organization layer* of integration in the DIPS framework.

### 3.1.1.3 Organizational Integration

Organizational integration, overlapping with process integration, provides a high level of process integration that concerns additional process information and metainformation used primarily for project management purposes. It involves controlling and monitoring the process, possibly in association with a superordinate business process (managing productivity and quality metrics, resources, schedules, manpower, budgets, etc.). Ideally, instead of being managed in a separate process, business information would be stored in data structures in common with the development process, being accessible in different views of the common development process detailing selected issues, e.g. date/timing issues, thus assuring integration. It may involve association of the organization structure of the team of developers with the development process. It may also involve organizational issues concerned with integrating several different processes in a coherent way, e.g. in a virtual organization. The process modeling language already provides the feature of process linking for this purpose, but obviously organizational issues will also need to be studied beyond technical ones, e.g. how to integrate different development cultures, but this is beyond the scope of the current project. Thus, issues such as these are so far only partially covered by the DIPS project, but some of them have been taken up separately [MS97] and are considered important extensions for future consideration.

### 3.1.2 Process Modeling Language Requirements

In order to design the architecture for a DIPS process support framework for software engineering environments based on the DIPS process modeling language, requirements may be derived from the DIPS process modeling language that should be met by such a framework. The goals of
the DIPS language may be translated to requirements for specific DIPS integration services. Since DIPS aims at integrating process and repository issues, process and organizational issues, as well as development and deployment issues, there is no strict distinction among different services conventionally providing separate support for these issues.

In the DIPS system, process services should provide storage and management of processes and artifacts with version and configuration management to support process and (coarse-grain) data integration, conventionally provided by data repository and data integration services, they should support process definition, enactment, browsing and querying (although specification of a query language e.g., “SQL for processes,” is beyond the scope of this thesis), conventionally provided by process and control integration services, also contributing to process integration, and they should provide process monitoring to support organizational integration, sometimes provided by management integration services, if at all. Furthermore, process services should include deployment issues by supporting configuration, release and delivery of processes respectively their contained software, which are not normally provided by conventional framework services. Tool integration services should allow tools to be used in processes that support high-level (fine-grain) data integration, including CHIPS-generated tools and existing tools. Messaging services should support communication among heterogeneous distributed artifacts to support independence of artifacts and tools from platforms, from the framework, and from each other. A limited form of presentation services should facilitate building user interfaces for process views, but presentation services for individual tools are not discussed. Further issues that need to be supported are security and management of access privileges for developers and for users, although they are not studied here and are beyond the scope of this thesis.

Furthermore, general requirements that should be met by the DIPS framework include support for heterogeneous artifacts, tools, and platforms (beyond messaging services), and support for wide-scale distribution, for individual processes and linked processes, and for development and deployment, while providing some independence between development and deployment activities. Heterogeneous systems and wide-scale distribution require issues such as compatibility (with existing systems and tools), simplicity, reliability, availability (accessibility of distributed data), and efficiency to be observed, and appropriate distribution policies to be used, if the framework is to be widely adopted.

Of particular interest are distribution policies that support operating efficiency by minimizing resource usage (memory, disk space, network
bandwidth, time, etc.), whereby time is of greatest concern to developers and users: while no hard 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 latency is already noticeable under typical network bandwidth assumptions), and few involve communication over a wide area network (whose latency is tolerable for infrequent operations). Thus, while the language specifies features to be provided, it does not specify how they will be used, i.e. what individual operation types are required and how the language features should be mapped to a distributed architecture.

3.2 ARCHITECTURE LAYERS

3.2.1 Overview

The framework architecture’s design is first presented in an overview from the point of view of abstraction layers, where different aspects of process support and integration services are presented that either concern only development, or only deployment, or that concern both. In subsequent sections, the focus is on development support and on deployment support in greater detail, in particular on the employed distribution policies.

![Fig. 3.1: Four-layered DIPS framework/environment architecture](image)

In order to provide independence among development and deployment, yet allow processes to be shared, we propose a framework implementing the language based on a layered architecture, with vertical dependencies among layers [Par74]. As illustrated in fig. 3.1, a four-layered architecture is proposed: basic and distribution services are used, upon which common distributed process services are based. These define process services shared among development and deployment and provide process interoperability and process awareness, while development and
deployment systems each extend them with specific features, and both use the process data structures and operations defined in the process services layer. Tool integration services allow development tools to be plugged into processes via the common process services. Process-specific tools, respectively processes, form the uppermost layer of a complete environment's architecture, though they are not part of the process-invariant framework.

3.2.2 Basic and Distribution Services

On the lowest architectural layer, a basic set of services including distribution services is required. The distribution services handle distribution and replication of processes and artifacts and include a messaging system allowing distributed entities to communicate. This provides a level of abstraction from location and from heterogeneous systems, using the same interfaces and protocols on different platforms. The distribution services also provide transactions with a limited form of ACID protection (atomicity, consistency, isolation, durability) [GR93], handle synchronization of replicas, and manage access privileges and security. Such services are typically provided by object request brokers (ORBs) for heterogeneous distributed objects extended by specific services, e.g. CORBA ORBs with additional CORBA Services [OMG98][OHE96] for replication and transactions, although scalability for Internet-wide distribution demands limits on messaging, transaction and replication protocols [ABN95]. Although the subsequent presentation of our implementation in the next chapter will reveal that we did not use CORBA itself but implemented DIPS in Oberon based on our own ORB, inspired by CORBA, at this point the design should still leave open a possible choice of specific platforms.

3.2.3 Common Distributed Process Services

Based on the distribution services, the common distributed process services form the intermediate architectural layer that handles those aspects of the process modeling language that are common to both development and deployment. They provide the basic data structures for processes with their associated operations, but without defining rules and policies. Thus, they define atomic and compound process nodes and standard and interprocess dependencies as basic process composition elements, together with creation, deletion, move and copy operations, as well as some basic browsing operations and process user interface components that are used both by the development system and the deployment system. Only structural dependency information is handled, not the optional process-specific dependency semantics that are defined by tools. Association of artifacts and various attributes (access privileges, date/time, etc.) with process nodes is handled. Specification of configurations and configuration sets is supported. Serialization of multi-
dimensional process structures for persistent storage and transmission via network is provided. Processes are managed in on-site process bases (process management systems) at developer and user sites (similar to databases, but holding (complete or partial) local replicas of processes), with different access privileges for developers and users, and distribution and replication of nodes and artifacts of processes is supported, but without defining policies. Based on the distributed messaging system, simple messages and messaging protocols are defined and supported that allow distributed developers and users to access and copy processes to carry out development or deployment activities, and message forwarding among artifacts or tools is also handled. A global addressing scheme is defined that allows precise versioned identification of every artifact and configuration, also distinguishing among originals and replicas where appropriate.

Common process services are the key to being able to use a common process representation beyond individual processes and beyond development time, providing a base for process interoperability and process awareness among developers as well as between development and deployment.

3.2.4 Development System

Based on the common distributed process services, the development system implements all the rules, operations and user interfaces for process definition, enactment, monitoring and browsing, allowing processes to be defined, linked, evolved and enacted. It thus extends the basic operations and defines rules and policies specifying how the basic elements and operations provided by the common process services may be used to assemble valid processes, how processes evolve when iteration or evolution operations are performed, and it enforces constraints on enactment concerning states of process nodes. Specification of configurations and configuration sets and release of products through specification of appropriate access privileges is supported, which is restricted to confirmed (read-only) process nodes respectively artifacts. Although partially concerning both development and deployment, these activities are carried out by developers and are therefore supported by the development system.

The development system comprises a distributed process engine that provides 2D and 3D graphical process views for definition and enactment operations as well as for navigation and query operations on the process structure and on artifacts. Based on the simple messaging protocols it implements messaging protocols to support distributed operation according to specific policies. As detailed below, it defines specific distribution policies for process nodes and artifacts, specifying when and how elements are distributed and replicated, and manages active
replication (replication with subsequent updates) of evolving processes among developer-site process bases. Thus, it implements the policies that define when and how the basic distribution and replication services provided by the lowest layer are made use of. Atomicity of distributed operations is ensured by making use of transactions, and partial and full process replicas are synchronized. Distributed messaging protocols provide read-and-write or read-only access for developers to remote copies of their processes as well as to other processes, and allow them to link processes together.

3.2.5 Deployment and Runtime Systems

Based on the common distributed process services, the deployment system extends the basic operations and provides operations to access and browse distributed processes and to perform deployment activities. It handles copying via network of completed and released software products, specified as configurations and configuration sets, from developer-sites to user-sites, directly or via intermediate servers, thus providing passive replication (one-time replication without updates) of read-only process parts. It even allows a selected configuration spread over multiple distributed sources to be downloaded in one operation. It manages user-site process bases, where all retrieved configurations are inserted, and dependencies are established locally reflecting those in the original processes. This allows validation of configurations and configuration sets at user sites, prevention of invalid installations, and straightforward removal of undesired configurations. Deployment and configuration validation are only possible by basing the deployment system on the common process protocols and using the same process information the developers create and use. Users thus have their own unique base of configurations typically originating from many different processes from different developers. The graphical user interface for accessing processes is similar to the one in the development system, although providing more limited information (e.g., no proprietary information from developers), and retrieve accesses are read-only.

The runtime system loads executable software respectively instantiates objects by retrieving the required artifacts directly out of the local process copy (replica). Thus, the runtime system is slightly modified compared to a conventional one, since its loader retrieves object code artifacts directly from the local user-site process base, instead of loading files from the local filing system. As with conventional files, access is by name only, i.e. the programming language used for the artifact’s implementation does not require any versioning concept, but unlike files, where versions are at best specified by paths (e.g. in Oberon, UNIX, MS-DOS, Windows, or other systems), if at all, or managed by separate configuration management tools, installed valid configurations guarantee that correct versions of all required artifacts are accessed. Process bases
may also be shared among several user workstations, e.g. diskless network computers, or if desired, downloading of software may be delayed until it is invoked. Ultimately, all local software will be stored in the local process base, and the local filing system will only be used to store data files, or will possibly become obsolete if similar “information processes” are used for these purposes too. Association of data files with their applications conventionally occurs (rather arbitrarily) through filename extensions or file attributes (creator and type attributes on the Macintosh); double-clicking (opening) a file typically invokes a search for an appropriate application according to rules that often remain unclear to the user, in particular where several matching applications are available. Thus, introduction of process concepts for data files too has the potential of improving file management and reducing uncertainties.

3.2.6 Tool Integration Services

Tool integration services allow development tools (editors, compilers, etc.) to be plugged into the process framework in order to extend the framework to a full software engineering environment. High-level tool integration requires tools to operate via the process, and not directly on files, e.g. to access versioned artifacts, similar to the runtime system. The services thus define extensible messages for communication among artifacts, which are accessed via the process, whereby artifacts may be either conventional data files or documents, or persistent objects that are instances of model parts of tools (classes in the object-oriented sense), as seen earlier. An artifact containing source code text may be an instance of a tool model consisting of extensible tool components for editors and compilers such as text editor, scanner, parser, declaration analysis, type checking and code generation. For CHIPS-based tools, specific messages are used that access related artifacts in their process context, to validate formal dependencies or to create new artifacts, e.g. an object code artifact’s creation tool may be implemented as a compile message sent to its related source code artifact (an instance of an editor-compiler tool model), which compiles itself and returns the object code artifact. Message types also include requests to provide specific views of an artifact, e.g. to create HTML from text-based artifacts used for their WWW presentation, containing appropriate hyperlinks to related artifacts. Other tools, e.g. non-object-oriented ones, may be supported via adapters, but with less sophisticated integration, e.g. limited to tool invocation, thus allowing conventional files to be used as artifacts too.

Since typically only developers will use tool integration services and tools, only they will be able to open the heterogeneous editable artifacts requiring these tools. Users will thus often not be able to access such artifacts directly, but, access privileges permitting, they may be able to send them messages to browse or query the artifacts indirectly, e.g. request messages to present some artifact contents in HTML on the
WWW, when users perform browsing and deployment activities via WWW, or in another format. E.g. users may not be able to instantiate a compiler tool to create, modify or browse an object code artifact, but they may be able to access an existing object code artifact's content for downloading and instantiate it on their runtime system to execute it as software. All artifacts destined for users, such as documentation artifacts, or object code artifacts making up a software product, must be able to present themselves to users (as well as to servers such as release servers and archival servers) in a development-tool-independent way, either by including any necessary tool support in the artifact itself or by using a tool-independent format such as HTML or, for object code artifacts, object code that is directly downloadable, instantiable and executable by the runtime system used by both developers and users.

3.3 DISTRIBUTED DEVELOPMENT FRAMEWORK

3.3.1 Overview

The development framework consists of those parts of the framework which are required by developers actively participating in development processes. They include the distribution services and the common distributed process services, both also shared with the deployment framework detailed later, as well as the development system and the tool integration services. The tool integration services mainly support integration of CHIPS-based tools, thus only few of these services are required if no CHIPS-based tools are used. If tools and/or parts of the framework are already available as process-based software themselves, participation in development processes will also require the deployment framework, i.e. the deployment and runtime systems, in order to use this and other process-based software, which should ideally be increasingly the case after initial bootstrapping activities.

This section focuses on the development framework and describes the distribution policies used therein, which have been derived from the types and from typical usage patterns of development operations, and the next section will similarly focus on the deployment system.

3.3.2 Development Activities

3.3.2.1 User Operation Types

In order to be able to design a development framework that efficiently supports operations and minimizes resource usage, basic operation types need to be characterized and their usage patterns need to be established. Here, operations are not distinguished by mode or enactment state, but rather by their resource usage and access characteristics to distributed entities, i.e. local and remote read and write access to process structure and artifacts. Then, usage patterns showing the number of different basic operations carried out in a given time interval are established.
Typically, in distributed collaborative software engineering, multiple developers work on different parts of a process concurrently, e.g. on nodes within different compound nodes, although two cannot work on the same node or artifact simultaneously. Developers use the process framework’s graphical user interface to perform definition and enactment operations. An operation always originates on one node in the process, called “source” with respect to that operation, and may involve other nodes in the process, in the same compound process node as the source, called “near” logical distance, and/or any other nodes of the process or of other processes, called “far” logical distance. All accesses to an atomic node’s artifact are handled by the atomic node, involving some of the node’s attributes too (holding metainformation about the node or artifact, e.g. dependencies, states, tools, developer, date/time constraints, etc.). Therefore, no operation can involve only artifacts. A message from an artifact to another artifact is routed via their respective atomic nodes.

Operations performed by developers on an evolving software product, i.e. during development and release (belonging to deployment), include process definition and enactment, with edition and confirmation of artifacts, navigation, browsing and querying of processes and artifacts, and establishment of configurations and releases (access privileges on configurations). From these, five types of basic access operations may be identified (which will be abbreviated), as follows:

1. **Struc**: edition of process structure including node attributes through process definition and enactment, including process iteration and evolution: Operations which modify dependencies, states and other attributes of processes (except for operations of the next two types), which may involve multiple nodes at once.

2. **Cont**: edition of contents, i.e. artifacts, of atomic nodes during enactment: Operations which modify artifacts of atomic nodes; these operations only involve one artifact at a time but may effect multiple accesses to the artifact, and some also modify node attributes such as states.

3. **Eval**: execution of evaluation procedures on artifacts during enactment: Operations which evaluate formal conditions associated with atomic nodes (concerning the artifacts under development, which may effect read accesses to other artifacts). These may create new artifacts of atomic nodes, and may thereby also modify attributes such as states.

4. **Nav**: navigation (browsing) on process structure only: Operations which only perform read accesses and only on the process structure and attributes, and may involve multiple nodes.
5. **Qry**: queries and browsing on both process structure and artifacts: Operations which only perform read accesses and which may involve accesses to process structure, attributes, and artifacts, and to multiple nodes.

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

Since some operations involve only the process structure and attributes and others additionally involve artifacts, this suggests that an analysis of operations should distinguish between accesses to process structure (with attributes) and to artifacts. This distinction then may or may not be reflected in the architectural design.

### 3.3.2.2 Usage Patterns

After defining basic operation types, usage patterns are derived by analyzing a carefully estimated load of operations performed by developers on a process of a specific size. The results may then be used to design the distributed development architecture, in particular its distribution policies. Preliminary measurements performed with the prototype implementation have confirmed the orders of magnitude of the usage patterns, and the design may thus be improved in a bootstrapping procedure.

A typical nearly-completed process will contain up to ca. 500 nodes, with ca. 100 nodes in the original process definition made up of ca. 20 compound nodes each containing on average 4 further nodes, and after process iteration and evolution with every one of these ca. 100 nodes represented in the current layer plus on average in 4 history layers, thus making up ca. 500 nodes. The total number of history layers in the whole process will typically be much larger than 4, but not every node has an older version in every history layer. Although larger processes, with thousands of nodes, are possible, such processes will typically be defined as several smaller independent processes that are linked together. Also, although several processes may run in one environment, this should have little influence on the distribution policies within one process. Typically, between 1 and 12 developers will work actively on a process. The average size of one node (atomic or compound), including dependencies and other attributes, but without artifact or tools, is ca. 1KB, and the average size of an artifact is ca. 50KB (similar sizes for each in volatile or persistent memory).

On such a process, all developers will typically effect several thousand operations per 24 hours, whose observation allows specific usage patterns to be obtained. Since the quantitative usage patterns are based on
estimates, only qualitative results should be deduced. Furthermore, the focus of the usage pattern analysis is not on absolute measures but rather on the comparison of relative numbers of accesses to distributed entities, distinguished by read and write accesses, by access to process structure and artifacts, and by local, near and far logical distances. Only the results are presented here, but more details may be found in [SMW97]. Five qualitative results have been obtained from the usage pattern analysis (whereby the second and third results have been derived together with the first one):

1. Read accesses to the process structure are much more frequent than any other read or write accesses.

2. Read accesses to the process structure are much more frequent than to artifacts.

3. For the process structure and for artifacts, reads are much more frequent than writes.

4. Write accesses to artifacts only occur where the effecting operation originates.

5. For the process structure and for artifacts, for reads and for writes, access frequencies to the process structure and to artifacts are greater for “near” logical distance than for “far” logical distance.

3.3.3 Design of Architecture

Support for processes distributed among several workstations or sites, with different developers participating in a process at their individual distributed workstations, as well as support for linking processes across different sites, demands a distributed architecture for the process framework. The architecture is designed by considering the requirements as well as 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 with different distribution policies, i.e. distinguished by storage location of process structure and artifacts, as this is of primary concern when designing a distributed system [Mul93].

3.3.3.1 Architecture Types

First of all, focusing on the technology of the repository that constitutes the heart of a framework [GJM91], object-based and non-object-based distributed architecture types may be distinguished. A non-object-based solution such as a distributed filing system would signify that code and data are stored separately, which 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 entities
(structure and artifacts) to be stored in the form of objects, which is preferable considering the heterogeneous types of data involved, since it better supports data integration on a high level [BD94][ESW92].

An object database (ODBMS) [Cat94][Cat97] is an object-based solution. An ODBMS provides a high abstraction level for objects, including access to objects through protected actions having ACID properties (atomicity, consistency, isolation, durability) [BHG87][GR93]. However, the protocols ensuring ACID properties demand considerable resource usage (particularly communication bandwidth) that prevent scalability of the architecture to global dimensions. Assuming an ODBMS client/server architecture using a central storage server, access to objects by clients typically involves (costly) transferal of the objects to the clients to execute them locally, which requires a homogeneous environment. Support for heterogeneous platforms, tools and objects is therefore 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 to use middleware services such as an object bus or 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 [OHE96][Ber96][ORV96]. All objects are accessed via messages on the machine they reside, i.e. they remain on the host where they are executable. 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. Transferal of messages instead of whole objects requires less communication bandwidth, thus providing a greater possibility of scaling to large dimensions. While the abstraction level of objects is lower than in ODBMSs and access to objects through protected actions has 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. Thus, an ORB architecture type is deemed suitable and is therefore chosen for the process framework.

3.3.3.2 Architecture Variants

The storage location of process entities represents the most distinctive feature of the distributed architecture, therefore it constitutes the prime feature for distinguishing and comparing architecture variants. In order to be able to perform meaningful comparisons, three extreme architectures are initially discussed: 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 (table 3.1 below), and a hybrid architecture is then derived from this which attempts to combine properties of different variants in an optimal manner, and which will then be used as a basis for the actual implementation. The main property that is compared in the three variants is efficiency of operations, distinguished by read and write accesses, accesses to process structure and artifacts, and accesses over source, near and far logical distances. Further properties for comparison include support for heterogeneous objects, required effort to provide object consistency, redundancy and availability, and finally, simplicity of design.

Client/Server Architecture: The simplicity requirement suggests that all process entities should reside on one central storage server (as objects), which 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 objects are accessed on the single ORB server via messages, which may require considerable communication bandwidth (particularly for messages involving large contents such as artifacts), leading to low efficiency and possibly network congestion. Heterogeneous objects are unlikely to be supported well since they all have to be executable 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 source is involved, and a backup server could take over as a whole whenever the central server fails. Since this represents the simplest architecture, it was the one implemented in the initial distributed framework prototype.

Distributed Architecture: The realization that every participant in a software development process often works on a different part of the process suggests that process entities should be distributed in such a way that every participant's part (which could involve both structural and content information) resides on the developer's own workstation (respectively the nearest machine that always remains on-line, if personal workstations are shut down while not in use). The process entities 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 backups need to be performed at every host. Even though failure of a host would only concern a small part of the process entities, the loss of some structural process information could result in large parts of the process hierarchy to become inaccessible (similar to losing some directory information in a hierarchical filing system). Object consistency is easily achieved except when objects have to be moved due to a failure. In order to assess read and write efficiency, a fair assumption (derived from result (5) in the section on usage patterns above) is that most processes will be defined in such a way that the logical
distance (source/near/far) of nodes correlates with the geographical
distance of hosts where processes are stored (respectively their network
latency). This leads to an increasing access efficiency with decreasing
logical distance, as accesses occur via messages forwarded by ORBs.

**Replicated Architecture:** Result (3) stating that in general reads are
much more frequent than writes suggests that all process entities should be
replicated on all hosts, so that every host has a complete set of all process
entities, 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 be executable there, heterogeneous objects and platforms are practically
impossible to support, which is a significant drawback of this architectural
variant. The ORBs 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, e.g. for whole
artifacts. Write efficiency values here include the cost of keeping
replicated objects consistent, and sophisticated protocols are required to
ensure a tolerable level of object consistency.

### 3.3.3.3 Deriving the Optimal Architecture

After describing the three extreme architectural variants, an optimal
design for the actual distributed architecture is derived from a
combination of the variants. The assessment criteria used to define the
relative importance of individual properties emphasize the significance of
efficient operations, where the design should consider the usage patterns,
as well as support for heterogeneous objects. Of comparatively lesser
concern are simplicity of design, including efforts required to achieve
high availability and reliability (including redundancy and consistency of
replicated objects).

**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 (1)) and that these operations are supported optimally by the
replicated architecture, that one is chosen for the process structure. This
obviously requires the values from the replicated architecture to be
selected likewise for the structural write operations, even though here the
distributed architecture seems more favorable, but result (3) stating that
reads are much more frequent than writes confirms the better choice. To
store the content information (artifacts), 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 (4) stating that artifacts are
(logically) only written locally, and result (2) implying that even reads
(let alone writes) of content information are not very frequent, thus not justifying the added overhead of replication.

This leads to a design where the process structure is replicated on all hosts, and the content information (artifacts) is distributed on all hosts without replication, i.e. every host carries a part of all content information. This involves separating storage of process structure information (including attributes) and artifacts of an atomic process node, since the structural information is replicated and the much larger artifacts are not. Therefore, the artifacts exist as individual (heterogeneous) objects, separate from their associated atomic process nodes, which may act as proxy objects for the artifacts. Artifacts are accessed via messages forwarded by ORBs on the host they are stored persistently and where they are executable, and they merely need to satisfy the process message protocols in order to be able to participate in processes.

The only two features where the client/server design is optimal can only be partially considered in the hybrid design (signified by lighter cell shading in table 3.1), 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 objects are coordinated by and broadcast from one so-called master host, leading to simpler protocols to ensure consistency, although the simplicity of the other client/server or distributed designs is not achieved. Furthermore, considering that the total size of the structural process information is much smaller than the total artifact size, simpler protocols suffice for the replication of only the structural information than for the fully replicated design.

The crucial and frequently accessed structural process information (result (1)) is replicated on all hosts providing excellent read efficiency, availability and redundancy for this information. This also allows another host to take over as master relatively easily if the previous master fails. The less essential and much larger content information (artifacts) is not replicated, but following the idea of central data location in the client/server design, the content data is backed up from every host to 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 on some hosts, particularly those who have attained a non-modifiable state, 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 objects 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.
Also, the evaluation principles would be similar to the static case, therefore they are not investigated further here.

### 3.3.3.4 Hybrid Architecture Overview

Table 3.1 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.

The development framework provides management of the developer-site process bases (similar to databases, but holding (complete or partial) local replicas of processes) at every developer’s site (host). Every process base may manage replicas of many processes, both where the developer is actively (read and write) or passively (read-only) involved, and in the process base, links among process replicas are established that reflect the links among the originals. Additionally, hosts, developers, users, and access privileges must be managed per process, although these issues are not studied in-depth here and are beyond the scope of this thesis.

The hybrid architecture thus provides replication of the process structure on the hosts of all actively participating developers in a process, since developers often require information about different parts of a process (e.g. related artifacts and dependencies), which justifies their replication overhead, as updates of the process structure are less frequent than updates of artifacts. The architecture provides non-replicated distribution of artifacts that corresponds to the distribution of developers, i.e. every artifact is stored on its developer’s host, since artifacts are writable by only one developer at a time and updates are relatively frequent and should thus occur locally without replication. Moreover, artifacts can be based on very heterogeneous tools which would make replication on heterogeneous hosts difficult.

Process node replication is synchronized by a specified master host per process using a simple primary-backup scheme [Mul93]. The master also ensures durability of all process entities (nodes, dependencies, and artifacts) through operation logging and backups. Since the number of developers working on the same process is typically small, e.g. 1 - 12 participants, and processes typically contain less than 500 nodes, the number and sizes of messages are small enough to allow small-scale replication of the process structure, taking into account that there are no strict real-time requirements for response times. Larger projects would typically be run as several separate but linked processes, as most messages are not propagated along links to other processes. With reduced response time requirements, some replication is feasible even for Internet-wide distribution of developers, e.g. in a virtual organization. More flexible
distribution and replication schemes, e.g. with partial or dynamically changing replication, are conceivable, but more complex.

<table>
<thead>
<tr>
<th>Features</th>
<th>Architectures</th>
<th>Client/Server</th>
<th>Distrib</th>
<th>Replic</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Location</td>
<td>Central</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Distrib</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Replic</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Read Efficiency</td>
<td>Struc</td>
<td>Src</td>
<td>0</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near</td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Far</td>
<td>0</td>
<td>0</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Cont</td>
<td>Src</td>
<td>-</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Far</td>
<td>-</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>Write Efficiency</td>
<td>Struc</td>
<td>Src</td>
<td>0</td>
<td>++</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near</td>
<td>0</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Far</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Cont</td>
<td>Src</td>
<td>-</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near</td>
<td>-</td>
<td>+</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Far</td>
<td>-</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>Heterogeneous Objects</td>
<td></td>
<td>--</td>
<td>++</td>
<td>--</td>
<td>++</td>
</tr>
<tr>
<td>Consistency</td>
<td>++</td>
<td>+</td>
<td>--</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Redundancy</td>
<td>--</td>
<td>--</td>
<td>++</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td>--</td>
<td>--</td>
<td>++</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Simplicity</td>
<td>++</td>
<td>+</td>
<td>--</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Key

| x     | ++     | +      | -      | --     | 0      |
| applicability | excellent | good | very poor |

Features:
- Data Location: central, distributed, or replicated;
- Efficiency: efficiency of read or write operations (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: facility to provide object consistency;
- Redundancy: degree and facility to support redundancy;
- Availability: degree and facility to support high data availability;
- Simplicity: overall simplicity of design.

Tab. 3.1: Distinctive features of architectures
Since all distributed operations are routed via the single master host in a sequential manner, there is no danger of deadlocks occurring. Also, as has been seen, most operations are performed locally only (e.g. most browsing, querying, and enactment operations), few operations involve several hosts (e.g. some browsing, querying, and evaluation operations), and even fewer ones concern all hosts of a process (process definition operations and those enactment operations that change a node’s state, at most three per node: planned to editable, editable to confirmed, possibly rollback). Therefore, even for a large process with 500 nodes and 12 participants, there will typically only be a few thousand distributed operations, each of which lasts at most a few seconds, thus considering that a process typically is defined and enacted in the course of several weeks, there should not be a bottleneck phenomenon at the master that would severely limit performance.

**Illustration:** Figure 3.2 illustrates the hybrid architecture by a small example of two processes, each consisting of one compound node only: Process A resides only on one host; process B (which has a current and a history layer) is replicated on two hosts, while the individual distributed objects (artifacts) of its atomic nodes each reside only on one host. (Backups of distributed objects on a master host are not shown.)

*Fig. 3.2: Distributed development system, illustrating replication of process structure (process nodes) and non-replicated distribution of associated artifacts (shown as icons)*
3.3.4 Messages and Protocols

Simple messaging protocols are required that forward definition and enactment operations to replicated entities and that provide synchronization among them, supported by transactions. Additionally, distributed messaging protocols are required that allow developers respectively hosts to be added to or removed from a process, that allow users to connect to a process for browsing and retrieval purposes with limited access privileges, and that allow links to be established on existing processes. Furthermore, messaging protocols and standardized extensible message definitions are required that allow tools to access their artifacts (since these should be contained in the process and not in a filing system), and that allow tools respectively artifacts to interact with others in their process context and in other parts of the process.

Thus, support for heterogeneous artifacts and tools on heterogeneous platforms, and support for high-level semantic integration of CHIPS-generated tools is provided mainly by appropriate messaging protocols. This allows for independence of artifacts from platforms, thus enabling heterogeneous objects to be used, it allows for independence of artifacts and tools from the process framework, and it allows for independence of individual artifacts from each other. It requires a basic set of extensible messages to be defined and adherence by the artifacts, tools and process support framework (development and deployment systems) to the defined message protocols.

3.3.5 Addressing Scheme

An addressing scheme is used that allows identification of nodes or artifacts based on their persistent storage location, although at runtime the addresses may temporarily be extended by pointers to volatile object locations. In principle, persistent or volatile, every node must at all times be able to access its neighbor nodes in a process (linked by dependencies), every atomic node must be able to access its artifacts, and vice versa every artifact must be able to access its associated atomic node in the process, and thereby access its context in the process, i.e. artifacts are process aware both under development and after deployment. Typically, if not yet loaded, nodes or artifacts are loaded from persistent to volatile storage in order to be able to process access requests.

Every process node has a globally unique origin address, specifying the process’ master host’s Internet address, specifying by name the process on that host, and numerically specifying the node in that process, e.g. <"129.132.57.174", "Timings", "32014"> or <"mac-1942.ethz.ch", "GenDist", "32052">. This three-tuple is called the origin node address, and the master’s host address contained therein is called the origin master address. In order to allow unique identification of process entities at all times, during development and deployment, an origin node address and an
origin master address, once assigned, should never change (i.e., should always point to the same data), although if need be the master’s Internet address may point to different physical hosts during the course of time, provided that the processes are moved accordingly at the same time. For the Internet address, the IP address may be used directly if it does not change, whereas with dynamic IP addressing, a suitable hostname should be used instead that always references the same machine, since this address should always represent the same host respectively data. It is sufficient to store the origin master address and process name once per process, in its top compound node, and not in every node, since they are valid for the whole process and the top compound node of a process is easily accessible from any node within. However, if only a part of a process is copied elsewhere, e.g. upon deployment, it must be ensured that the origin master address and process name are contained respectively inserted in the (possibly different) top compound node of the process copy.

Nodes and artifacts are not distinguished in the origin node address, since all accesses to nodes and artifacts are by messages, and an atomic node acts as proxy object to its artifact, i.e. all artifacts are initially accessed via their atomic node. Thus, an atomic node responds to those messages it is able to handle (e.g. about metadata) and forwards the others to its artifact. The process structure (nodes and dependencies) is replicated on all participating hosts, whereas artifacts are not replicated. Thus, an access to an artifact is forwarded to the process structure replica on that host where the artifact is located, the atomic node thus holds that host’s Internet address, called the origin artifact address. Although an artifact may subsequently be accessed on that host directly instead of via the master host, for simplicity as well as for consistent identification purposes, the artifact is still identified by the original address that points to the master host. During development, until it is confirmed, an artifact may be moved to a different host when it is assigned to another developer, causing its origin artifact address to change, but after confirmation, it may not be moved any more. Some development addressing concepts are illustrated together with deployment addressing concepts in the corresponding section on deployment below, in fig. 3.3.

For development purposes only, the master host additionally manages in the process a list of Internet addresses of all hosts that currently participate in the process, i.e. that hold replicas of the process structure and that may hold some artifacts too, and a list of developer names with their access privileges. These lists are constantly updated as developers and/or hosts are added or removed to the process, and in a similar way as the process structure, they are propagated to the other hosts to facilitate transferal of the master to another host in case of the master’s failure. All process replicas also include an ordered list of backup master hosts to be probed by the other hosts in this case. However, since an origin master
address must never change in order to provide distinct identification at all times, this introduces an additional difficulty. The origin master address is retained for identification purposes, while the actually used master will be found by querying hosts of the backup master list; ideally, this should be a temporary situation only, thus if the master cannot be restored, its Internet address should then be assigned transparently to the temporary master host causing it to become the new permanent master, which should hold an identical copy of all processes the failed master held.

A configuration is addressed similarly to a node, except that instead of the node number, its address holds the configuration name, and once assigned, it must also not be modified any more. Thus, a configuration is addressed by the globally unique three-tuple specifying its process' master host's Internet address, specifying the process on that host, and specifying by name the configuration within that process. A configuration set, which may contain several configurations but which is independent from a process, is addressed by the host where it is stored (its master host) and the configuration set (addressed by name) on that host.

### 3.3.6 Release Activities

The release activities form part of deployment activities, yet they are performed by developers or similar persons, and they must therefore be supported by the development framework. Release comprises specifying a configuration that defines which process nodes have deliverable artifacts, e.g. a typical choice could be to release only object code and documentation but no proprietary source code, then setting the access privileges of this configuration (nodes with artifacts and selected attributes) to readable for users, and allowing users to connect to the process for retrieval.

As will be discussed in the next section, mainly for performance reasons most users will preferably retrieve their software from intermediate servers designed to cope with frequent download demands, rather than directly from the developer sites, although in principle software retrieval should be possible from any process replica, since replicas are identical at every site, except possibly for management of access privileges. Therefore, retrieval operations on the original processes are relatively infrequent and thus they are not taken into account in the development distribution policies of processes.

Released configurations may in principle not be modified any more and should be made available on reliable servers at stable addresses for a long duration; if they need to be physically moved, their released addresses should not be changed but should instead be mapped to the new location. In particular, developers should be obliged to keep (older) released processes available even when they later choose to (additionally) release
newer revisions (updates). Old revisions may still be in use for a long time and users need to be assured that all information concerning their software remains available, even if this conflicts with developers’ commercial interests. The appropriate release duration is difficult for developers to judge, although possibly an archival system may be devised for very old releases, e.g. which are more than ten years old. While old releases should not be deleted, they could be enhanced by newer information about problems and/or about the existence of newer releases (with appropriate links), but the choice about selecting a release should ultimately remain with the users.

For marketing purposes, we also envisage software component releases to be registered in searchable component catalogs on the WWW that carry information such as component purpose and pricing [SPP98], and that include links to processes in order to allow retrieval of selected configurations. Processes may also be translated to a graphical visualization on the WWW that is independent from the process framework, i.e. that may be read by potential customers with just a WWW browser, as we have demonstrated [SMW96]. Processes thus provide origin, dependency and configuration information about software components that allow users to make informed retrieval choices.

3.4 DISTRIBUTED DEPLOYMENT FRAMEWORK

3.4.1 Overview

The deployment framework consists of those parts of the framework that are required by users in order to retrieve and employ software developed in a process, i.e. process-based software. They include the distribution services and the common distributed process services, both also shared with the development framework, as well as the deployment and runtime systems. However, the other development-related parts of the framework, i.e. the development system and the tool integration services, are not required by users.

Deployment activities are typically carried out by users and comprise retrieval of selected configurations via network, installation (and removal) of configurations on user-site process bases together with validation of installed configurations, and execution of retrieved software.

3.4.2 Deployment System

3.4.2.1 User-site Process Bases

In order to perform software retrieval (respectively delivery, from the developer’s point of view), a user’s deployment system accesses a specific configuration of the desired software, selected perhaps by following a link from a WWW-based software catalog, on another server and copies it via network to the user’s local (user-site) process base. This large-scale replication of process parts among numerous widely distributed users is
relatively unproblematic and scales well since it exclusively involves read-only process parts (configurations), i.e. the replicas do not receive update messages, and the replicas do not need to be registered with the originals, unlike during development.

The copied configuration includes process nodes with dependencies and artifacts, not just the artifacts that hold the software, in order to retain the software’s dependency and other metainformation, i.e. retain the artifacts’ process awareness even when the software is deployed. To this end, appropriate dependencies and links are also established locally among the installed configurations in a user-site process base that reflect the dependencies and links in and among the original processes. This is possible since origin node addresses, including origin master addresses, never change and thus allow unique identification of processes, configurations, nodes and artifacts at all times. As discussed in the section on configurations in the previous chapter, this allows installed configurations to be validated and conflicting configurations to be recognized, and upon removal of configurations it similarly allows additional superfluous configurations to be removed respectively enables the removal of (shared) configurations that are still in use to be prevented.

Every user thus has an individual user-site process base on the user’s workstation’s hard disk (respectively on a server nearby in terms of network latency, e.g. in the case of diskless network computers), where (copied) process parts may be inserted and removed, similar to the developers’ developer-site process bases, except that in principle no modification of the process copies is permitted beyond their insertion and removal. A user-site process base typically holds process fragments (configurations) copied from many different sources that form a unique “puzzle” of process aware software installed on the user’s machine. Besides executable software, it may also hold associated documentation, and for developers perhaps even software not intended to be executed, but used only for reference purposes during development. In the case where installed software should be modified after installation, e.g. as in a file holding user preferences, such parts may either be installed separately from the process base in the conventional filing system, or an additional mechanism is required (provided e.g. by access privileges) that marks it as proprietary in the local process base after installation, to indicate that it may be modified locally but that it differs from the original and thus should not be used any more in the case that the local process base acts as source for a default installation elsewhere.

If the user is also a developer, the local process base may serve as both the developer-site and the user-site process base, holding at the same time processes where the developer is actively involved and replicas are
actively synchronized, holding processes where the developer is limited to read-only access, and holding processes (or process parts) that contain the framework, tools and other software the developer employs, as a user. Software under development may even be used by its own developer once a part is completed, simply by defining it in a configuration and releasing it to its own developer, e.g. for his/her own usage or for extended testing purposes, although a developer can also access his/her own software without formally releasing it, e.g. in the day-to-day edit-compile-test cycle. Furthermore, a process may still be under development and at the same time hold parts, e.g. in its history, that have already been released. This is a common situation with software that is released in different versions or upgrades, perhaps over the course of several years.

3.4.2.2 Distribution Policies

As seen before, every configuration (and configuration set) and every node respectively versioned artifact has a globally unique origin address that allows it to be precisely identified regardless of the actual location of the copy being inspected (including information such as origin, revision, variant, dependencies, and other metainformation). This allows a configuration (or configuration set) to be copied from any convenient source and not just from the developer. Thus, besides user-sites, many servers may manage process bases holding copies of configurations and serve as software repositories holding process aware software for users and for other servers. If the user allows it, a user-site process base may even become a software server for another user. Besides identification, the origin address also identifies the original location from where the entity (process, configuration, node or artifact) may be copied (respectively where the artifact’s actual origin address may be found), e.g. if it is not (yet) available elsewhere. Typically, a newly released configuration will immediately be copied to several release servers, perhaps initiated by the developers via multicasting [Law98], from where it will again be copied to further mirror servers, etc., from where actual users will download it, in order to avoid all users having to access the original source. Thus, the first “users” will typically be the developers’ company’s release servers, and further “users” might be company-wide intranet software servers in some of the actual users’ companies, where the actual users could access the configurations for local installation or even for immediate execution, if they are not to be installed. Furthermore, different update policies could be enforced e.g. per company or per application, e.g. with one company allowing users to individually perform updates of their own choice, and another company adopting a centrally managed style, broadcasting updates to all its users to ensure consistent and standardized installations on all involved hosts.

Thus, while download requests are originally addressed to the developers’ original servers, they are subsequently redirected to more conveniently
accessible servers, if possible. This requires a redirection mechanism for
download requests that initially routes these to a nearby server, and then
possibly to other appropriate servers in a hierarchical manner, until
finally accessing the developer’s original server if the request cannot be
satisfied elsewhere. Every server would have an ordered list of other
servers to consult when searching for the most convenient software
sources, typically organized in an as yet to be defined decentralized but
hierarchical way (in order to prevent loops), e.g. inspired by the
Internet’s Domain Name System (DNS) [Moc87]. An enterprise may thus
have a large process database, or possibly a process cache, that holds most
software used in the enterprise and can thus serve most of its employees’
software requests. If copyright or licensing restrictions are to be
enforced, then most individual users would typically not even have access
to the original developer’s site but only to their enterprise’s server, which
likewise would restrict access to its software to outsiders, in which case
redirection is a necessity and it does not serve only to enhance download
efficiency.

3.4.2.3 Addressing Scheme

Upon installation, i.e. copying of configurations consisting of process
structure and artifacts to user-site process bases, additional addresses are
created besides the origin addresses. Whenever a configuration, holding
the process structure with associated artifacts, is copied to a user process
base, its origin master address continues to point to the master (whose
address never changes, even if a process is temporarily accessed via a
backup master), and its origin node addresses of atomic nodes continue to
point to the hosts with process structure replicas holding the original
artifacts. But additionally, a deployed master address is inserted in the
local process copy pointing to the local host, and deployed artifact
addresses pointing to the artifact copies on the local host are inserted into
the local atomic node copies, which themselves are now addressed by
deployed node addresses. Thus, deployed process copies both hold
deployed addresses for local access purposes and retain origin addresses
for identification purposes, although deployed artifacts need only hold
their associated deployed node addresses, since these nodes know their
original nodes and artifacts. Thus, whenever process information or an
artifact is to be accessed locally, e.g. at runtime to invoke (execute) an
artifact holding object-code, the deployed addresses are consulted first in
order to preferably access local copies whenever available. This is the
case both for manual invocation and for automatic invocation, e.g. when
an artifact needs to access another dependent or linked artifact, e.g. due to
recursive module loading.

Fig. 3.3 provides a simplified illustration of addressing concepts, showing
two linked processes X (replicated) and Y (not replicated) developed on
three hosts and partially deployed on host D (three of four artifacts
deployed). Origin master addresses in processes, node numbers and origin artifact addresses in atomic nodes, and artifacts are shown, but deployed addresses are omitted, and node numbers in compound nodes are also omitted for simplification reasons.

Fig. 3.3: Simplified development and deployment addressing illustration with two linked processes developed on three hosts and partially deployed on host D

3.4.2.4 Cache and Server Process Base Management Policies

Different management policies for process bases are envisageable, both on user-sites and on intermediate servers, whereby some sites would use a specific policy for the whole site and other sites would manage several sections each with a different policy. Per site, a part of the available persistent storage space could be reserved for persistent storage of
configurations, i.e. which are installed and removed upon manual requests, and the rest would be available for a software cache, where installation and removal of software would be managed automatically. Thus, upon installation in a process base, a configuration could be either manually designated as persistent or automatically designated as cached, depending e.g. on factors such as frequency of use, download times, age of the software (i.e., continued availability of the originals), and ownership of the software (copyright and licensing requirements).

Although not a requirement, an advantageous complement to our deployment scheme would be an endogenous ownership mechanism for software as proposed by Cox [Cox96][Cox97] which would allow e.g. a pay-per-use scheme for software (with hardware tamper-proofing support) instead of the conventional pay-per-copy or pay-per-license schemes. In this case, software stored in a user-site process base would be more of a liability than an asset, since it uses disk space but is not owned, thus cache management of software could be advantageous here. It would also allow user access privileges to be greatly simplified, as explained next, as all software could be copied freely and instead only its execution or instantiation would be charged.

3.4.2.5 User Management

Although management of users is not studied in this thesis, different types of access privileges may be required, not only per user but also e.g. per node, per process, per company, etc. Typically, developer sites respectively release sites would register access privileges only per company and not per individual user, with company-wide intranet software servers then managing access privileges for their individual users. However, the question remains whether access privilege databases should be replicated (actively) together with the (passively) replicated (deployed) software, or managed separately. This not only requires complex user management databases but also raises questions of how the sensitive purchaser information may be managed and whether customers should be registered with developers or not.

Such problems would be eliminated to a significant degree by adopting a pay-per-use scheme for software, as presented in the previous section. If software could be copied freely, only the developers’ access privileges would need to be managed, where processes are replicated on a much smaller scale only, and only a few general categories of access privileges would be required, such as internal developer (of the same company), external developer, and user. Thus, when the paradigm shift of integrating development and deployment issues is introduced, it may be appropriate to introduce at the same time another paradigm shift concerning software ownership [Kuh96].
Fig. 3.4: Internet and intranet development and deployment scenario (involving 4 components: A, B, C, D with simplified linked processes; versions, configurations, and artifacts not shown)
3.4.2.6 Scenario

Fig. 3.4 illustrates a simplified possible Internet and intranet development and deployment scenario, involving four components and several developers and users.

3.4.3 Runtime System

As seen earlier, the runtime system provides dynamic loading and linking features, as in a modern operating system [Fra97][WG92] (issues such as binary compatibility of object code should also be considered [Fra94]). When an object code artifact is to be invoked (executed), either manually or recursively by other loading artifacts, it is loaded directly from the local user-site process base, if available, and not from a file as in conventional systems. If it is not available locally, it is automatically downloaded first whenever possible and usually installed locally, but it may even be executed immediately without persistent local installation. Configurations are validated, as described in the section on configurations in the previous chapter, i.e. checked for dependency conflicts with other configurations (and perhaps for non-dependency conflicts, if supported), either upon installation or possibly only upon execution, if an appropriate detection mechanism is available. Such a mechanism could be integrated in the loader, e.g. in the Oberon system [WG92] a list of loaded modules is managed and the module loader verifies the module key of imported modules. In this way, the loader implicitly recognizes interface-based dependency conflicts among loading and loaded modules due to the module key. This essential feature is usually already supported by the compiler, loader and linker, although typically in a language-dependent way. It could be extended to explicitly and language-independently detect dependency conflicts based not only on interfaces but also on implementations, both for revisions and variants, and possibly even to detect non-dependency conflicts once these are managed in the process too.

Two cases may be distinguished when object code artifacts are invoked. Firstly, (in Oberon terms) a single module is (partially) loaded that does not depend on other modules already loaded (is not imported by any of them), e.g. invoked by an up-call or a command, and secondly, that module may recursively load other imported modules (and loading of the first module is completed, and dynamic linking is performed). Loading of an imported module is relatively straightforward, since it should be found preferably through a direct predecessor dependency or link from the importing module (object code artifact), or at least within the transitive hull of all predecessor dependencies or links. In either case, this should lead to exactly one match (i.e. a specific version), even if the search among these nodes is performed by unversioned filename only (e.g. “A.Obj”), which should be the default – it is essential that the
programming language (with compiler and runtime system) need not be aware of versioning concepts. A conventional file-based loader thus needs to be modified to search for the imported module by following predecessor dependencies or links from the importing module, instead of searching in the filing system, although the filing system may still be accessed for software that is not yet available in processes. If the imported module is not yet installed locally, it may either be retrieved from an appropriate remote server, or loading may be aborted, according to the selected policy that limits the search range (which is potentially infinite). Also, deployed software needs to retain its process awareness both in persistent and in loaded state, i.e. even a loaded module contains a field identifying its (local copy of its) node (its deployed node address), thus enabling its dependencies to be accessed at any time. E.g. in fig. 3.5, consisting of two linked processes “Server” and “Talk” with two revisions each, using a specific version of “Talk” (e.g. with node id 6) requires the appropriate version of the imported module “TCP” (e.g. the one with node id 3) from the process “Server” to be loaded.

Loading of a non-imported module may be somewhat more complex, since the desired version is potentially not fully specified. Searching is performed among the successor nodes of the module which initiated the up-call (or perhaps of the loaded operating system or other modules in the case of a user-executed command), which may lead to several matching (valid) versions or to none, if the required module is in a different linked process, since links are unidirectional (links only point to predecessor nodes, but corresponding successor nodes are not registered), unlike dependencies that always exist in pairs of opposites. Policies must determine which of several versions to select (e.g. always the newest, respectively those compatible with already loaded or installed modules) and within what (potentially infinite) limits to search for matching artifacts (local process base, filing system, or specified remote process base servers). An artifact found in a different process may only be used if a link and predecessor dependency path exists back to the source artifact of the search. E.g. in fig. 3.5, if “Talk” is to be used, it is not immediately clear whether the one with node id 5 or 6 is to be invoked, but it may be determined e.g. if one of the “TCP” versions is already loaded.
Fig. 3.5: Sample linked processes available for deployment (installation and invocation)
Leer - Vide - Empty
Chapter 4
Framework Implementation

4.1 Introduction
An experimental prototype based on the design presented in the previous chapter has been implemented. The prototype implementation has grown in an evolutionary manner and its purpose is only to provide a proof-of-concept demonstration, therefore not all features have been implemented yet or have been implemented precisely as described in the design, and the prototype is structured differently in some details. Similarly, experiences gained with the prototype have lead to modifications of the design, therefore the emphasis here is more on an overview than on issues which are continuously undergoing changes, e.g. a detailed description of process operations is omitted. The focus of this chapter is mainly on those implementation aspects I provided or where I was involved in-depth, while presenting other issues as required for the understanding of the complete system.

The Oberon System 3 (Gadgets V1.5) [Gut94][WG92][WG89][Wür94] and its Oberon programming language [Wir88] were chosen as platform and language for the implementation, due to their advantages such as simplicity and flexibility, dynamic linking and loading, strongly typed language, built-in object and messaging system, gadgets graphical user interface, and relative independence from underlying platforms. This however required a distributed messaging and simple object request broker (ORB) system to be implemented in Oberon, as these did not yet exist. If the implementation were to be redone, the CORBA ORB architecture and the Java language and system would be considered today besides Oberon, but these were not yet widely available at that time. Similarly, the Oberon-2 language [Mös95], although desirable, was not available in time for the Oberon systems we used for us to consider it. Usage of Oberon for our own implementation also influenced the design of the process modeling language and process framework, since it lead us to assume that software components will increasingly be implemented in modern programming languages and systems providing features such as dynamic linking and loading. This greatly simplifies the build process.
which therefore does not require sophisticated support, unlike in other process frameworks [CS97].

We started using Oberon System 3 on both Apple Macintosh and Sun SPARCstation UNIX computers, and during the course of the project, both platforms had to be migrated, the Macintosh platform from 68K to PowerMac computers, and the SPARCsations from SunOS 4 to Solaris. Therefore, we required up to four different variants of certain low-level modules (e.g. networking, or access to platform-specific features), but the majority of our Oberon modules could be used unmodified on all four platforms. Even where variants were not distinguished, the evolution of the prototype provided many revisions. This clearly demonstrated the need for a powerful version and configuration management system, even if our own system could only be applied yet to certain subsections of the project towards its completion due to the limited-functionality implementation.

4.2 Module Hierarchy

This section provides an overview of the main module hierarchy that forms the DIPS process framework. Fig. 4.1 shows an overview of the hierarchy of the main components, while fig. 4.2 shows the detailed (but still simplified) module hierarchy of the components. Referring to the components defined in the architecture’s design, the basic and distribution services, the common distributed process services, the tool integration services, the development system and the deployment system can be identified (no separate runtime system has been implemented yet, as explained below). The system was initially implemented only for single-workstation/single-user operation, comprising all modules respectively components on the left half of each of the two figures (basic services, common process services, development system consisting mainly of user interface, tool integration services, and tools). Soon thereafter, the system was extended for distributed operation, by adding all modules respectively components on the right half of each of the two figures (distribution services, distributed development system, distributed deployment system, and WWW server), and by modifying some of the original modules (e.g. the common process services).
Fig. 4.1: DIPS main component hierarchy overview

Fig. 4.2: Simplified DIPS module hierarchy
In the following, all of the main modules are briefly listed in a bottom-up order, and in subsequent sections, some issues are presented in greater detail.

4.2.1 Basic and Distribution Services

4.2.1.1 Single-workstation System

**ElemLists:** Manages ordered lists of objects (used by many modules; not all imports shown).

**Utilities:** Auxiliary module for managing dynamic attributes (used by many modules; not all imports shown).

**HFS:** Provides access to the Macintosh’s Hierarchical File System and files with Macintosh resource and data forks, used for non-Oberon artifacts and files.

**FileServices:** Hides platform-specific access to the host’s filing system.

**AppleEvents:** Provides access to Macintosh AppleEvents.

**Finder:** Allows external (non-Oberon) tools for non-Oberon artifacts to be invoked by Oberon commands on the Macintosh via AppleEvents and Macintosh Finder.

4.2.1.2 Distributed System

**TCP:** Provides TCP/IP transport for network communications (OSI transport layer and lower), standard data type serialization (OSI presentation layer), and domain name resolving; includes finalization of connections (used by many modules; not all imports shown) [Sch94].

**Queues:** Manages FIFO queues, e.g. of messages waiting for delivery in a server (used by many modules; not all imports shown).

**Users:** Manages a simple database mapping users to hosts and vice versa, replicated manually (used by many modules; not all imports shown).

**Talk:** Allows distributed project members to communicate, with characterwise (immediate) replication of text and graphics (Oberon gadgets), inspired by the UNIX “talk” application.

**ORBObjects:** Manages a list of currently loaded (instantiated) objects (e.g. processes and artifacts) to prevent inconsistent duplication; includes finalization of objects in the list.

**DistMessages:** Base module defining distributable message types based on the system’s “Objects.ObjMsg” type.

**DistServices:** Provides basic extensible synchronous and asynchronous communication services and protocols for clients and servers (OSI
application layer), running as state machines in background Oberon tasks; also provides connection dialog control (OSI session layer).

DistMessaging: Based on both DistServices and DistMessages, provides distributed messaging operations [Sch97a].

DistFiles: Based on DistServices, allows files to be sent via network.

ORBConnections: Manages a list of currently open network connections for reuse to reduce superfluous opening/closing overhead.

4.2.2 Common Distributed Process Services

ContextProt (context protocol): Defines basic DistMessages distributable messages to navigate the process context and create, assign, confirm and retrieve artifacts therein, thus providing the main interface of the tool integration services.

Places: The main process framework module, defines and manages node (based on type “Gadgets.Object”) and dependency objects, implementing basic operations for process definition and enactment; uses ContextProt messages to communicate with artifacts in the process.

PlaceDocs: Defines and manages processes as Oberon documents of type “Documents.Document,” preventing load inconsistencies via ORBObjects.

Configurations: Defines and manages configurations in processes.

4.2.3 Tool Integration Services

Originators: Allows Oberon or external artifacts to be created by specifying an appropriate combination of artifact type/name, existing template, or Oberon command.

Integrate: Allows CHIPS-generated artifacts to be compiled. Similarly, CHIPSCreators (a CHIPS module, not shown) allows CHIPS artifacts to be created.

CHIPSDocs (a CHIPS module): Defines and manages CHIPS artifacts as Oberon documents of type “Documents.Document;” used e.g. by CHIPSCreators.

4.2.4 Development System

4.2.4.1 Single-workstation System

PlaceFrames: The main development system user interface module, defines and manages process displays as an extension of type “Panels.Panel” and provides the user interface for process definition and enactment operations.

PlaceUtils: Provides auxiliary process handling operations.
SnapShot: Writes a current process snapshot in Apple QuickDraw 3D format (nested text), with or without configurations, for display by an external tool or browser plugin.

ViewManager: Allows artifacts to be opened with different compatible views or tools.

### 4.2.4.2 Distributed System

ReplProjects (replicated projects): The main distributed development system module, extends process definition and enactment operations in Places for distributed operation, managing synchronization among replicated process copies via master copy, and managing user (developer) access privileges.

DistViewManager: Manages distributed views of non-local artifacts (currently only static views).

### 4.2.5 Deployment System

Installer: Allows artifacts to be retrieved from distributed processes and installed locally.

WWW Server: Consists of several modules (described in a separate section). It uses many modules of the basic and the common distributed process services and also some CHIPS modules, since it must access process and artifact data structures for translation and presentation on the WWW. It also uses some development system user interface modules in order to provide a similar display on the WWW as for developers to avoid having to implement another display, although this is not a design requirement.

### 4.2.6 CHIPS

Tools generated by the CHIPS tool generator provide an example of possible tool integration in processes. They use messages extending ContextProt (and thus DistMessages) messages to communicate directly with artifacts in the process context instead of only using files on a lower level of integration.

CHIPS evolved from the original GIPSY compiler-compiler and incremental editor to a non-incremental version and was ported to Oberon. Besides the original comprehensive GIPSY thesis [Mar94], little documentation about the more recent CHIPS implementation exists yet, although the user interface of early versions is extensively documented [HM95a][HM95b][WM96], and [MWSS96][Mur97b] briefly outline some design issues. Therefore, I present an overview of its implementation here, but as this was the only project part where I was not involved in depth but which I mainly employed only as user, I refrain from going into details. Fig. 4.3 shows the simplified module hierarchy of the CHIPS
kernel, consisting of the basic message modules shared with the process framework, utility modules, the main lexical and semantical modules, document and view modules, a programming interface and an independent user interface.

Fig. 4.3: Simplified CHIPS kernel component and module hierarchy

Tool components are specified textually (as a language) as scanner, parser, and semantic extensions in the CHIPS/L language which is based on extensible attribute grammars (EAGs). CHIPS, consisting of the language-independent CHIPS kernel and CHIPS/L-specific scanner, parser, and extensions, then translates the specifications to Oberon source code modules implementing the tool components, which may then be compiled to create the tools, possibly together with external auxiliary and messaging modules defining messages based on ContextProt that allow artifacts to interact with each other.
Tools specified and generated by CHIPS include scanner, parser, and semantic extensions of CHIPS itself (allowing CHIPS to be developed and improved in a bootstrapping manner), an Oberon compiler front-end with scanner, parser, declaration analysis and type checking, an environment allowing process definitions to be specified textually and translated to framework-compatible processes (as an alternative to defining processes using the graphical user interface), some calculators, a timing specifier (to be used together with timing analysis, mentioned below), and others. Extensions of the Oberon compiler front-end include a HTML generator for Oberon source code (used by the WWW server), a communications module generator (used by the distribution services, described below), a source code formatter, and different types of timing analysis components (e.g. to check real-time constraints of primitive-recursive Oberon procedures).

4.3 Objects and Algorithms

4.3.1 Process Objects

A brief overview is given of the main objects and definitions provided by the implementation of the non-distributed process framework (figs. 4.4c-f), based on Oberon System 3 (figs. 4.4a and b), whereafter two selected issues I designed and implemented are described in somewhat greater detail, the serialization algorithm and the rollback algorithm. The extensions for distributed operation and the WWW server, both of which I also designed and implemented, are subsequently presented in separate sections.

ContextProt provides basic extensible messages and forms the main interface of the tool integration services. Distributable messages based on DistMessages include a message (ObjMsg) for artifact access (extended e.g. by confirmation messages), which includes an optional method to first collect an artifact's process context, and a message to create an artifact (CreatorMsg). Messages for local operation include one to assign an artifact's process node (which defines its context in the process) to it (SetContext) and one to retrieve neighboring nodes in the process context (GetNextContext). For process awareness, required to participate actively both in development and deployment activities, an artifact needs to handle at least the last two messages, by managing (setting respectively retrieving) its own deployed node address, plus optionally a pointer for quicker access to its loaded node (as in CHIPS-based artifacts), in order to be able to access its node and process context (e.g. for related artifact creation or confirmation, configuration validation, and loading of dependent artifacts). Artifacts without these capabilities, e.g. non-Oberon artifacts that do not implement Oberon messaging conventions, may either be accessed via specific (CORBA-style) object adapters, or are used only in a less integrated way.
Additionally, for high-level tool integration, tool-specific messages should be defined, otherwise default messages available in modules Originators and Places are used. One message, a creation message, should be an extension of "ContextProt.CreatorMsg;" it is sent to the node when its state is about to become editable, to create a new artifact; the state may become editable when the message returns with its "res" field set to "ContextProt.success." CHIPS tools often use the "CHIPSCreators.CreatorMsg" (extended from "ContextProt.CreatorMsg") or extensions thereof, together with the tool's name as a parameter in the node's "com" field, whereas "Originators.BasicCreatorMsg" is a default message to be used by non-CHIPS or non-Oberon tools, which can e.g. call an Oberon command defined in the node's "com" field (e.g. "Builder.Compile" for Oberon compilation). Similarly, the node's "view" field can be used to define an artifact's view as an Oberon command if non-CHIPS or non-Oberon document views are required, e.g. "Browser.ShowDef" for the Oberon module interface browser for Oberon object code, or "Finder.Open" to invoke a non-Oberon tool. Another message, a confirmation message, should be an extension of "ContextProt.ObjMsg;" it is sent to the node when its state is about to become confirmed, to confirm its artifact; the state may become confirmed when the message returns with its "res" field set to "ContextProt.success." CHIPS tools often use the "Structures.ParseMsg" (extended from "ContextProt.ObjMsg"), extensions thereof, or other tool-specific messages, whereas "Places.SignatureMsg" is a default message which currently always returns "ContextProt.success."

Besides defining such messages, a tool should also be able to handle these associated messages in an appropriate way, i.e. upon receipt of its tool-specific creation message via its node, a tool should create and assign a new artifact to its node (i.e. it should provide a "New" command that creates a new instance of the tool environment, the artifact), which is of a type that will understand the confirmation message, and upon receipt of the confirmation message, the tool should perform the desired syntactical or semantical checks on its artifact (e.g. parsing or timing analysis) and return an appropriate result to permit or deny confirmation. It is also envisageable to use messages which cannot automatically return a positive result, e.g. if confirmation involves checking conditions that cannot be formally expressed, thus requiring the developer to manually perform confirmation. Fig. 4.4g shows an excerpt of the TimingMsgs module definition used in the Oberon timing environment example seen earlier (fig. 2.5a), showing the two messages used during confirmation defined as extensions of "ContextProt.ObjMsg;" as mentioned earlier, the CheckTimingMsg is the confirmation message, whereas that one uses the TimingMsg internally to perform timing analysis on individual procedures.
For historical reasons, the nodes (or vertices) in the (multi-dimensional monopartite) graph that makes up the process structure are denoted “places,” in analogy to places and transitions in (bipartite) Petri nets, and the (directed) edges are denoted “dependencies.” Thus, module Places defines the main process objects, the nodes (atomic: “Places.Place,” compound: “Places.CompPlace,” extended from “Places.Place”) and dependencies (“Places.Dependency”). Every node manages numerous attributes, some as record fields and most as dynamic attributes (a facility inherited from “Gadgets.Object”), holding its node id, history layer, node name, current state, list of dependencies, a pointer to its artifact where applicable (in the “dObj” field), artifact filenames (process and unversioned, as detailed below), artifact host address, tool, creation function, required predicate, associated developer, date and timing information, and other metainformation. The top compound node of a process additionally manages the master host address(es) and lists with configurations, users, process replica addresses, variants (not yet implemented), and other information (currently, these fields are available in all compound nodes, but used only in the topmost one; a separate object type is therefore planned for the top compound node). Only simple user management is currently implemented, allowing write privileges to either one specified developer or to all users per atomic node, and allowing read privileges to a list of specified users or to all users per process, not distinguishing developers and other users. A configuration holds a list (in the “startPlaces” field) of node ids designating the leaf nodes of a configuration, and a filter defining the nodes to be included in the configuration from the transitive hull of predecessor nodes from the leaf nodes. Currently, the filter only uses simple filename-based rules (e.g. include “.Mod” or “.Obj”), but more sophisticated rules are envisaged.

Every node has a method field (inherited from “Objects.Object”) that holds a replaceable procedure (handler) of type “Objects.Handler” that processes messages whose type is based on “Objects.ObjMsg,” following Oberon System 3 messaging conventions. Standard Oberon System 3 messages are used to access node attributes, copy nodes, and handle loading and storing of nodes, and ContextProt messages are used to access artifacts. More specialized methods and procedures are used for handling definition and enactment operations, propagation of operations to encompassing compound nodes, and for some artifact access operations. For distributed operation, the handler and methods are replaced (extended) by ones defined in module ReplProjects which manage synchronization of replicated entities, using distributable messages, and also call the original procedures defined in Places.

As illustrated in fig. 4.5, as loaded in volatile memory, a node is an object that holds a pointer to a list object listing its dependencies, each of which is also an object, containing a pointer to a neighbor node and a field with
the dependency’s type (and every neighbor node has a respective opposite dependency). Eight dependency types are available: pred and succ for predecessor and successor nodes, sub and super for the process hierarchy, older and newer for the process history, and special pred and succ across different history layers. Usage of pointers and objects allows for efficient navigation, but pointers obviously require a translation for persistent storage, since they refer to objects in volatile memory. Further research is required to reveal whether alternative data structures such as lists with node id numbers would be more beneficial overall. Also, additional dependency types will be required when process linking is implemented, to hold addresses of remote nodes or configurations instead of pointers.

MODULE Objects;
IMPORT Files;

TYPE
    Object* = POINTER TO ObjDesc;
    ObjDesc* = RECORD
        ...
        handle*: Handler;
    END;

    ObjMsg* = RECORD
        ...
    END;

    FileMsg* = RECORD(ObjMsg)
        ...
    END;

    Handler* = PROCEDURE(obj: Object; VAR M: ObjMsg);

END Objects.

Fig. 4.4a: Excerpt of Oberon System 3 Objects module definition

MODULE Gadgets;
IMPORT Objects, Attributes;

TYPE
    Object* = POINTER TO ObjDesc;
    ObjDesc* = RECORD(Objects.ObjDesc)
        attr*: Attributes.Attr;
    END;

    ...

END Gadgets.

Fig. 4.4b: Excerpt of Oberon System 3 Gadgets module definition
MODULE DistMessages;

IMPORT Objects;

TYPE
  MsgPtr* = POINTER TO Msg;
  Msg* = RECORD(Objects.ObjMsg);
    Read*: PROCEDURE(VAR M: Objects.ObjMsg);
    Write*: PROCEDURE(VAR M: Objects.ObjMsg);
    Copy*: PROCEDURE(VAR M0, M1: Objects.ObjMsg);
    Len*: PROCEDURE(VAR M: Objects.ObjMsg): INTEGER;
    sender*: LONGINT;
    user*: ARRAY 32 OF CHAR;
    type*: ARRAY 64 OF CHAR;
    res*: INTEGER;
END;

  PtrMsgPtr* = POINTER TO PtrMsg;
  PtrMsg* = RECORD(Msg)
    ...
END;

  RemoteMsg* = RECORD(Msg); (*Dist*)
    ...
END;

  RemotePtrMsg* = RECORD(PtrMsg); (*Dist*)
    ...
END;

VAR
  New*: MsgPtr;
    ...
END DistMessages.

Fig. 4.4c: Excerpt of DistMessages module definition
MODULE ContextProt;

IMPORT Objects, Gadgets, DistMessages;

TYPE
    ObjMsgPtr* = POINTER TO ObjMsg;
    ObjMsg* = RECORD(DistMessages.RemotePtrMsg) (* Dist *)
        getContext*: PROCEDURE(msg: Objects.ObjMsg;
            homeContext: Objects.Object): BOOLEAN;
            env*: Objects.Object;
        END;
    CreatorMsgPtr* = POINTER TO CreatorMsg;
    CreatorMsg* = RECORD(ObjMsg) (* Dist *)
        ...  
        END;
    SetContextMsg* = RECORD(Objects.ObjMsg)
        ...  
        END;
    GetNextContextMsg* = RECORD(Objects.ObjMsg)
        ...  
        END;
    ...  
END ContextProt.

Fig. 4.4d: Excerpt of ContextProt module definition
MODULE Places;

IMPORT Objects, Gadgets, ElemLists, DistMessages, ContextProt;

TYPE
  Dependency* = POINTER TO DepDesc;
  DepDesc* = RECORD(ElemLists.ElemDesc)
    kind*: INTEGER;
    vertex*: Vertex;
  END;

  Vertex* = POINTER TO VertexDesc;
  VertexDesc* = RECORD(Gadgets.ObjDesc)
    mark*: Objects.Object;
    dependencies*: ElemLists.List;
    old*: BOOLEAN;
  END;

  Place* = POINTER TO PlaceDesc;
  PlaceDesc* = RECORD(VertexDesc)
    dObj*: Documents.Document;
    rollbackRoot*: BOOLEAN;
    ...
  END;

  CompPlace* = POINTER TO CompPlaceDesc;
  CompPlaceDesc* = RECORD(PlaceDesc)
    configs*, partners*, users*: ElemLists.List;
    ...
  END;

  PlaceMsg* = RECORD(DistMessages.RemoteMsg) (*Dist*)
    load*: BOOLEAN;
  END;
  ...
END Places.

Fig. 4.4e: Excerpt of Places module definition
MODULE Configurations;

IMPORT ElemLists, Places;

TYPE
  Configuration* = POINTER TO ConfigurationDesc;
  ConfigurationDesc* = RECORD(ElemLists.ElemDesc)
    name*, filter*: ARRAY 32 OF CHAR;
    startPlaces*: ElemLists.List;
  END;

StartPlace = POINTER TO StartPlaceDesc;
StartPlaceDesc = RECORD(ElemLists.ElemDesc)
  id: LONGINT;
END;

... END Configurations.

Fig. 4.4f: Excerpt of Configurations module definition

MODULE TimingMsgs;

IMPORT ContextProt, DistMessages;

TYPE
  TimingMsgPtr* = POINTER TO TimingMsg;
  TimingMsg* = RECORD(ContextProt.ObjMsg)(*Dist*)
    name*: ARRAY 32 OF CHAR;
    time*: LONGINT;
  END;

CheckTimingMsgPtr* = POINTER TO CheckTimingMsg;
CheckTimingMsg* = RECORD(ContextProt.ObjMsg)
END;

...  END TimingMsgs.

Fig. 4.4g: Excerpt of TimingMsgs module definition
4.3.2 Serialization Algorithm

For persistent storage and for transmission via network to manage process replicas, the multi-dimensional process structure is serialized into a file. A process is stored in a file of type “Documents.Document” that contains both a “PlaceFrames.PlacePanel” (an extension of “Panels.Panel”) that holds display defaults and methods and an object library of type “Objects.Library” that holds all node objects that make up the process structure, with their dependencies, attributes, and methods. Multiple documents or panels of the same process may be opened on the same machine, but these will only be different views of the same process, since loaded processes are registered in the ORBObjects list. Artifacts are stored in separate files, as will be explained below, but their storing can also be invoked automatically during process storing (if they are Oberon types).

For storing of the process nodes, Oberon System 3’s library mechanism is employed to store them in an “Objects.Library.” As defined by conventions in Oberon System 3’s base Objects module, this involves sending an “Objects.BindMsg” to all nodes in the process to bind (register) them in the library, then invoking “Objects.StoreLibrary” which sends an “Objects.FileMsg” (with id = store) to all objects (nodes) bound in the library to carry out their individual serialization and to invoke the separate storing of associated artifacts. Similarly, loading involves calling “Objects.LoadLibrary” (if the process is not yet available in the ORBObjects list), which calls the generator command for all of the library’s stored objects to allocate them, and then sends each of them an “Objects.FileMsg” (with id = load) that causes them to read their data from file to complete instantiation.

Fig. 4.5: Dependency data structures of atomic node C (bottom) in sample process A (top)
Following the library mechanism's conventions, it must be ensured that the "Objects.BindMsg" visits every node and every dependency in the multi-dimensional process structure exactly once, i.e. that the message is passed on from node to node in a well-defined order. Thus, in order to navigate among the history layers in the process in an orderly manner, information is used during serialization that has been recorded when the history layers have been created, i.e. when rollback operations have been performed. Since initiation of rollbacks is currently limited to one node per operation, that node can be identified by setting its "rollbackRoot" field during rollback, and since every rollback produces a separate history layer (which may contain many affected nodes), every history layer is visited exactly once by a subsequent traversal algorithm if it follows older dependencies only where the "rollbackRoot" field is set. Thus, the "Objects.BindMsg" commences with the topmost (current) history layer and recursively visits every history layer once, moving from one layer to the next older layer at every node whose "rollbackRoot" field is set. Per history layer, the "Objects.BindMsg" initially follows super dependencies as required to commence with the outermost compound node and then recursively visits all contained nodes by following the sub dependencies.

After binding, the library sequentially sends the "Objects.FileMsg" to all its bound objects, i.e. the nodes, which thus need to serialize only their own dependencies. For storage, the dependency's pointer referencing a neighbor node is replaced by that node's library reference (a number allocated by the library during binding). Loading is more straightforward: after allocation of all of its objects (the nodes), the library sequentially sends them each an "Objects.FileMsg," which causes every node to replace each of its dependencies' library references with a pointer to the respective neighboring node.

4.3.3 Rollback Algorithm

The rollback algorithm is rather intricate, yet has proven to work reliably. Due to space constraints, only a brief overview can be given here, but fig. 4.6 illustrates the result of two subsequent rollback operations carried out on the "Tool" sample process (as also seen earlier in fig. 2.9). While rollback (or iteration) is an enactment operation, the corresponding definition operation, for process evolution, has not yet been fully defined and implemented, and is expected to be even more complex, yet could probably reuse parts of the rollback implementation.

As mentioned earlier, a rollback may currently be initiated at only one node per operation (but may affect numerous dependent successor nodes), and always resets the affected nodes (respectively their copies created during rollback) to the initial planned state. Besides the normal rollback that retains the process history, a simple rollback is also available in
certain cases that does not preserve the history but simply resets the existing nodes and thus overwrites previous results.

Fig. 4.6: Tool sample process after enactment, rollback on node Src, further enactment, rollback on node Spec, and continued enactment

As explained in the chapter on the process modeling language, in a normal rollback, for all nodes affected by the rollback of the initiating node (the set of rollback-nodes), first the single common compound node containing all of them is searched recursively. That node and all its contained nodes form the set of containing-rollback-nodes, which contains the set of rollback-nodes. Then, all rollback-nodes are unmarked (their “mark” field, destined for temporary usage, is set to NIL), and all the other nodes in the process are marked. The unmarked nodes are collected in a temporary list, duplicated, assigned to their respective originals’ mark field, and the originals’ state is set to reflect a full rollback. The remaining (non-rollback) nodes in the containing-rollback-nodes set are also copied and assigned to their originals’ mark field, and the originals’ state is set to reflect a partial rollback. Then, for every rollback-node (which has previously been collected in a list), each of its dependencies is handled according to one of 28 cases, depending on the dependency type.
and the node’s state, duplicating, deleting, moving, and/or modifying dependencies. For instance, the older/newer dependencies among a (partially) rolled back node that remains in the current layer and the respective node in the next older layer are moved to the node copy in the newly created history layer, and new older/newer dependencies are inserted among that copy and the original in the current layer. In this way, a new history layer is created out of original or copied nodes (retrieved from their mark fields), reflecting the dependency structure of the original nodes, whereby fully rolled back nodes end up in the process history and their copies are inserted in the current layer, and vice versa partially rolled back nodes remain in the current layer and their copies are moved to the history layer. Finally, all compound places in the containing-rollback-nodes set are recursively traversed once more in a bottom-up manner (i.e., when the visiting algorithm returns) to allow possible state changes to be propagated to containing nodes as required.

4.4 DISTRIBUTED SYSTEM

4.4.1 Overview

Initially, the process framework was built only for single-workstation (single-user) operation, to explore the first implementation of process concepts. Oberon System 3 messaging concepts were already employed in view of facilitating independent management of process and heterogeneous artifact entities. Since the main benefits of process support become evident only when multiple developers/users are involved on multiple workstations, a distributed implementation was commenced soon thereafter. For reasons of simplicity, a client/server solution was first implemented which managed all process data and artifacts on a single server and transferred them to other workstations as required. Then, the current more efficient and more scalable distributed objects implementation was designed and implemented.

Besides addition of the extensive distribution functionality, implementation of the distributed object-based system required two modifications to the core of the initial single-workstation system, as reflected by the insertion of the two base modules DistMessages and ORBObjects in the module hierarchy. Firstly, all messages used for communication among processes and artifacts had to be based on a transparently distributable message type, provided by module DistMessages, as explained below. Secondly, every workstation required centralized (runtime) management of loaded distributed objects (processes and artifacts), provided by module ORBObjects, to prevent inconsistencies such as duplicate loading (invocation) of the same object, similar to the list of opened files in Oberon systems [WG92]. While these features did not exist in Oberon System 3, the system did provide basic object and message type definitions which could be extended for our purposes.
The distributed services, used by both the development and the deployment system, consist of several modules that provide basic communication services running as state machines in background Oberon tasks, designed to work with the limitations of Oberon single-process non-preemptive multi-tasking (here, process means an executing program, in operating system terminology). Since the Oberon system does not provide threads, special care must be taken to ensure that the single process is not unduly blocked. To avoid blocking of the process by network operations, network operations are performed in background Oberon tasks if possible (unless initiated by the local user), whereby incoming data from the network is only read when it is available in the in-buffer, and outgoing data may only be written when there is sufficient space in the out-buffer (else the data is queued if possible). Otherwise control is immediately passed on to other tasks, thus blocking waiting for the network is largely avoided unless it concerns an operation effected by the local user. Vice versa, few local (non-distributed) operations are envisageable that could be triggered by the local user whose execution time surpasses the generous network timeout delays, i.e. that could cause background network operations to be delayed until abortion. E.g., due to separate compilation, it is unlikely that a local compiling session initiated by the local user in between network polling by a background task lasts long enough to interfere with background timing constraints. Nevertheless, if need be, most distributed operations may be retried without harm, e.g. synchronization protocols utilize an operation stamp to uniquely identify operations, as detailed below.

Extended services include distributed messaging, described next, and a service for distributing files. Based on these, the main issue in the distributed development system is synchronization of process replicas and operations, described separately, whereas the deployment system currently includes a distributed installer, but no management of user-site process bases or adaptation of runtime systems, as these issues have not yet been implemented. However, the WWW server also forms part of the deployment system and could even be regarded as part of the development system or could be extended for such purposes. For deployment purposes, the WWW server provides process visualization independently from the process framework, therefore it uses only the common process services, but no distribution services besides TCP, and is described in a separate section.

4.4.2 Distributed Messaging

As detailed elsewhere [Sch97a], DistMessages defines message type "DistMessages.Msg," upon which all distributed messages in the process framework are based instead of directly on "Objects.ObjMsg." While "DistMessages.Msg" is also based on "Objects.ObjMsg" and thus retains compatibility with Oberon System 3 messaging, which therefore merely
required a recompilation for many involved modules, it allows the same message type to be used transparently both locally and in distributed operations, whereas conventional Oberon System 3 messaging is limited to local operations.

For distributed operation, a message from a source object has to be serialized (also known as sequentializing, linearizing, or marshaling), transferred to the destination host via network, reassembled there and passed on to the destination object on that host. If required, an answer is sent back the same way to the source object, whose execution is either blocked until the answer is received, or which receives the answer asynchronously, if any. So far, only one-way messages and synchronous (blocking wait) answering have been implemented.

Both serialization and reassembly of a message require information about the data types of the message's individual fields. Even if the message contains no pointers, it cannot be transmitted simply as an unstructured block of bytes, as source and destination platforms may have different data type representations, e.g. type sizes or byte ordering. Standard Oberon systems (V4 and System 3) do not provide an integrated and portable runtime type access mechanism (also known as reflection or metaprogramming) (although other Oberon systems do [Ste97]). Since we were using Oberon on four different platforms and did not want to be dependent on modifications to future Oberon systems, the best portable approach was to access the type information at compile-time to create message-specific communications support modules. Due to the dynamic linking and loading possibilities, no runtime flexibility is compromised in this way, as the specific support module procedures are accessed automatically via Oberon commands. Furthermore, reassembly (allocation and initialization) of an arbitrary received message requires the message to be a (heap-based) dynamic object, whereas the system's "Objects.ObjMsg" defines (only) a record type. The message-specific support module therefore defines a pointer type, allocation and initialization procedures, as well as read, write, copy, and length calculation procedures for every distributable message type, the latter four of which are assigned to the respective message fields (procedure variables defined in "DistMessages.Msg") upon its initialization. These six procedures are thus used per message to enable transparent and distributed operation via network, e.g. the write procedure performs message serialization. If the Oberon-2 language were used instead of standard Oberon, initialization could be simplified, since methods could be used instead of procedure variables, and methods are assigned automatically upon instantiation of the method's type. However, Oberon-2 compilers were not available in time on all platforms we used, therefore this was not an option for us.
Creation of the message-specific support modules is rather straightforward and is handled automatically by the "GenDist" environment, a CHIPS-generated Oberon environment (compiler frontend tool), consisting of Oberon scanner, parser, and communications support module generator. Every module containing distributable messages is handled and a support module is generated in source code whose name ends in "Gen"; four such modules were generated for the process framework (fig. 4.7), and further ones for specific CHIPS tools. For example, the generated source code for "Places.PlaceMsg" in module PlacesGen is shown in fig. 4.8; further details can be found in [Sch97a].

**Fig. 4.7: Automatically generated communications support modules (excerpt of DIPS module hierarchy)**
MODULE PlacesGen; (* Generated by GenDistMessages *)

IMPORT Objects, TCP, DM := DistMessaging, Places, DistMessagesGen;

TYPE
  PlaceMsgPtr* = POINTER TO Places.PlaceMsg;

PROCEDURE ReadPlaceMsg*(VAR M: Objects.ObjMsg);
BEGIN
  WITH M: Places.PlaceMsg DO
  DistMessagesGen.ReadRemoteMsg(M);
  TCP.ReadBool(DM.msgC, M.load);
END ReadPlaceMsg;

PROCEDURE WritePlaceMsg*(VAR M: Objects.ObjMsg);
BEGIN
  WITH M: Places.PlaceMsg DO
  DistMessagesGen.WriteRemoteMsg(M);
  TCP.WriteBool(DM.msgC, M.load);
END WritePlaceMsg;

PROCEDURE CopyPlaceMsg*(VAR MO, Ml: Objects.ObjMsg);
BEGIN
  WITH MO : Places.PlaceMsg DO
  WITH Ml: Places.PlaceMsg DO
    DistMessagesGen.CopyRemoteMsg(MO, Ml);
    Ml.load := MO.load;
  END;
END CopyPlaceMsg;

PROCEDURE LenPlaceMsg*(VAR M: Objects.ObjMsg): INTEGER;
BEGIN
  WITH M: Places.PlaceMsg DO
  RETURN DistMessagesGen.LenRemoteMsg(M) + 1
END LenPlaceMsg;

PROCEDURE InitPlaceMsg*;
BEGIN
  DM.NewMsg.Read := ReadPlaceMsg;
  DM.NewMsg.Write := WritePlaceMsg;
  DM.NewMsg.Copy := CopyPlaceMsg;
  DM.NewMsg.Len := LenPlaceMsg;
  COPY("Places.PlaceMsg", DM.NewMsg.type);
END InitPlaceMsg;

PROCEDURE NewPlaceMsg*;
VAR M: PlaceMsgPtr;
BEGIN
  NEW(M);
  DM.NewMsg := M;
  InitPlaceMsg;
END NewPlaceMsg;

END PlacesGen.

Fig. 4.8: Definition of automatically generated support module PlacesGen
4.4.3 Synchronization Protocols

Module ReplProjects provides the main part of the distributed development system. It extends the functionality in the Places and other modules to support distributed operation, passing on ContextProt and process operation messages among process replicas and distributed artifacts, and synchronizing process replicas via a master process copy. Per process, the master maintains a list of currently on-line hosts holding process replicas, which receive broadcasts of process operations. When a replica initiates an operation and is not yet in the list, it is added to it, and when it does not respond, it is removed from the list.

Whenever a process is to be loaded or an enactment operation is to be carried out and the process is not the master itself, the local process replica is synchronized with the master via a "ReplProjects.SyncProjMsg." If the local copy is not current any more, the current process is first retrieved from the master (possibly after referral to a newer master if it has changed), before the loading or the operation can be completed, in order to ensure consistency. The master's host address only needs to be specified for the first loading operation (which implies registration of a new developer with the process), together with the process name, thereafter the process name suffices, as the process replica that is obtained from the master holds the master's address (both in loaded and persistent state). After synchronization, when an enactment operation is carried out, it must be sent to the master, using a "ReplProjects.PlaceMsg," and the master must broadcast it to the process replicas on the other hosts that are currently on-line excluding the sender host, or if it has been carried out on the master, it must be broadcast directly to all the other hosts. Only the initiating node and its new state are specified in the message, which suffices for every replica to re-enact the operation locally.

Only the relatively infrequent enactment operations that modify node states or the process structure need to be sent and broadcast, i.e. at most three operations per node (planned to editable, editable to confirmed, possibly rollback). Other operations, in particular long-duration editing operations of artifacts (during which the associated node remains in state editable) occur only locally, as artifacts are not replicated. Definition operations all need to broadcast, but only upon completion, i.e. upon mode change from definition mode to enactment mode (when definition operations are concluded and enactment operations begin (again)), by sending the results as a whole. However, unlike the currently implemented broadcast of enactment operations, distribution of definition operations has not yet been implemented, since the definition operations themselves have not yet been fully designed or implemented; for large processes, it is envisaged that a locking or reservation scheme for process parts to be defined or evolved needs to be designed, which could also be applied to
the management of temporary developer workspaces during enactment. Since definition operations are typically of longer duration, unlike enactment, enactment would either continue on the process structure (if existing) as it was before commencement of these definition operations, or enactment would be suspended until definition is completed; in either case, modified or newly defined parts of a process should only be broadcast once they are ready for enactment (again).

Besides its name and its master’s host address, a process holds a unique key for additional identification and security (derived from the creating host’s clock time) that allows two processes that have been created with the same name on different hosts (quite a common situation) to be distinguished, requiring one of them to be renamed before the host may participate in distributed process operations (thus, currently a process name must be unique on all hosts participating in that process, since process names are mapped to unversioned filenames, unlike artifact names that use versioned filenames, as explained below); strictly, however, it would suffice to distinguish different processes with the same name by their different master addresses. Furthermore, a process holds an operation stamp that accompanies most messages and which is incremented on the master by every process operation, thereby allowing outdated process replicas or repeated messages to be identified easily. Thus, if a host goes off-line and a process replica thereby misses some broadcasts from the master, the stamp indicates whether a replica must be updated (synchronized) before it may initiate operations.

Automatic broadcast of enactment operations as described above has been implemented successfully, whereas broadcast of the less frequent definition operations has not yet been implemented, as mentioned above, but new or modified processes may meanwhile be distributed manually (via filing system). Concerning broadcast operations, there is no danger of distributed deadlocks, since all distributed operations are handled sequentially by the single master host. However, in principle it is possible that in between synchronization and execution of the enactment operation, a second enactment operation is attempted. This could lead to a temporary inconsistency, but this is recognized due to the stamp, and it causes an operation to be lost (but it may be repeated). Although as currently implemented the distributed system is well usable, it is preferable to prevent such situations from occurring, and support for non-atomic (asynchronous) long-duration operations is also desirable, therefore a locking scheme for process parts is envisaged, as mentioned above. Locking operations for different stages of process definition and enactment operations will need to be defined in intricate detail. Ideally, the locking scheme should cope with very large processes and thus operate at a smaller granularity than whole processes, in conjunction with the workspace concepts and access privileges yet to be defined in detail, in
order not to unduly restrain concurrent operations [SH97]. Transactions
could be implemented that make use of the locks to provide for a higher
level of ACID protection (atomicity, consistency, isolation, durability)
and properly abort (or postpone) conflicting operations [GR93], with
distributed operations running as transactions internally consisting of
several phases (e.g. involving synchronization and an enactment
operation), but which are executed atomically on a selected part of the
process.

Similarly, not all cases of operations involving several distributed artifacts
are implemented in the most desirable way yet, since this would require
more integrated, message-based tools. For instance, if a source code
artifact and its associated object code artifacts are intended to be placed on
different hosts in a process, both artifacts are initially created on the
source code artifact's host, and other symbol file artifacts needed for
compilation must also reside on that host, but the object code artifact’s
"final address" node attribute is set to its destination host, which causes it
to be automatically copied to that host upon confirmation, i.e. after
compilation. It would be more desirable to implement tools such as the
compiler to collect their input data respectively emit their output data via
messages, since these can be transparently forwarded to the correct
location, on different hosts, via the process. Most CHIPS-generated tools
already use automatically distributable messages, e.g. in the Oberon
timing environment example seen earlier (fig. 2.5a and fig. 4.4g), upon
confirmation a timing check is performed on the source code module
artifact by a "TimingMsgs.CheckTimingMsg," which causes a "TimingMsgs.TimingMsg" to be sent per implemented procedure to its
associated predecessor timing specification artifact to collect the specified
timing information required for the timing check, which works regardless
of the distributed or non-distributed location of the artifacts involved. But
even for CHIPS-generated tools, not all operations are based on messages
yet that are easily distributable, in particular fine-grain navigation on an
artifact’s text, token list or syntax tree, unlike the more coarse-grain
higher-level semantical operations such as timing analysis.

4.5 SYSTEM INTEGRATION LEVELS

Besides integration types discussed earlier, an additional integration issue
is how an implementation of a process support framework can be
integrated in an existing system. Different levels (degrees) of such system
integration are possible, which will first be discussed conceptually, then
briefly related to actual systems.

A local file system is always required to provide persistent storage for
processes and their artifacts, both for processes under development and
for deployed processes. However, different addressing schemes are
possible for processes and artifacts, providing different integration levels
Framework Implementation 127

of the process framework components with the host system, depending on
the degree to which the implementation fulfills the design objectives.

4.5.1 File System Mapping

On the lowest system integration level, as implemented, no modifications
are required in the host system or in tools used, as the process framework
is added to the system as a separate application. This signifies that the host
system’s file directory system provides the basis for addressing persistent
storage, and files are used for interaction with tools and with the runtime
system. This requires artifacts to be addressable independently from
processes, as individual files, and therefore to be stored separately. For
development, non-integrated or external tools typically work on files
addressable by filename and provide data integration only on the carrier
level. And even tools providing data integration on a higher level, such as
CHIPS tools, still use files addressable by filename only. For deployment,
the runtime system’s loader requires executable object code to be in a
specific file format and addressable via specific search paths by filenames
representing module names. Thus, in order not to require modifications
to the host system or tools, a mapping of artifact addresses to filenames is
required, and processes are stored in separately addressable files.

4.5.1.1 Development

Processes use Oberon System 3’s document conventions to allow them to
be opened conveniently, i.e. through identification of the filename’s
ending “.Proj” and the file’s contained generator procedure
“PlaceDocs.NewDoc.” As presented earlier, a process is stored in a file of
type document holding an object library containing the process structure’s
node objects with dependencies and metainformation, whereby every
atomic node holds its node name and two different filenames for its
artifact, its process filename and its unversioned filename.

An artifact can be of many different types, both Oberon and external
files, as defined by the artifact’s tool, and can thus not be identified by
type or filename ending. All artifacts of a process are stored in a
subdirectory of the same name as the process (without ending) in the same
directory as the process. An artifact’s process filename commences with
the number of its node’s id and continues with its unversioned (ordinary)
filename. Thus, the combination of process directory in its path and node
id in its name provides a unique name for every artifact, and the inclusion
of its unversioned filename in the name is redundant and is provided for
manual inspection purposes only. E.g. a process called “GenDist.Proj”
may have two versions of an artifact with unversioned filename “A.Mod”
associated with atomic nodes 32043 and 32029 (with node name “ASrc”),
called “GenDist/32043-A.Mod” and “GenDist/32029-A.Mod” respectively.
In order to interact with tools that cannot work on versioned artifacts or
with versioned filenames (e.g. the compiler or loader), the required
artifacts are copied back and forth between the local working directory and the process' directory (similar to the checkin/checkout model often used in configuration management [CW98]), using the unversioned artifact filename in the working directory (e.g. "A.Mod" or "A.Obj"), which may apply to different artifacts at different times. Usage of the conventional file directory system to address artifacts in this way allows conventional tools to be used, but cannot hide filenames from developers and makes it difficult to enforce access restrictions or prevent inconsistencies caused by working on artifacts outside a process.

4.5.1.2 Deployment

The simple implemented installer retrieves artifacts in a configuration from their distributed origin locations in their process and writes them in the local working directory using their unversioned filenames (e.g. "A.Obj"), which allows them to be accessed by the runtime system's conventional loader. Thus, upon installation, the artifacts lose their process awareness and versioning and dependency information and become normal files, unlike when using a deployment system at a higher system integration level. As no user-site process base management is available yet, no deployment addresses are used and no configuration validation is performed yet, and only one version can be installed in a local working directory, overwriting previous versions, but this avoids the need to modify the runtime system's loader.

The next desirable enhancement, still on the lowest system integration level, file system mapping, would be to manage user-site process bases which are organized in the same way as development processes currently are, with installed processes (as separate files) managing their artifacts as files in their own dedicated "safe" directories (protected by appropriate file access privileges e.g. in UNIX systems) and allowing configurations to be validated, and copying artifacts to the working directory as required using their unversioned filenames for interaction with the conventional unmodified runtime system. Depending on the artifact's ability, deployed artifacts could then retain their process awareness at least in their persistent state, to support configuration validation, while in their loaded state, usage of process awareness (i.e., the address of the associated node) to support loading of appropriate dependent versioned artifacts would require modifications to the loader.

4.5.2 Virtual File System

A virtual file system implements a layer of functionality on top of the file directory system providing transparent mapping between versioned and unversioned filenames, while presenting the conventional file system's unmodified interface to tools and applications. Thus, it can hide artifact versions from tools that cannot work with versions, providing unversioned filenames to tools for artifacts that are effectively versioned,
but thereby allowing access only to one version per artifact at a time, as specified by rules e.g. in the process. Virtual file systems are used by various configuration management systems, particularly UNIX-based ones [Leb94][FKR94].

A virtual file system thus would provide a higher system integration level than direct file system mapping, since it would eliminate the need to rename or copy artifacts in between operations, thus removing some potential sources of inconsistency. As it could be used to allow conventional tools and runtime system loaders to be employed that cannot handle versioned artifacts, by mapping different filenames to an artifact or different artifacts to a filename as needed, tools and runtime systems could be used unmodified, and all modifications would be concentrated on the transparent insertion of the virtual file system in the system’s conventional file directory system.

4.5.3 Process Integration

On the highest system integration level, support for processes would be directly integrated in the system on top of the file system, but replacing the file directory system. Unlike most systems, Oberon already provides a clear separation of file (storage) and directory (addressing) concepts, which would be beneficial for an implementation in Oberon, since a file system is always required for persistent storage of processes and artifacts, whereas addressing of files would be performed directly by the process management system. Besides, Oberon systems (V4 and System 3) already provide an impressive demonstration of the feasibility and benefits of integrating concepts usually managed separately in conventional operating systems [WG92], e.g. texts. With process integration, versioned artifacts would be accessed directly from the process avoiding the detour of using filenames and directories and exposing files to outside tools. Thus, the file system would be split up into two parts, one being managed by the development and/or deployment system holding on-site process bases containing software under development and containing deployed process aware software, and the other part being managed by the conventional file directory system holding e.g. old non process aware software and data files not related to software development (unless in future information processes are introduced for such data files).

System integration of processes however would require all involved tools and the runtime system’s loader to be modified to access artifacts not via file directory but via process and dependencies, which would constitute a major paradigm shift in the design of most software [Kuh96]. As explained in the section on the runtime system’s design in the previous chapter, the loader must search for appropriate object code artifacts by following dependencies and links instead of searching within the file directory, leading to an initially much more restricted search that allows
appropriate versions to be determined without the loader actually having to be aware of versioning concepts. This allows correct dependent versions to be retrieved using simple unversioned filenames only while avoiding difficulties of managing different versions in the same filing system.

Similarly, tools must be modified to access artifacts via the process instead of via filing system. An artifact is selected for editing by specifying its process node, via the process user interface, instead of selecting it from a file directory listing. Tools that access several artifacts, such as the compiler that may require symbol files besides source code, must search for these by following dependencies, similarly to the loader. Artifacts are created, accessed, copied and deleted via appropriate process browsing and editing operations instead of via filing system operations, since an artifact is always associated with its atomic process node. Although this may appear to be restrictive, replacement of hierarchically organized file directory trees by process addressing and linking concepts could prove interesting, comparable to the introduction of liberal hypertext concepts via WWW that have made Internet navigation much more attractive.

### 4.5.4 Actual System Integration Possibilities

Referring to the different conceptual levels of the process framework’s system integration presented above, different possibilities are conceivable for using existing systems, technologies and standards in the implementation, i.e. implementing different levels of system integration in actual systems. As indicated earlier, Oberon provides numerous advantages, although similar implementations would be possible based on most other operating systems, with some restrictions and perhaps increased efforts. These would also only provide system integration on the lowest level, file system mapping, whereas Oberon is probably best suited for modifications to provide system integration at the process integration level, if system modifications are acceptable. An interesting alternative would be to use a distributed objects system such as CORBA that provides relative language and platform independence with objects on a higher abstraction level than the filing system, allowing a system to be built to a large degree on the process integration level, although not completely hiding the underlying filing system. If restriction to a single language and system remains an option, Java would be an interesting alternative besides Oberon, since it also provides relative platform-independence, although system integration higher than file system mapping could be difficult, if the Java runtime system is to be used unmodified, unless it is enhanced with CORBA distributed objects, which could be an appealing combination. The section on commercial opportunities in the next chapter discusses actual implementation possibilities from the point of view of their commercial potential. An orthogonal issue is WWW integration, as discussed in a subsequent section, which aims at simplifying distributed
access to processes and the process framework and should work at any system integration level.

4.6 GRAPHICAL USER INTERFACE

The graphical user interface (GUI) of the actual prototype is presented in this section, although only its main parts and without going into details, as I was only partially involved in its design. Also, while not all features discussed have been implemented, the graphical user interface cannot yet fully display even the implemented ones, e.g. not all dependencies are shown in the prototype’s Oberon and 3D views (and inserting text in 3D graphics and drawing dependencies as arrows instead of lines is inherently complex in the employed Apple QuickDraw 3D format), therefore other sections in this thesis mostly depict manually drawn processes for more complete illustration. However, mainly for testing purposes, all implemented features can be visualized textually (e.g. where graphics are not available), e.g. commands are available to textually list all dependencies of a node in the “Oberon.Log;” similarly, configurations available in a process, or the nodes contained within a specified configuration may also be listed textually.

Besides those operations supported directly via the graphical process representation, the other most frequent operations are available as buttons in displayed panels, while many more operations are available as Oberon commands. Fig. 4.9 shows the main user interface panels that provide process operations: The “Place.Panel” (top left in figure) is used for process browsing, definition, and enactment, and contains amongst others nine buttons for definition (process, node, and dependency creation and deletion) (top left in panel), four buttons for enactment (state change and rollback) (bottom left in panel), and four buttons for navigation across history layers (bottom right in panel). The “Configurations.Panel” (bottom left in figure) allows configurations in processes to be listed, inspected, defined, modified, and removed, amongst others. Oberon System 3’s Gadgets user interface’s standard “Inspector” (right in figure) is used to edit node attributes (in System 3 terms, a node object is the “model” of a displayed node).

A new process may be created and defined using the buttons on the “Place.Panel” to open a new graphical process view, to insert nodes and dependencies, and to move them to the desired graphical positions. Alternatively, a new process may initially be specified in a textual format in the CHIPS-generated “PlacesEnv” environment, which allows the text to be parsed and, if successful, translated to the DIPS process format and displayed graphically in a “PlaceViews” view, from where it can be stored in the DIPS process format. Thereafter, it may be continued to be edited and enacted normally using the “Place.Panel,” e.g. to specify additional node attributes such as tools to be used. Translation of a process back to
As indicated earlier, Oberon System 3 document conventions are used to store processes, therefore the "Desktops.OpenDoc" command is used to open processes. When a new process is created, the generator command is also required as a parameter (or a button on the "Place.Panel" creates a new initially untitled process), or when a process is retrieved from a remote site for the first time, its address is also required, e.g. "Desktops.OpenDoc Tool.Proj (PlaceDocs.NewDoc) "mac-1942.ethz.ch"" initially opens the "Tool" sample process. Other distributed operations are available as specific commands, e.g. declaring a process copy as the master copy and allowing other developers on other hosts to access it, or opening a replicated process copy in a "local" mode without synchronization with the master, if its remote master host cannot be reached.
COMPOUND PLACE
  STRING Name 'Tool'
  INT ID 0 INT X 30 INT Y 90
BEGIN
PLACE
  STRING Name 'Spec'
  INT ID 1 INT X 70 INT Y 30
  STRING icon 'Icons.Note'
COMPOUND PLACE
  STRING Name 'Impl'
  INT ID 2 INT X 40 INT Y 120
BEGIN
PLACE
  STRING Name 'Src'
  INT ID 3 INT X 50 INT Y 70
  STRING defName 'Test.Mod'
PLACE
  STRING Name 'Obj'
  INT ID 4 INT X 60 INT Y 180
  STRING defName ''
  STRING icon 'Icons.Tool'
  STRING com 'Builder.Compile'
  STRING view 'Browser.ShowDef'
END
EDGE 1 2
EDGE 1 3
EDGE 3 4
PLACE
  STRING Name 'Test'
  INT ID 5 INT X 170 INT Y 130
EDGE 4 6
EDGE 2 6
PLACE
  STRING Name 'Res'
  INT ID 6 INT X 80 INT Y 220
EDGE 1 5
EDGE 5 6
END

Fig. 4.10: Alternative textual specification of the “Tool” sample process

Figs. 4.11a-e show the different 2-dimensional graphical Oberon process views available for that process after opening. One 2D view shows the contents of one compound node on one history (or the current) layer, with the contents of further hierarchically contained compound nodes (depicted as larger icons) available as separate views. Fig. 4.11a shows the hierarchically top compound node on the current layer. Contents of a compound node are only visualized on one level of the containment hierarchy per view, i.e. if a compound node (“Tool”) contains another compound node (“Impl”), the latter one’s contents (“Impl”) are visualized in a separate view. Thus, fig. 4.11b shows the contents of the “Impl” compound node as obtained after clicking on its icon in fig. 4.11a. The neighboring nodes are then depicted as smaller icons, and clicking on
“Tool” returns to the enclosing compound node of that name. Multiple views of a process may be open simultaneously, additional views being opened using the “View *” button on the “Place.Panel.” Figs. 4.11c-d show the top compound node respectively the “Impl” compound node on history layer 2 (the displayed history layer is written to the “Oberon.Log”), and fig. 4.11e shows history layer 1, consisting only of the “Impl” compound node.

Figs. 4.11a-e: Five 2D views of Tool sample process as shown on the DIPS prototype.
Although the number of displayed nodes within a compound node is not a priori limited, a large number of nodes would require the display to be scrolled. For clarity, it is preferable to define processes in such a way that this number does not normally exceed ca. a dozen nodes, and to make use of the containment hierarchy to structure large processes. Thus, every compound node should contain only a few nodes per level of the containment hierarchy, but these could again include compound nodes, whose contents are then displayed in separate views.

Clicking on a displayed atomic node icon opens up that node’s artifact in an appropriate editable or non-editable view, i.e. as an Oberon text, Oberon document, Oberon module interface browser output (for object code artifacts), Oberon document with an additional pull-down menu featuring specific CHIPS tool commands (for artifacts edited with CHIPS-generated tools), or external (non-Oberon) document. Fig. 4.12 shows an artifact’s editable view of the sample Oberon module in the Oberon timing process seen earlier (figs. 2.5a–c), depicting the tool-specific pull-down menu holding the commands allowing the artifact to be parsed, and if successful, the timing check to be performed that is required for its confirmation.

Fig. 4.12: Editable artifact’s view showing tool-specific pull-down menu
Fig. 4.13: Excerpt of a generated 3D snapshot in Apple QuickDraw 3D
External documents are opened on the Macintosh using the “Finder.Open” command, which sends an AppleEvent to the document (file) to open itself, as if the file had been double-clicked in the Macintosh Finder, whereupon the Macintosh operating system takes care of opening it in the appropriate application (e.g. word processor, graphical editor, etc.). External tool integration is thus currently limited to tool invocation on the Macintosh (external artifacts may be created via templates), with the developer being responsible for storing the artifact in Oberon’s working directory (since this cannot be enforced). The artifact is then however copied from there into the process directory upon the next state change of its node, retaining Macintosh-specific file parts (resource and data forks) and attributes (e.g. type and creator), as provided by module HFS; on UNIX machines, the Macintosh-specific file information is kept in separate files.

Besides the 2-dimensional graphical Oberon process views, 3-dimensional graphical process views may be generated on demand in Apple QuickDraw 3D format, a nested textual format, depicting a 3D snapshot of the process structure as it is at that instant. One 3D view is generated per compound node, showing its contents on all history layers (3rd dimension), but as in the 2D views, not showing contents of further hierarchically contained compound nodes. The generated text, describing a 3D figure (fig. 4.13), may be opened using a suitable application such as Apple’s Simple3DViewer or a corresponding Web browser plugin, whereupon a 3D view is displayed that may be turned, panned, and zoomed. Figs. 4.14a-b depict displays of the 3D views generated for the sample process “Tool”: fig. 4.14a shows its top compound node, available on the current layer and on one history layer underneath (with compound nodes “Impl” each being shown as larger figures), visualizing the text shown in fig. 4.13, and fig. 4.14b shows compound node “Impl,” available on the current layer and on two history layers.
4.7 WWW Integration

A demonstration of WWW integration is provided by the Web server implemented in Oberon [GIPSY98]. It allows processes to be viewed with simply a Web browser, and optionally a plugin for Apple QuickDraw 3D graphics [App98], but requires no parts of the process framework for browsing. Therefore, it allows developers who do not have the process framework to investigate processes, and it allows users to browse processes in order to make informed download and installation decisions without requiring the process framework. For this purpose, it could be integrated in or linked from a WWW software component catalog. Once deployment services are available to users to enable direct software downloading and installation, it could serve straightforwardly as user interface for these services, allowing them to be implemented more simply. On the long run, it could replace the elaborate user interface for developers too, although these issues are more intricate, since development requires both read and write accesses to processes and involves complex operations, whereas deployment requires only read accesses to non-local processes and uses relatively simple operations. Also, access privileges are not yet managed; typically, only certain parts of processes would be published.

In order to be able to serve current and timely information on the Web, and also to avoid requiring a large database of Web pages, the Web server dynamically creates Web pages on demand (text and graphics), per HTTP request, although it also supports recursively writing selected or all pages of a process or configuration for static serving or archiving purposes. In order to allow the Web page algorithms used for dynamic serving to be used unmodified also for generation of static Web pages, filename lengths have been limited according to the host filing system's requirements: for dynamic serving alone, the filename part of URLs could be longer than permitted in the implementation. Dynamic generation of Web pages for HTTP requests is relatively instantaneous (fractions of a second), both for HTML pages containing numerous hyperlinks and for 3D images, only generation of 2D images has a slightly noticeable delay of a few seconds, due to the runtime of the GIF generation algorithm used for 2D images. WWW page generation is currently limited in some cases to processes and artifacts residing on their master host. The WWW representation was first presented in [SMW96], although only with manually created pages at that time, since the dynamic WWW server was implemented later.

4.7.1 WWW Representation

The WWW representation provides a separate HTML page for every process node in a process. The page contains hyperlinks to other process nodes' HTML pages to represent dependencies that make up the DAG process structure.
Fig. 4.15: Web page of an atomic node showing hyperlinks representing dependencies and Oberon source code artifact representation containing hyperlinks (cursor on A.AMsg illustrating hyperlink, bottom line)

An atomic process node displays its artifact's contents in an appropriate way within its page, e.g. Oberon source code as text that may contain appropriate hyperlinks to versioned artifacts that are imported modules (fig. 4.15). A compound process node contains within its page both a 2-
dimensional GIF graphic showing its contained nodes with their dependencies in its history layer and a 3-dimensional graphic in Apple QuickDraw 3D format additionally illustrating process history in the 3rd dimension (artifact revisions) (fig. 4.16 as generated, fig. 4.17 as displayed).

Fig. 4.16: Web page of a compound node, as generated
Fig. 4.17: Web page of the compound node of fig. 4.16, containing 2D image of one process layer (left) and 3D image of the whole process structure (right), with clickable nodes in both images (cursor on TestLog node illustrating hyperlink, bottom line)
Fig. 4.18: Web page of a compound node in a specific configuration, containing 2D image of one process layer (left) highlighting the two nodes of the selected configuration on that layer, and 3D image of the whole process structure (right) highlighting the four nodes of the selected configuration in the process structure (cursor on older BSrc node illustrating hyperlink, bottom line)

Similar to the nodes’ HTML pages, both 2D and 3D graphics display nodes containing clickable hyperlinks representing dependencies that lead to the respective nodes’ HTML pages, thus allowing for straightforward
navigation. For 2D graphics, both server-side (<IMG ISMAP> HTML tag) and, for newer browsers, client-side (<IMG USEMAP> and <MAP> tags) image mapping are supported, to provide the displayed nodes' hyperlinks, while for 3D graphics, the hyperlinks are contained directly in the graphics' node descriptions. A 3D graphic can be turned, panned and zoomed within the page, as supported by the required browser plugin for Apple QuickDraw 3D. Nodes are colored according to their state and may display additional node attributes. The top compound node additionally lists available configurations in a process. Thus, where available, configurations may also be selected, in which case a compound node's representation highlights those nodes that belong to the selected configuration (fig. 4.18). In future, more elaborate WWW representations are envisageable, particularly once browsers are available that progress beyond the conventional page-based user interface model [Dar98][Tai97].

4.7.2 Implementation

The implementation consists mainly of the two modules HTTPServer and WriteHTML, plus several utility modules (fig. 4.19). HTTPServer implements a simple Web server, and WriteHTML coordinates writing of all Web pages, text and graphics, and also writes HTML pages. Since WriteHTML handles most types of process data, directly or by invoking appropriate functions, it uses a large number of process-related modules and even some CHIPS modules for CHIPS-based artifacts (and WriteHTML touched the limits of possible numbers of imported modules allowed by the employed Oberon compilers, therefore some functionality had to be moved to a separate module, named WriteHTML0).

HTTPServer processes HTTP 1.0 [WWW98b] requests from clients (GET and HEAD methods), based on TCP/IP. It extracts the URL from the HTTP request and forwards it to WriteHTML's main procedure, which parses the URL, creates the appropriate text or graphic, and returns it in an anonymous file together with its MIME type to the server, which sends

![Fig. 4.19: Simplified WWW server component and module hierarchy](image-url)
both to the client as appropriate. If necessary, HTTPServer also provides server-side image mapping (identified by "/cgi-bin/imagemap/" and "?" in a URL) by translating the URL to another one: first, the imagemap’s URL is passed to WriteHTML to obtain the imagemap (text), then the server parses the imagemap to find the matching URL, which is again passed to WriteHTML to receive the appropriate page.

WriteHTML parses a URL and creates and returns the appropriate data (as HTML 3.2 text [WWW98c], other text, or graphic). Currently, only processes and artifacts residing on the HTTP server’s host are accessible. A URL specifies the required process on that host and the (atomic or compound) node in that process, in a specific configuration or not, or it may specify an atomic node’s artifact, or a compound node’s 2D GIF graphic, 3D Apple QuickDraw graphic, or imagemap. Other URLs that are handled include a list of all available processes, a list of all Oberon System 3 module definitions, and all Oberon System 3 module definitions. Therefore, MIME types returned by WriteHTML include HTML text, plain text, GIF image, 3D image, and others. Although users need not be aware of the structure of URLs when clicking on hyperlinks (the server “http://129.132.57.174/" provides a list of processes with all required hyperlinks), the structure used is very similar to node addresses, e.g. “http://129.132.57.174/GenDist.Proj.html” refers to the top current node in the process, “http://129.132.57.174/GenDist.32043.ASrc.html” refers to node 32043 (called ASrc) of process GenDist on the specified host, and “http://129.132.57.174/GenDist.32051-Vlmod.html” displays node 32051 while highlighting a configuration called “Vlmod.”

To write HTML text for an atomic or compound process node, the process is loaded locally (if not yet loaded) and the node’s appropriate metainformation is accessed. All the node’s dependencies are translated to HTML hyperlinks pointing to the neighbor nodes’ HTML pages. Additionally, for an atomic node, its artifact’s type is determined, and if appropriate and available, the artifact is loaded. Most textual artifacts are translated to HTML that is inserted into the atomic node’s page, whereas for others, only a hyperlink to the artifact may be provided. For object code or symbol file artifacts, the module interface (text) is first provided by a module interface browser called BrowserT (similar to the system’s Browser, except that the result is written into a text instead of on screen), which is then processed similarly to source code artifacts. An “OberonMsgs.WriteHTMLMsg” is sent to these, which causes them to be handled by a CHIPS-generated Oberon environment (compiler front-end tool), consisting of Oberon scanner, parser (for both module definitions and implementations), declaration analysis, and HTML generator components. The tool outputs HTML that contains a hyperlink for every imported identifier that points to the appropriate place where the identifier is declared. In particular, the artifact being handled determines
its location in the process context so that the hyperlink points into the correct version of the referenced artifact in the process, e.g. “http://129.132.57.174/GenDist.32043.ASrc.html#AMsg” refers to a different definition of “A.AMsg” than “http://129.132.57.174/GenDist.32029.ASrc.html#AMsg” does. Only if it is not available in a process, an unversioned artifact may be referenced, such as a system module interface text, e.g. “http://129.132.57.174/Objects.html” refers to module Objects.

Compound node HTML pages contain graphics in 2D (<IMG> tag) as GIF, with associated client imagemap (<MAP> tag), and 3D (<EMBED> tag for the plugin) as Apple QuickDraw 3D. Since standard Oberon systems (V4 and System 3) do not provide a port system for the display (although other Oberon systems do [Szy92][Obe98]), this means that display functions usually have to be programmed separately for every device (screen, printer, etc.) instead of being implemented once each in a generic way. Thus, in order to create 2D GIF images of processes, this would require an additional major implementation of process display functions similar to the screen display implementation, but which would write to an off-screen memory area (printing has not been implemented). To avoid this, a GIF is generated by displaying processes on-screen normally as provided by the functions in module PlaceFrames without requiring modifications of these, then the display data is read from the video memory and translated to a GIF image by module GIF, using an adaptation of a published GIF generation algorithm for interlaced GIF 87a images [Gra98]. This also explains why Macintosh-based Oberon instead of UNIX-based Oberon was chosen for our Web server operation, since it is neither practical to access video memory on a multi-user (UNIX) machine, nor is it easy with the X11 display libraries employed in SPARC-Oberon (although the Web server has been implemented for both platforms). Generation of 3D images is comparatively simpler, since such an image is written directly to a file and not displayed by Oberon but by an Apple QuickDraw 3D browser plugin or tool. The implementation of 3D image generation is provided by module SnapShot and for use by the dynamic Web server essentially required only addition of hyperlink generation for nodes.
Leer - Vide - Empty
Chapter 5
Discussion

5.1 EXPERIENCES
The implemented experimental prototype allows the presented concepts to be applied, tested and evolved. Limited functionality, as described in the implementation chapter, as well as a limited user interface, have prevented the prototype from being used on a broader basis yet or from being fully applied to its own development process, but it already serves as a viable demonstration of the concepts and their potential. We have used the prototype for defining and enacting reasonably realistic small and medium-sized processes, with distributed processes running on up to four heterogeneous workstations simultaneously, including both Macintosh and UNIX platforms, and both LAN and WAN operation have been demonstrated successfully (e.g. with a distributed process running simultaneously at ETH Zurich and EPFL Lausanne sites).

Processes defined initially contained up to ca. 30 nodes, and after several rollback operations attained up to ca. 200 nodes and ca. 15 history layers. One of the processes is for the CHIPS kernel, thus applying the project to a part of itself in a bootstrapping manner, with configurations that allow the CHIPS kernel to be retrieved from a remote source and installed in a single operation. Other processes include a student exercise to create a scanner and parser for a given grammar, making use of the CHIPS tool generator, a process using the timing specification and Oberon timing analysis environments (tools) to demonstrate development of real-time software with timing constraints being checked automatically during development, and numerous other small processes to test specific features e.g. of the process modeling language or its user interface, such as hierarchy, rollback, configurations, etc., or to make use of specific CHIPS-generated tools; some processes are also available on the WWW in the dynamic WWW server demonstration [GIPSY98]. These processes show that the concepts are viable and provide the anticipated benefits to developers and users.

Experiences gained with such processes have resulted in an evolutionary design of both the process modeling language and the distributed
framework architecture up to the present form, and have also uncovered numerous open questions and issues for further research, as presented in various chapters of this thesis. While a presentation of the complete design path with every individual design decision taken during this evolutionary process is beyond the scope of this thesis, it aims instead at presenting the results in a coherent form, thus leading to the concepts discussed throughout this thesis.

Most Oberon development and operational work has been carried out on early-generation Power Macintosh computers running 80MHz PowerPC processors and connected via Ethernet LAN, and the Web server is also running on such a machine. Considering average process sizes of well under 100KB, and maximum typical processes of up to 500 nodes resulting in process sizes of a few 100KB at most (excluding artifacts, which are stored separately), performance limitations have not yet been an issue on these machines. Even for the more complex algorithms (e.g. rollback or serialization), runtime is dominated by operating system issues such as disk access, network latency, or on-screen drawing of complex graphics, and operations typically occur in hardly-noticeable fractions of a second. More complex distributed operations such as synchronization may sometimes take a few seconds, if a large process is to be synchronized (transmitted) that involves disk accesses and message sizes of a few 100KB each. Only in rare cases where a distributed operation cannot be serviced immediately because one of the involved machines is blocked by another operation, longer delays may occur. Generation of dynamic Web pages (HTML and 3D) also in most cases occurs in relatively instantaneous fractions of a second, the only notable exception being generation of 2D GIF images, as the employed GIF generation algorithm usually has a runtime of a few seconds, which may incur a slightly noticeable delay, if a fast network connection is used.

5.2 Examples

This section presents examples illustrating all features of the process framework employed in the complete lifecycle of software components, highlighting integration benefits of using a single system, in addition to specific features already illustrated in examples in earlier chapters. Although not all features have been fully implemented yet, this section should give an idea of how the process concepts may be used in actual operation and may be regarded as a concluding summary and feature overview. The order of operations carried out may vary greatly, depending on the processes involved. E.g., a new process may actually commence with deployment operations, e.g. for process framework components and the tools to be used in the process, whereas deployment of the new process itself only occurs after its definition and enactment.
Similarly, a process obviously cannot be browsed before it has been
defined, although other processes will typically be browsed before.

5.2.1 Process Browsing

Process browsing is the most basic process operation and is often used in
conjunction with other process operations. It involves opening different
views of processes (in 2D and 3D), artifacts, and node attributes (as
described earlier in the user interface section in the previous chapter),
within the process framework or on the WWW, e.g. by clicking on
displayed nodes, operating buttons on user interface panels, or executing
appropriate commands, in order to navigate among process structures,
along dependencies, and to view artifacts. Nodes of linked processes (not
yet implemented) may be displayed using different icons from local
nodes, with their full views also being opened upon clicking on them. As
seen before, all processes and nodes are identified in a straightforward
URL-like manner. Access to specific processes or nodes and artifacts may
be restricted to certain developers only, with other developers and users
not being permitted to access them.

5.2.2 Process Definition

A new process is created and defined using commands and the process
framework’s graphical user interface (GUI), as seen earlier. Process
name, location (primary and backup hosts), and developers are specified,
process nodes are created and dependencies are inserted among them, as
restricted by the language’s rules and previously inserted dependencies.
Nodes for artifacts making up different parts of the evolving product and
representing different process steps as well as dependencies are defined
following e.g. a simple abstract process model such as the waterfall
model, where one or more nodes each could represent contract,
specification, design, implementation, validation, testing, etc. Fig. 5.1
illustrates (small but realistic) sample process definitions. Subsequent
possibilities for process iteration also allow iterative abstract process
models to be applied. Dependencies (links) may also be established on
nodes in other processes, which remain unmodified. Node attributes such
as artifact name, tool, location (in a distributed setting), responsible
developer, access privileges, and others are assigned as known and
required, which may be delayed until the node is to be enacted, since
process parts may be edited even after previous ones have already been
enacted.

Links on metaprocess levels (not illustrated here, since not yet
implemented) allow tools and the framework itself to be specified in a
process (several examples have been given in the chapter on the process
modeling language). Tools (and the framework) are specified as
configurations or configuration sets (sets of configurations in other
processes, where the tool components have been created) that ideally
specify a precise version of a tool and thereby define the formal conditions (if any) for the associated nodes' dependencies. Alternatively, lower integration of tools is also possible, where a configuration set only specifies a general family of similar tools as variants, thus allowing a choice of different tools to be used, or tools that are not (yet) available in a process (e.g. in order to use existing tools, or for bootstrapping purposes). Existing tools may be specified by incorporating them in existing general configuration sets or by defining new configuration sets for them. In the current implementation, tools are specified textually only, as commands with parameters, by defining the node attributes for creation function and required predicate, as (metalevel) linking to the tools' processes (to additionally specify tool origins and versions besides tool commands) is not yet supported.

Defining a process not only utilizes the process framework, but also involves configuring the framework by adding tools to provide a complete process-specific environment. Thus, as mentioned above, tool specification and utilization typically already involves deployment issues, even before the process that uses the tools is defined, as these tools are installed on the developer's workstation or nearby server.

![Diagram showing definitions and linked processes](image)

**Fig. 5.1: Definitions of “Tool” sample process (left, complete) and linked “Server” and “Talk” sample processes (right, excerpts of topmost hierarchy only)**

### 5.2.3 Process Enactment

A defined process may be enacted immediately by the process framework by interpretation, there is no intermediate coding, instantiation, or compilation. Enactment is only possible for fully defined nodes, and is typically restricted to a few developers. Only one developer may enact a node at a time, although this privilege may be transferred automatically to another developer within a specified team. Enactment involves executing the tool specified with an atomic node, which in the case of manual tools (e.g. editors) creates a new artifact that is empty or partially complete (e.g. from a template), or in the case of automatic tools (e.g. compilers) creates a completed artifact. For manual tools, the artifact is then
completed by the developer as necessary, which obviously constitutes the main effort in the process and involves creative work. Thereafter, the artifact is manually or automatically confirmed. The node thus traverses the three defined states “planned,” “editable” and “confirmed,” and the aim of the process is that all nodes on the current layer attain state “confirmed” (illustrated below in fig. 5.3, together with rollback).

It is envisaged that for editing, compilation, testing, etc., developers work using their own small local development systems, separate from the process, or, preferably, they work in their own protected workspace within the process, in order to hide minute day-to-day modifications of artifacts under development from other developers, with the process acting primarily as check-in/check-out repository managing and coordinating coarse-grain modifications, e.g. upon node state changes.

5.2.4 Variants

When defining a process or a part of a process, concurrent nodes may be specifically designated as variants, i.e. alternatives (not yet implemented). A set of alternatives, e.g. called “platform,” and its variants, e.g. “Mac” and “SPARC,” are registered in the process as process-wide names, and these can then be used to specify all nodes in the variants. Variants have different enactment rules, e.g. confirmation of dependent nodes occurs per variant, thus preceding nodes involving other variants only need not be confirmed beforehand. Likewise, configuration and deployment occur per variant, i.e. a variant may be a specific feature of a configuration (with different alternatives from the same set excluded from the same configuration).

As a realistic example of using variants in our own development (although not yet supported by the process framework), multiple variants were developed for module TCP, besides multiple revisions (whereas a common version of module HTTP could alternatively use any TCP variant, fig. 5.2). TCP was implemented for four platforms: 68K Macintosh, Power Macintosh, SPARC SunOS 4, and SPARC Solaris. Furthermore, it was developed for both Oberon V4 and Oberon System 3.
on each of the four platforms, which fortunately did not need to be distinguished in module TCP, otherwise eight variants would have been required. Also, it was compatible with multiple revisions of each of the individual Oberon systems, thus avoiding an increase in the number of required versions. However, its own development progressed in several revisions, thus finally resulting nevertheless in numerous versions (if all versions had been managed in a single DIPS-supported process, this would have resulted in a medium-sized process of several hundred nodes).

5.2.5 Process Iteration
In order to allow artifacts to be revised, and also to support iterative process models, process iteration or rollback operations may be performed on process nodes, which records the previous results in the process history.

Fig. 5.3: Result of a sequence of enactment and rollback operations performed on the “Tool” sample process

Nodes (respectively their artifacts) may be selected for revision, upon which these and all of their dependent (successor enacted) nodes are transferred to the process history, creating a new history layer, and new
empty process nodes with respective dependencies are created in the current process layer. There, new artifacts can be created which may also use parts of the previous artifacts or even be completely identical, but which must in any case be confirmed again as enactment proceeds (fig. 5.3).

5.2.6 Process Evolution

Even after enactment, processes may be modified. New dependencies and nodes may be appended to existing ones anywhere, both in the current layer and in the process history, either as separate nodes or as related variants. In contrast to appending, insertion of new dependencies and nodes in between existing ones is more restricted, since existing nodes may become dependents of new ones: this may only occur where the dependents have not yet been enacted. If they have already been enacted, a rollback must first occur in order to create unenacted nodes. In either case, both the previous and the new structure are recorded in process history. E.g., the “TCP” implementation first consisted of three modules which were later combined into a single one (fig. 5.4).

Since process evolution operations preserving the process history have not yet been implemented in the prototype (except for appending nodes and dependencies that do not affect existing enacted nodes), a possibility meanwhile is to create a new process copy for every evolution operation. Thus, the process structure is copied (giving the copy a new name) and a “PlaceUtils.Reset” operation is performed on the copy, which removes all history layers, leaving only the current one, resets all nodes to state planned, and removes all artifacts. Thereby, the copy is reset to the state before enactment, as if the process had just been newly created, allowing it to be modified through the usual process definition operations (distributed copies must be updated manually, via filing system). Alternatively, if the process history need not be preserved, a process may be edited (possibly after a rollback), thus modifying the process structure,
and then distributed manually (via filing system) to hosts that already had a previous copy.

5.2.7 Configurations

Any consistent and contiguous set of confirmed (i.e., read-only) nodes in a process may be defined as a configuration (e.g. “TCP” and “Server”), according to certain rules, and a configuration may contain at most nodes of one alternative per alternative set. Configurations are registered per process, and enactment of a process may continue even after configurations have been defined, since confirmed nodes are not affected by enactment of other nodes (even if a confirmed node is transferred to the process history due to a rollback, its node identity and possible configuration are not affected). It is envisaged that configurations may include nodes in multiple processes (fig. 5.5).

Possible configurations involving more than one node (assuming configurations may involve multiple processes); configuration: "name": ("<process-name>-<node-id>"):

- "Server Old": (Server-1, Server-2)
- "Server New": (Server-3, Server-4)
- "Talk Old": (Server-1, Talk-5)
- "Talk New": (Server-3, Talk-6)
- "Server & Talk Old": (Server-1, Server-2, Talk-5)
- "Server & Talk New": (Server-3, Server-4, Talk-6)

Fig. 5.5: Possible “Server” and “Talk” configurations

Configuration sets are similarly defined as sets of configurations, but are registered independently from any process (not yet implemented), allowing configurations in otherwise unrelated processes to be grouped together (e.g. if “TCP,” “Server,” and “Talk” were defined in three separate processes). Configuration sets also prevent configurations from becoming too numerous when potentially large numbers of different combinations are possible in configurations, but few combinations are actually used.

5.2.8 Release

Release only concerns configurations and configuration sets, i.e. confirmed (read-only) nodes defined as consistent sets to be used together.
Release involves defining access privileges of selected configurations and configuration sets to allow access beyond the immediately involved developers. Access privileges could be categorized for developers in the team, developers in the same company, licensed external developers, other external developers, licensed users, other users, etc., and release generally involves reducing restrictions to allow access by a more general access circle. It may also include improving availability of the configurations e.g. by copying them to release servers that are better suited to cope with expected download traffic.

Release may also involve marketing measures such as pricing, announcement of new availability of configurations, insertion in published electronic software catalogs on the WWW, etc., but these issues are beyond the scope of this thesis.

5.2.9 Retrieval and Installation

Retrieval (respectively delivery) and installation are deployment activities. A user retrieves published (released) configurations from a source (original or, preferably, copy) where he/she has corresponding access privileges (obtained e.g. through payment or license) via the deployment parts of the process framework (possibly integrated in the WWW). Typically, via WWW a link from an electronic software catalog is followed, and the required configurations are selected for downloading, although retrieval may also be delayed up to the moment the required software is invoked.

![Diagram of possible installations of "Server," "Talk," and other products at a user site](image)

Fig. 5.6: Possible installations of "Server," "Talk," and other products at a user site

Compatibility of the selected configurations with those already in the user's process base is checked where possible, and if in order, they are downloaded via network and inserted in the user's process base. Configurations are downloaded as complete parts of processes, i.e. including all nodes, dependencies, artifacts that make up the software, and other artifacts in the configurations, thus installed software retains its
process awareness that allows configurations to be validated and incompatibilities to be recognized, unlike conventional files. Appropriate dependencies are established on shared configurations already existing in the process base, which thus forms the user’s individual “puzzle” of configurations holding software typically retrieved from many different distributed sources. E.g., fig. 5.6 illustrates possible installations at a user site.

5.2.10 Loading and Linking

Selecting a software component for execution typically incurs not only its loading but also loading of required components or libraries, e.g. imported modules in Oberon systems. Thus, the correct dependent components must be loaded. The loader searches for these in the local process base by following dependencies, assuring that only compatible versions are loaded, as validated upon installation, unlike in conventional filing system-based installations where dependencies are at best specified manually by search paths. E.g. in the installation of fig. 5.6, if “HTTP” is loaded, the appropriate installed “TCP” (which is imported by “HTTP”) is also loaded, amongst others. If not yet installed locally, downloading may also be incurred when the required software is invoked, upon which the configurations are validated, before the components are loaded. Upon loading, dynamic linking capabilities are assumed as provided by many modern languages and systems, although the concepts could be extended to cope with statically linked entities.

5.2.11 Removal

Utilization of space in the local process base may be managed either manually or automatically, i.e. as a cache that automatically removes less frequently used components. In both cases, for every candidate component to be removed, a check may be performed whether the component is still required by other components in the local process base, thus allowing its relative importance to be assessed, and removal of components still being depended on may be restricted. E.g. in the installation of fig. 5.6, if “Talk” is to be removed, “TCP” may not be removed unless “HTTP” is also deleted.

5.3 Related Work

5.3.1 Systems and Projects

5.3.1.1 Process-centered Software Engineering Environments

Numerous research projects covering process-centered software engineering environments have been carried out in recent years. Garg and Jazayeri [GJ96b] discuss nine such systems: Adele2, Arcadia, Articulator, MARVEL, Melmac, Matisse, Merlin, Process WEAVER, and SynerVision (running in HP’s SoftBench environment). Finkelstein et al. [FKN94] have collected descriptions of ten European projects: EPOS,
Discussion

SOCCA, MERLIN, OIKOS, ALF, ADELE-TEMPO, SPADE, PEACE, E³, and PADM; and [Lon94] compares them. Garg and Jazayeri [GJ96b] have also collected 30 previously published significant papers on the topic, including overview papers, opinion papers, and some on specific environments or process modeling languages. Ambrioli et al. [ACF97] provide a recent assessment of several environments according to specific criteria. Most environments focus on a specific selected issue, particularly on providing a process modeling language with sophisticated but specialized features.

Sharon and Bell [SB95] provide a brief overview of several commercial systems, e.g. frameworks such as Digital’s CohesionWorX [Wel91], Hewlett-Packard’s SoftBench platform [Cag90][BEM92], or IBM’s WorkBench environment, and process management tools such as Hewlett-Packard’s SynerVision. Such frameworks, as well as Digital’s CASE interface services [BEM92] or IBM’s AD/Cycle [BEM92], focus mainly on providing one specific type of integration but not the others. E.g., CASE interface services and AD/Cycle focus on data integration (tool integration) by managing a repository, SoftBench focuses on control integration by providing a messaging service, and CohesionWorX also provides some control integration through standards such as CORBA [OMG98]. Process support is usually not provided, or is at best managed by a separate non-integrated tool. Product, process, and repository issues are rarely integrated in a meaningful way, although some are starting to propose some similar approaches as we do, e.g. in [HKW97], and deployment support is not provided at all. Some visions of globally cooperating developers and software marketing have been presented in [BMS95]. Decentralized collaborative process definition and enactment via the Internet is beginning to appear [Mau98], e.g. in the Serendipity-II environment [GAHM98].

Workflow or Computer Supported Cooperative Work (CSCW) is an area partially overlapping with process support [KK94], and it features numerous commercial and research systems, although the focus is usually on distributed cooperative editing of individual documents and less on repository issues or on modeling of workflows or processes. Whereas CSCW typically focuses on concurrent editing of shared documents by several participants, respectively on subsequent merging of document versions, DIPS focuses on more coarse-grain cooperation, since a document may be edited by only one person at a time, although by several persons in sequence; merging is an orthogonal issue.

Nevertheless, in view of increased WWW integration of the DIPS system, emerging HTTP-based WWW workflow protocols being developed by the Internet Engineering Task Force (IETF) [IET98] such as the Simple Workflow Access Protocol (SWAP) and Distributed Authoring and
Versioning (WebDAV) for collaborative authoring could prove interesting in supporting some artifact or process editing aspects of distributed process operations [Rei98], and URL-like addressing of components is becoming more popular, e.g. as in [JV97].

DIPS focuses mainly on integrated distributed process support, providing built-in process integration and some organization, control and data integration, while allowing CHIPS-based tools to be used to enhance data integration, and DIPS additionally covers issues which do not seem to be addressed by other systems. The DIPS design targets support for heterogeneous artifacts, tools and platforms, essential in loosely-coupled widely distributed virtual organizations, by building on a CORBA-style distributed objects base. DIPS focuses on a much broader scope than other systems in multiple ways, by integrating product, process, and repository issues already in the process modeling language, by allowing distributed processes to be linked together, by providing framework-integrated deployment support, and by providing a common understanding of processes for developers, managers, external developers, users, as well as tools.

5.3.1.2 Configuration Management Systems

Sharon and Bell [SB95] provide a brief overview of several commercial configuration management tools such as Software Maintenance & Development Systems’ Aide-De-Camp (ADC) (meanwhile evolved to TRUE Software’s TRUEchange [TRU98]), Atria Software’s ClearCase (now by Rational Software) [Rat98], or Continuus Software’s Continuus/CM [Con98]. Such systems, as well as Tichy’s RCS system [Tic85] (and successor SCCS), are typically concerned mainly with source code configuration management for development and build tasks, but do not support configuration management, release or delivery activities of object code or other artifacts, i.e. deployment activities. Furthermore, configuration management tools are often separate tools, not integrated into an environment, that use proprietary data structures to manage versions and configurations, either in a so-called vault, a database accessible only to the configuration manager, in a repository such as a PCTE-based one [ECMA93b], requiring all tools to adhere to the repository’s interface, in a virtual file system that intercepts I/O calls and redirects them to a repository, possibly causing compatibility problems, or by using special filename conventions (e.g. extended naming) [Leb94]. WWW integration and distribution are beginning to be addressed, e.g. in [RHHT96], and the necessity of better integrating version and configuration management and process issues is now starting to be recognized [Joe97][NS97][HKW97].

The DIPS approach is different in that process and configuration management support are integrated by using the process as common data
structure. Thus, artifacts are versioned already upon creation and in principle need not leave the DIPS process environment during their whole lifecycle, since deployment occurs out of the process. Mapping of artifacts to unversioned files is only required if non-integrated development tools or runtime systems are used together with the DIPS system.

5.3.1.3 Software Deployment Systems

A recent survey of release management, delivery, installation and configuration registry systems by Alexander Wolf's research group (who is also current vice-chair of ACM SIGSOFT) finds that numerous systems exist that support selected individual deployment activities, but only the survey's authors' research system, the Software Dock, covers most deployment activities [HHHW97a].

Tivoli's TME 10 [Tiv98] is a software distribution system that allows dependencies to be managed, but assumes centralized control. AT&T's ship [Fow95] is a release management system for a proprietary reuse architecture only that allows developers to distribute software releases to user sites, but does not involve the user. Current configuration management systems, e.g. ClearCase [Rat98] or Continuus/CM [Con98], support primarily software development at the source code level, but not deployment. The FreeBSD porting system [Fre98], using "make" [Fel88], supports building and installation, but not management of dependencies separately from makefiles. Marimba's Castanet [Mar98] is a client-server content delivery system only, which does not support managing dependencies, similar to 20/20 Software's PC-Install [Twe98] or Netscape's SmartUpdate [Net98]. Different systems exist to manage configuration information locally, e.g. the Desktop Management Interface (DMI) [DMI98] or the Microsoft Registry [Mic98a], but they do not support the deployment task. In Microsoft Windows 98, there is a Version Conflict Manager [Mic98a] that installs drivers during Windows 98 setup, but occasionally in a way that can cause problems, as it sometimes intentionally replaces newer versions by older ones. Whereas installers often replace older versions by newer ones, doing the opposite is remarkable enough to be mentioned even in the non-technical media [Liv98], since it may cause newer applications depending on the newer versions not to function any more, including Microsoft applications. In the Forest project at Sun Microsystems Laboratories (SunLabs), an integrated configuration management and build system is being developed for Java [For98].

The recent Open Software Description (OSD) specification based on the eXtensible Mark-up Language (XML) and jointly submitted by Marimba and Microsoft to the World Wide Web Consortium (W3C) (Aug. 14th, 1997) [MM98] is the first open, industry standard data format that may help automate software distribution over the Internet. It aims to provide
an interoperability standard for software description and allows dependencies among multi-platform, multi-language, multi-vendor software components to be described. It will thus provide an important complementation to Marimba’s Castanet content delivery system [Mar98], although it does not yet go as far as the DIPS system to directly use process information.

Thus, except for the Software Dock’s Software Release Manager and perhaps future OSD-based systems, release management systems typically lack either adequate support for dependency management or distribution [HHHW97b][HHW96], and none directly integrate process information as DIPS does. Dependencies are typically specified manually, and even in the Software Dock’s Software Release Manager, which is probably the most complete deployment system and the one most similar to the deployment part of DIPS, dependency, version, author and retrieval address information are entered manually into a database.

While the Software Dock is apparently the only system covering deployment issues to a reasonable degree, there seems to be no system at all except our DIPS system that addresses both development and deployment issues in a single integrated system. In particular, there seems to be no other system that integrates development and deployment as DIPS does by using the process as a common information base for both tasks. Furthermore, no other system seems to extend deployment to include runtime issues as DIPS does to allow loading of process aware components directly out of user-site process bases.

5.3.2 References and Standards

5.3.2.1 ECMA Framework Reference Model

The European Computer Manufacturers Association (ECMA) has defined a reference model for frameworks of software engineering environments (SEEs), first published in 1990, and, together with the National Institute of Standards and Technology (NIST) of the US Department of Commerce, has modified it a number of times [ECMA93a][Ear90][CN92]. SEE architectures typically distinguish between a set of tools in support of specific phases of a software process, and a set of process-independent infrastructure facilities or services, denoted the SEE framework. The framework thus provides a set of commonly needed (core) facilities to simplify tool construction and to provide integration at a higher level than usually found in operating systems. The reference model is a conceptual and functional framework which allows describing and comparing systems, but does not itself provide a standard or define architectural issues for any specific implementation. It has become known as the “toaster” model, because an initial 3-dimensional graphical representation showing tool slots in the framework reminded of slices of toast in a toaster, but the figure has been misunderstood to represent the structure
of the framework’s reference model and has subsequently been removed in later editions of the ECMA report, since the reference model is only a catalog of services and does not describe any specific architecture for the framework.

The reference model defines seven groups of services: (1) object management services, (2) process management services, (3) communication services, (4) operating system services, (5) user interface services, (6) policy enforcement services, and (7) framework administration services. The reference model presents a comprehensive list of possible services in a framework and may serve as a basis to compare frameworks, and it illustrates the large number of issues which could be of interest in a framework beyond the issues discussed in this thesis. However, most SEE frameworks implement only a small number of all possible services, and in this thesis we are mainly concerned with object management and process management services. Also, the emphasis in building SEE frameworks is shifting from comprehensive full-featured yet isolated SEEs to smaller, simpler, more flexible and interoperable ones. In this sense, our DIPS framework concentrates mainly on process management services, and on some object management services, whereas some other services could be provided by a basic underlying distributed objects system.

5.3.2.2 PCTE Open Repository Standard

The Portable Common Tools Environment (PCTE) [ECMA93b][WJ93][BEM92][Ear91], initially defined by ECMA, is a specification of interfaces for SEE frameworks to be used as the basis for distributed portable environments, addressing some of the ECMA Framework Reference Model services. An abstract specification exists as well as specific language bindings, e.g. for C and Ada. By providing a tool interface for an entity-relationship-attribute model object repository, it focuses mainly on data integration through data repository and data integration services. It has been adopted in some commercial environments, e.g. SGFL’s East is a PCTE-based UNIX software engineering environment providing heterogeneous interoperability across multiple UNIX workstations, as mentioned in [SB95].

5.4 COMMERCIAL OPPORTUNITIES

In research, it is permitted to pursue visionary ideas without having to consider immediate commercial applications. Nevertheless, the long-term goal should still be to provide benefits which could be exploited commercially, e.g. the incorporation in software products of the ideas presented would provide advantages for developers, users, and companies employing the products. Therefore, possible applications of the ideas to commercial products are briefly discussed here.
The prototype implementation is an experimental proof-of-concept system for demonstration purposes, but it is necessarily incomplete. Further efforts are required to extend the limited-functionality prototype to a fully functional product. As presented in the system integration section on actual system integration possibilities in the previous chapter, various prospects exist for such implementations, fulfilling the design objectives to different degrees. Furthermore, even a partial implementation only is also envisageable, for instance to support mainly the deployment but not the development phase, by providing translation of processes in proprietary environments to the DIPS process modeling language, to support process linking and software deployment only, but not development. Nevertheless, the long-term goal should be to support both development and deployment phases in a system inspired by the DIPS project.

The popularity of WWW and other Internet protocols demonstrates impressively that simplicity, openness and interoperability with existing systems are of utmost importance for the successful adoption of a new system [Shi98b][RSWH98]. In a global setting, different organizations and developers cannot be required to modify their systems, tools, policies, practices, and existing processes to conform to a new system, therefore the new one must be as little intrusive as possible. In this sense, the lowest level of system integration, file system mapping, seems to be the most realistic level achievable in the foreseeable future, with process integration remaining a more long-term goal. Thus, no modifications are required to existing systems or tools, and existing software may still be used both alongside a new process framework and within processes. Mapping of artifacts to conventional files with unversioned filenames allows existing heterogeneous tools to be used in process steps, although their process integration is then typically limited to tool invocation, and allows unmodified runtime systems to be used to load deployed software. Process links and metaprocess links specified in the alternative textual form provide the essential openness to interoperate with any existing process or file, where desired, regardless of the process modeling language or system used. The relative simplicity of the dependency, hierarchy, history dimension, and configuration concepts of the process modeling language increase the chance for a potential wide adoption of such a system. Also, no specific process definition or configuration management policies are enforced, allowing any type of abstract process model to be applied, and allowing any programming language to be used for development. In particular, versioning concepts refrain from requiring explicit version numbering, and version management remains clearly separated from programming language concepts, according to conventional practice. Thus, an implementation on a low system integration level should have realistic chances of adoption from a
conceptual point of view, provided the implementation is itself based either on systems whose source code is freely available, such as Oberon or Linux, or on a widely portable and interoperable environment such as CORBA distributed objects. Also, it is more realistic to assume that a new system will be implemented on top of an existing filing or object system rather than replace it.

With such an implementation, benefits for users include a simpler acquisition, installation, and removal of software. Benefits for developers additionally include an improved management of software under development, concerning versioning, interoperability with dependent software, management of heterogeneous artifacts and tools, integration of formal methods in the development process, and management of the process. For companies, these benefits could translate to reduced development respectively maintenance costs. Benefits for software development companies additionally include simplified distribution, marketing and support of software, as processes could be presented on the WWW and also be accessible via software catalogs, as in the Virtual Software House (VSH) project [SPP98]; other companies again could act as brokers to aid customers in finding and assembling appropriate applications. Also, for practically any type of firm, maintenance costs of employee workstations nowadays typically amount to the original hardware and software procurement costs many times over, particularly in terms of time spent by employees managing their workstations, therefore simplifying configuration and installation activities has the potential to greatly reduce information technology (IT) costs anywhere (in particular TCO: total cost of ownership of computers) [Leg97]. The Open Software Description (OSD) interoperability standard for software and dependency description recently proposed by Marimba and Microsoft [MM98] indicates that the commercial potential of some of these issues is now starting to be recognized.

Beyond implementing a system that can be introduced with minimum disruption to existing ones, an even better opportunity would be to define broadly accepted standards for such a system, which would allow it to be implemented on numerous platforms and in particular by several independent vendors, which would greatly increase chances of widespread adoption. Thus, inspired by the ideas presented in this thesis, standards and protocols would have to be worked out to precisely define the process modeling language including primitive operations, a minimally required set of distributable messages, etc. Ideally, a broad consortia such as Object Management Group (OMG) could carry out this work, e.g. defined as an extension to CORBA [OMG98], or preferably the World Wide Web Consortium (W3C) [WWW98a] or the Internet Engineering Task Force (IETF) [IET98] could conduct it and define the standards as WWW or Internet standards, laying emphasis on WWW integration of potential
process frameworks, which would be beneficial due to the popularity of the WWW. On the longer run, higher-level integration of process concepts perhaps in the Java Virtual Machine would be attractive, since such a system could be effortlessly and widely distributed as a constituent of WWW browsers.

5.5 CONCLUSIONS AND OUTLOOK

The “software crisis” is a software engineering issue that has been widely discussed in recent decades. Apart from the fact that it is a relative issue that need not be perceived as a crisis, in discussions most authors typically dwell on a solution to one specific problem, e.g. of a technical, organizational or management, or economical kind, disregarding other issues. We believe that solutions should address multiple issues, in particular taking into account evolving requirements, and should emphasize those issues where solutions are realistic, and while no comprehensive solution is imminent, significant contributions are nevertheless possible. Evolving issues include emerging distributed component technologies and expanding wide-scale Internet connectivity, due to which software is increasingly being developed as individual dependent components that are assembled to make “systems of systems” of software applications, and the components are more widely available, and development increasingly occurs in heterogeneous and widely distributed teams such as in virtual organizations.

Due to the higher numbers of software components and their increased availability, the technical issue of describing component interoperability, the organizational issues of managing components during development and deployment, and the technical, organizational and economical issue of supporting component reuse, are becoming increasingly important. Therefore, conventional software engineering requirements focusing on development of individual applications are enhanced to emphasize issues that concern interoperating components, both in distributed development and deployment.

Usage of formal methods during development is an essential technical issue to describe software interoperability on a high semantical level and to thereby support component reuse, yet their use is currently realistic only in certain special cases. Nevertheless, the DIPS process modeling language supports usage of formal methods during the process, and the associated CHIPS compiler-compiler provides an instance of a specific method. While current state-of-the-art manages component interoperability information only on the low interface level, managing it on the next-higher intermediary originator level is realistic, pragmatic and often sufficient, compared to the generally unattainably high semantic level. Originator level interoperability information of software components comprises information about software origins and
dependencies which is essential information for widely distributed loosely
coupled development of heterogeneous dependent software components, as
well as for their wide-scale deployment, and it contributes to increased
component reuse. This is provided by the DIPS process modeling
language, which manages originator level interoperability information
both during development and deployment of components, and thus
addresses both technical and organizational issues in software engineering.
Unlike most process modeling languages, the process focuses on the
structure of the evolving software product, providing coarse-grain
management of the product’s artifacts and their dependencies, essential for
wide-scale development and deployment interoperability, while leaving
fine-grain management up to individual tools used in the process with
little restriction.

Development and deployment interoperability also require not only local
management of interoperability information, but also inclusion of such
information from remote sites, i.e. linking of different distributed
processes besides distributed operation of individual processes. Thus, the
scope of process information is extended beyond individual processes,
since consumers (developers and users) of components require
interoperability information about such components. Similarly, the
temporal scope of processes is extended beyond development time to
provide interoperability information required by users during deployment
(installation and removal) of dependent components. This is achieved by
employing a common process modeling language that allows artifact
origin, dependency, evolution and other information to be registered
during development and made available to other developers as well as to
users to support deployment activities.

Simplicity, openness, interoperability, and scalability are key features
required by a system that is potentially to be adopted on a world-wide
scale, since disruption of existing operations must be minimized and
compatibility and vendor-independence are essential. These issues are
considered in the relatively simple concepts of the process modeling
language and its distributed framework, which allow existing tools to be
used and existing process models to be described or linked to, yet
provides a fully reflexive language that even allows metaprocess issues to
be described by itself, e.g. the environment and tools used during
development. The process modeling language integrates issues in a
coherent way that are normally modeled separately, including integration
of process definition, enactment and evolution in a single multi-
dimensional process structure, integration of product and process
evolution, integration of process and repository issues to simplify version
and configuration management without requiring explicit version
numbers, integration of development and deployment support, and
integration of metaprocess issues.
As presented earlier, there are dozens of projects and systems, both commercial and in research, that study or implement process-centered software engineering environments or frameworks, i.e. that deal with development issues, but only very few are starting to deal with issues such as integrating several processes, as is required e.g. in virtual organizations, or integrating repository and configuration management concepts in the process, which are essential in providing product and process information to other developers as well as for deployment purposes. Also, few projects are emerging only now that handle deployment issues, perhaps only one of which comprehensively covers the topic. However, there is apparently no project yet that integrates both development and deployment issues in a conceptually unified system. Nevertheless, it is clear that integration and interoperation of development and deployment issues has a great potential in facilitating work and reducing costs for both developers and for users. The DIPS process modeling language together with the distributed system that implements the language is thus different from other systems in several significant aspects. It is apparently the only system that integrates development and deployment issues in a unified concept and extends the scope of processes both beyond individual processes through process linking and beyond development time by supporting deployment activities, and it provides multiple integration issues in the language including repository and configuration management concepts, yet remains relatively simple, open, and interoperable with existing systems. While the DIPS project so far addresses mainly the open technical issues of artifact, process and organizational integration, it is clear that non-technical organizational issues, e.g. concerning software development cultures in virtual organizations, also deserve to be studied more intensively in future.

As presented in the section on contributions in the first chapter, the design and realization objectives have been met through the design of the process modeling language, the design of the distributed process framework, and the implementation of an experimental prototype of the framework for demonstration purposes. For the implementation of the language in the distributed framework, a modular design has been presented that separates development and deployment functionality, and a prototype implementation has been demonstrated. Different implementation possibilities have been discussed, from realistic usable implementations not requiring modifications to existing systems up to full-fledged integrated systems as a longer-term vision, including WWW integration, and intriguing commercial opportunities have also been discussed. It is envisaged that deployment will encompass management of user-site process bases that hold installed process aware software artifacts together with their corresponding process part copies, therefore software components will never leave the process context during their whole
Discussion

lifetime from development to installation and removal, alleviating problems experienced with current file-based software installations. This obviously constitutes a paradigm shift from current development and deployment practices, and it is possible that non-technical issues provide greater obstacles to its introduction than technical ones [Kuh96][Rog95][Coc97]. Nevertheless, a possible scenario is that vendor-independent standards are established that will allow such systems to be widely adopted. Also, the paradigm shift of development and deployment integration should be coupled with the paradigm shift of software ownership by introducing pay-per-use remuneration that supports component reuse economically, since both new schemes can benefit from each other.

While globally accepted standards could be inspired by the concepts presented in this thesis, much work remains to be done. While this thesis provides solutions to existing problems, demonstrating workable solutions already now, it additionally identifies numerous open questions and incomplete issues with the solutions presented, which is undoubtedly of equal importance if the ideas presented are to be applied on a larger scale. These issues are mentioned in context in their appropriate chapters, and although they should not pose insurmountable difficulties, provide ample opportunities for further research, e.g. for extensions to the process modeling language, more precise specifications of the language, incorporation of workspace concepts in the language, elaboration of development and deployment concepts in the framework, more extensive prototype implementations and discussions, formulation of standards, and others.
Leer - Vide - Empty
References


References


References


[DeM95b] T. DeMarco, Mad About Measurement, in [DeM95a], 1995, chapter 2, 11-44.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
</table>
References

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.


References


References


[MS97] T. Murer and D. Scherer, Structural Unity of Product, Process and Organization Form in the GIPSY Process Support Framework,
182 References


References


[Sch94] D. Scherer, TCP/IP for MacOberon and PowerMac Oberon, Oberon Newsletter, Institute of Computer Systems, ETH Zurich, Dec. 1994, 7.


[SH97] H. Singh and J. Han, Exploiting User-Group Relationships for Increasing Concurrency in Software Engineering Environments,


References


References


Curriculum Vitae

Feb. 3rd, 1968  Born in Muscat, Oman; Swiss nationality by inheritance
1973           English school, Melbourne, Australia
1973-1977      2 English schools, near Lisbon, Portugal
1977-1979      English school, Miri, Sarawak, Malaysia
1979-1986      German school, The Hague, The Netherlands
1986-1987      German school, Caracas, Venezuela
May 21st, 1987 “Abitur” exam
Sept. 23rd, 1987 Auxiliary exam to obtain “Maturität”
1987-1988      EPF Lausanne, computer science engineering studies; “Propédeutique I” exam
1988-1992      ETH Zurich, computer science engineering studies; minor: traffic planning and transport technology
Aug.-Nov. 1991 IBM Zurich Research Laboratory, 3 months’ practical work (computer science internship)
Nov. 20th, 1992 Conclusion of studies with “Dipl. Informatik-Ing. ETH” title (computer science engineering diploma)
1992-1998      TIK Laboratory, ETH Zurich, research and teaching assistant (projects ADAM and GIPSY) and doctoral studies
Nr. 1  
Kausale Beschreibung von Systemen  
Hans Peter Gisiger  

Nr. 2  
Compiling Sisal for the ADAM Architecture  
Srdjan Mitrovic  
1993, ISBN 3-7281-2051-0

Nr. 3  
Compiling the Behaviour Part of LOTOS  
Eric Emile Dubois  

Nr. 4  
The ADAM Architecture and its Simulation  
Olivier Maquelin  
1994, ISBN 3-7281-2113-4

Nr. 5  
GIPSY: Ein Ansatz zum Entwurf integrierter Softwareentwicklungssysteme  
Reto Marti  

Nr. 6  
Teilautomatische Herstellung von Codegeneratoren für Mikrokontroller  
Hans Bärffus  

Nr. 7  
SVOX: The Implementation of a Text-to-Speech System for German  
Christof. Traber  
1995, ISBN 3-7281-2239-4

Nr. 8  
Transparent Fault-Tolerance for Process Control Systems  
Thomas Siegrist  
1995, ISBN 3-7281-2242-4

Nr. 9  
Routing in Weittraum-Rechnernetzen  
Horst Hodel  

Nr. 10  
Execution Analysis of Real-Time Software Running on Heterogeneous Distributed Computer Systems  
Andreas Danuser  
Nr. 11
Prosodische Modifikation von Sprachsegmenten für die konkatenative Sprachsynthese
Beat Pfister
1995, ISBN 3-7281-2316-1

Nr. 12
A Framework for Dynamic Protocol Configuration
Thomas Peter Plagemann

Nr. 13
Execution Architecture of the Multithreaded ADAM Prototype
Philipp Färber

Nr. 14
Towards Computer Supported Audio Conferencing
Christoph A. Burkhardt

Nr. 15
Comparison of Classic and Hybrid HMM Approaches to Speech Recognition over Telephone Lines
Hans-Peter Hutter

Nr. 16
An Object Oriented Petri Net Approach to Embedded System Design
Robert Esser

Nr. 17
Theory of Evolutionary Algorithms and Application to System Synthesis
Tobias Blickle
1997, ISBN 3-7281-2433-8

Nr. 18
Eine objektorientierte Softwarearchitektur für Multimedia-Kommunikationssysteme
Daniel P. Ingold
1997, ISBN 3-906469-00-X

Nr. 19
Verifikation von Implementatioen abstrakter Datentypen für sicherheitskritische Systeme am Beispiel von Oberon
Daniel Schweizer
Nr. 20
Group and Session Management for Collaborative Applications
Erik Wilde

Nr. 21
Project GIPSY: Facing the Challenge of Future Integrated
Software Engineering Environments
Tobias Murer
1997, ISBN 3-906469-02-6

Nr. 22
Rapid-Prototyping eingebetteter Systeme
Ruedi Mattmann
1997

Nr. 23
A Multipoint Communication Architecture for End-to-End Quality of
Service Guarantees
Daniel Bauer
1997, ISBN 3-8265-2638-4

Nr. 24
On the Emulation of Impairments in ATM-Networks
Hans Kurmann
1997, ISBN 3-906469-03-4

Nr. 25
Emulation of LAN MAC Service for ATM Networks
Andreas X. Meier

Nr. 26
Controlling Segmental Duration in Speech Synthesis Systems
Marcel P. Riedi

Nr. 27
Ein Komponentenmodell für die Verbindungssteuerung in
Multimedia-Netzen
Stefan Frey

Nr. 28
The Architecture of an Interactive Multimedia Communication
System
Urs Röthlisberger